introduction to python for data science

Getting Started with Python, Pandas, and Matplotlib

NIK PIEPENBREIER

Table of Contents

About this Book

This book was developed to guide you, the reader, through the basics of Python as it relates to data science. It covers off what I feel you need to really get started with Python for Data Science.

It's a (sometimes) quirky and (a hopefully) enjoyable read. My aim was to hold your hands just enough, without getting in the way of trying things out yourself.

A Quick Note on Copyright

Published by Niklas Piepenbreier

About the Author

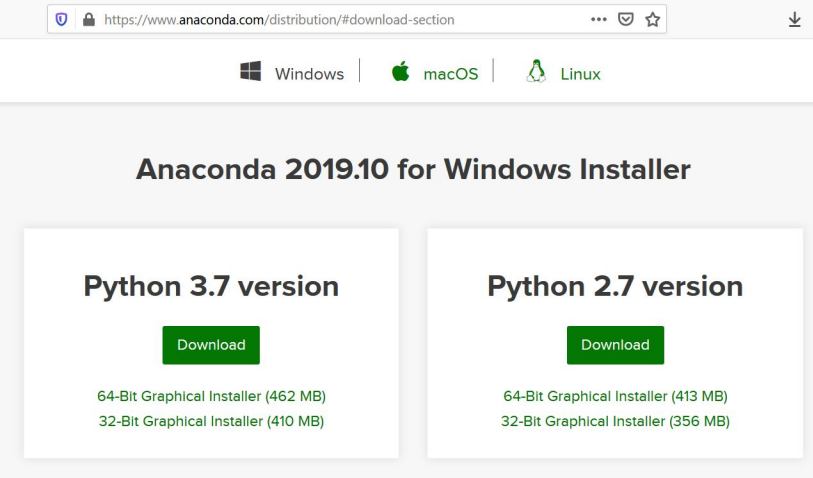
Nik Piepenbreier is the owner and founder of datagy - a company that seeks to help you find ways to activate your data. datagy was born in 2019 in Toronto, Ontario.

Nik has worked in the realm of data management and analytics for over seven years. He works with organizations to activate their data, build up data fluency, and communicate data more effectively.

What is Python?

Python is an open-source, interpreted language that's been around since the early 1990s. It has an extensive and helpful community behind it, constantly updating popular libraries. You can download Python directly from https://www.python.org/.

In this series, we'll use Python through a popular distribution called Anaconda which includes many data science libraries that we'll be covering off. You can download Anaconda directly from https://www.anaconda.com/.



See the image above. You can select the system you want to download Anaconda for. Be sure to download the latest version of Python (and ignore any Python 2.x version). We'll be using Python 3.x in this book.

Jupyter Notebooks

Throughout this series, we'll be using Jupyter Notebooks. As described on the Jupyter site, *the Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text*.

Jupyter Notebooks allow you to run code snippets individually while providing helpful context using Markdown text. More information on how

to format text using Markdown can be found here:
https://www.markdownguide.org/

Jupyter notebooks support over 40 different programming languages and support interactive outputs. Given they support many Python libraries, they are convenient to use for reproducible analysis.

Jupyter

Jupyter comes pre-installed with Anaconda. We'll be using Jupyter Notebooks in this series, but everything is equally possible in Jupyter Labs.

What's in a Notebook?

When you first create a new Notebook, you'll notice very little in it, but there's quite a bit running behind the scenes. Visually, the notebook combines and integrates code and its output into a single document, including the addition of markdown text, mathematical equations, and other media. When you first boot it up, it'll look like a generic text editor.

There are two main concepts that we want to cover here: (1) *kernel* and (2) *cell*.

Cell

Cells make up the body of a notebook and are the pieces that you actually see. A *cell* contains the text to be displayed or the code to be executed by the kernel. For our purposes, a cell can either be a code cell or a markdown cell - the primary difference is that a code cell has code to execute, while a markdown cell contains Markdown-formatted text. The cell you're reading right now is actually a markdown cell, and it doesn't execute any Python code.

In []:

```
#This is a code cell. It executes some Python code!
#Run the code by selecting the cell and hitting Shift and
Enter.
#Or by selecting the play button to the left.

print("You ran your first cell!")
```

Out[]:

```
You ran your first cell!
```

See how the empty brackets to the left of the code cell changed from [ ] to contain a number? That's the sequence in which your code runs. After you run your cell, the output (if there is one) is displayed below the cell.

Kernel

The *kernel* is the computational engine that executes the code in a notebook document. What is interesting about a kernel is that its state persists over time and across cells, i.e., it exists for the whole document, not for one particular cell.

Conventions Used in this Book

Code is used throughout this book! To help guide you through it and to emulate a Jupyter environment as best as possible, any code you put in is prefaced by an “In”:

In []:

```
print("hello world!")
```

Any code that results from that action by an “Out”:

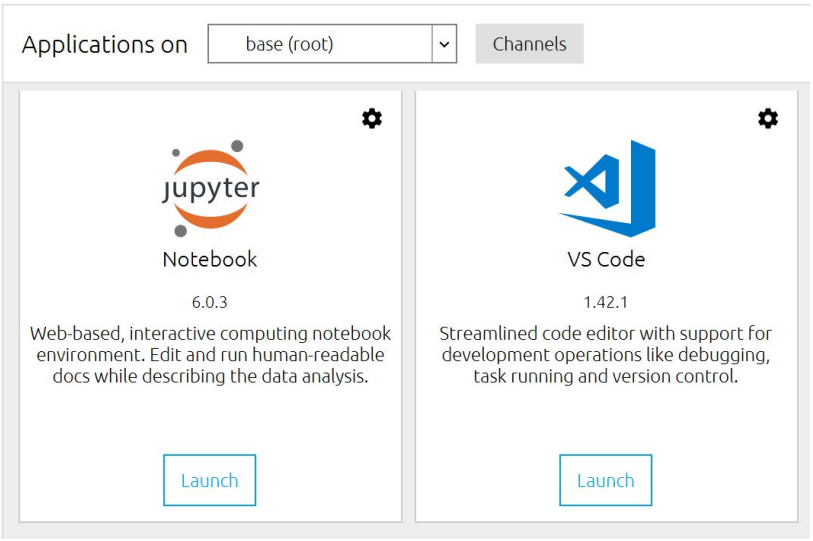
Out[]:

```
hello world!
```

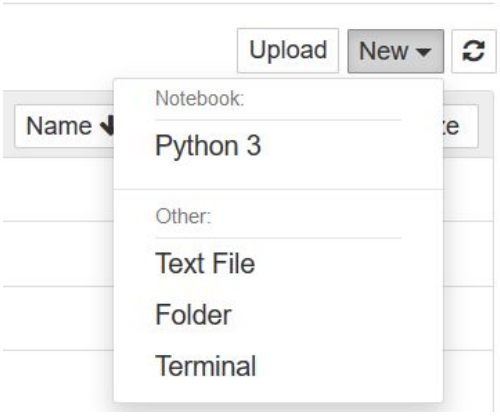
Creating a Notebook

Let's begin by creating a notebook.

Start by opening Anaconda. You'll see a window that looks like this:



Select New --> Python 3 Notebook and you're off!



Python Data Types

Now that you've created your first notebook, let's explore some of the data types that Python has to offer. You can always access a python data type by using the type() function on any object. We'll explore this is more detail in a moment.

Numeric Data Types

- **Integers** are positive or negative whole numbers (i.e., those without a fractional part)
- **Floats** are any real number with a floating-point representation

Let's create some of each of those data types and explore how they're represented. We can find out the type of an object by using the type() function. If we want to output the type of an object, say, object1, we could type *print(type(object1))*.

You can begin typing into a cell and then select the Run button in the toolbar to execute it (or use the Shift+Enter keyboard shortcut).

We can assign an object by simply typing the object's name, followed by an equal sign, and its value.

In []:

```
int1 = 1
float1 = 1.0
print("int1's type is: ", type(int1))
print("float1's type is: ", type(float1))
```

Out[]:

```
int1's type is:  <class 'int'>
float1's type is:  <class 'float'>
```

Boolean Data Types

Booleans are data that equate to either a True or a False

In []:

```
bool1 = True
print("bool1's type is: ", type(bool1))
```

Out[]:

```
bool1's type is:  <class 'bool'>
```

Booleans can be helpful to equate items. Let's see how to compare our previous integers and floats to see if Python recognizes them as the same value.

In []:

```
float1 == int1
```

Out[]:

```
True
```

Notice that we used 2 equal signs. A single equal sign is used to *assign* values, while two equal signs *compare* the two values. A thing to note here is that while the two values are different types, their value is still the same.

String Data Types

Strings are collections of one or more characters put into single, double, or triple quotes. Strings are immutable but carry an interesting structure that we'll assess in the next section.

In []:

```
string1 = 'Hello, World!'
print(string1)
```

Out[]:

```
Hello, World!
```

Something we should point out here is that we *technically* don't need to include the print() command at the bottom of each of these cells. Notebooks will by default return the last object (if it returns anything), as we demonstrate below.

In []:

```
string1
```

Out[]:

```
'Hello, World!'
```

Python Data Structures

Now that we have an understanding of the different data types that exist in Python, let's explore some of the different data structures that are available.

- **Lists** are ordered collections of data items that can be heterogeneous (but don't need to be) and are placed into square brackets. Heterogeneous means that they can contain items of different data types. They are mutable (changeable) and iterable.
- **Tuples** are ordered collections that can be heterogeneous (but don't need to be) and are placed into parentheses. They are not mutable and are helpful when items are used for reference and should not be changed.
- **Dictionaries** are unordered sets of key:value pairs (with a requirement that keys are unique within a dictionary), and are placed into curly braces.

In []:

```
list1 = [1, 2, 'hello']
tuple1 = (3, 4, 'goodbye')
dictionary1 = {'make': 'Toyota', 'year': 2020}
```

Python Data Structure Indexing

Python allows us to retrieve data structure items by calling their *index*. By convention, most indices in Python begin at index 0. An index is called by using square brackets following the structure name.

There's a number of conventions that would be helpful to know:

- Indices begin at 0
- Negative indices begin at -1 and call the last item (this is particularly helpful when you don't know the number of items that exist, or if this number changes)
- Structures can be sliced using the : character (e.g., structurename[1:3] will return the second and third item
- Slices are exclusive, meaning the slice [1:3] would return items 2 and 3, but not 4

- Slices can omit numbers to include everything from either beginning to end (e.g., [:4] would return all items up to the fourth one, but not the fifth one)

Let's try this out on the list and tuple we created above.

In []:

```
list1[0] #return the first item
```

Out[]:

```
1
```

In []:

```
tuple1[:2] #return all items up to, but not including, the
third one
```

```
(3, 4)
```

In []:

```
list1[-1] #return the last item
```

Out[]:

```
'hello'
```

The difference here is with dictionaries. Remember, dictionaries are unordered, so there's no point in calling an index number. Instead, an item is called by its *key*:

In []:

```
dictionary1['make']
```

Out[]:

```
'Toyota'
```

We mentioned earlier that strings have interesting properties. In some ways, strings represent data structures. As such, we can retrieve parts of strings by calling their indices.

In []:

```
string1[:3]
```

Out[]:

```
'Hel'
```

Updating Data Structure Values

Recall from above that we mentioned that lists and dictionaries are mutable. This means that we can directly update values within each structure. This will not work for strings or for tuples.

In []:

```
list1[0] = 4
list1
```

Out[]:

```
[4, 2, 'hello']
```

In []:

```
dictionary1['year'] = 2019
dictionary1
```

Out[]:

```
{'make': 'Toyota', 'year': 2019}
```

For Loops and Iterating Over Objects

For loops allow us to iterate over a sequence (including, but not limited to, lists, tuples, dictionaries, and strings). The keyword *for* allows us to iterate over objects and execute a number of statements. Let's try this using the list we generated above.

In []:

```
for item in list1:
   print(item)
```

Out[]:

```
4
2
hello
```

What this for loop does is iterate over each item in the list, and for each item execute the print command. The fact that we called each object "item" doesn't matter - we could have just as well used anything else. For example:

In []:

```
for x in list1:
   print(x)
```

Out[]:

```
4
2
hello
```

We can make these loops more powerful by adding in conditional statements. For example, if we wanted to print True if the item equaled 'hello', and False otherwise, we could write:

In []:

```
for item in list1:
   if item == 'hello':
       print('True')
   else:
       print('False')
```

Out[]:

```
False
False
True
```

We can also ask Python to run through a set of code a number of times using the range() function. The range() function returns a sequence of numbers, starting at 0 (by default), and incrementing by 1 (by default), and ends at a specific number.

In []:

```
for i in range(11):
   print(i)
```

Out[]:

```
0
1
2
3
4
5
6
7
8
9
10
```

Notice here that while we specified range(11), the printing ended at 10. The range function goes up to, but doesn't include, the last integer.

We can extend this by using adding additional parameters. The range() function accepts: range(start, stop, increment by)

In []:

```
for i in range(0,10,2):
   print(i)
```

Out[]:

```
0
2
4
6
8
```

Here we asked Python to print every item beginning at 0, incrementing by 2, and ending at (but not including) 10.

To apply for-loops a little more, let's take a new list that includes the values from 1 through 5. We want to create a list that has the squares of each of these numbers. We can easily add to a list by using the .append() function.

We'll also create an empty list called squares, so we have something to append to.

In []:

```
original = [1,2,3,4,5]
squares = []

for i in original:
   squares.append(i**2)

squares
```

Out[]:

```
[1, 4, 9, 16, 25]
```

This is where the power of for loops really shines through. Instead of writing the function multiple times, we asked Python to iterate through the list for us.

We can extend this even further with if/else statements. Let's use the same original list and add even numbers to a new list named even and odd numbers to a list odd. We'll use the modulus (%) operator for figure out of a number is even or odd. The modulus operator returns the remainder of a division between two values. For example, 5%2 would return 1.

If an even number is divided by 2, its remainder should be equal to 0. If it's equal to anything else, the number is odd.

In []:

```
even = []
odd = []

for i in original:
   if i % 2 == 0:
       even.append(i)
   else:
       odd.append(i)

print('Odd:', odd)
print('Even:', even)
```

Out[]:

```
Odd: [1, 3, 5]
Even: [2, 4]
```

For loops allow us to easily create new objects based on existing objects. They also provide easy ways to apply functions to a given set of data.

List and Dictionary Comprehensions

Generating lists in Python can be a pain sometimes, especially when working with existing lists or items that have patterns. Because of this, list comprehensions provide a simple way to generate lists. They are frequently used to make new lists where every element is the result of some operation applied to another iterable object.

List comprehensions have concise terminology for replacing cumbersome for-loops for iterating over other items. Let's look at an example. We'll create a list of squares from the number 0 through 5.

Using a for loop, we would write:

In []:

```
squares = []
for i in range(6):
   squares.append(i**2)
squares
```

Out[]:

```
[0, 1, 4, 9, 16, 25]
```

By writing a for loop, we used 3 lines. This was a very basic example, but it still required some menial set up, such as generating an empty list.

Let's now take this on using a list comprehension:

In []:

```
squares = [i**2 for i in range(6)]
squares
```

Out[]:

```
[0, 1, 4, 9, 16, 25]
```

Here, we were able to accomplish the same thing using just 1 line of code.

We can think of this as: for each item in this list, evaluate an expression and apply it to the list indicated.

We can also add conditions directly into list comprehensions to make them more useful. Let's begin by identifying every odd number in the range from 30-50.

In []:

```
odd = [i for i in range(30, 51) if i%2 == 1]
odd
```

Out[]:

```
[31, 33, 35, 37, 39, 41, 43, 45, 47, 49]
```

Let's do something a little more complicated again: we'll add an else statement to a list comprehension. Let's generate a list that says if a number is more or less than 5, in the range of 1-10.

In []:

```
lessmore = ["less" if i < 5 else "odd" for i in range(11)]
lessmore
```

Out[]:

```
['less',
 'less',
 'less',
 'less',
 'less',
 'odd',
 'odd',
 'odd',
 'odd',
 'odd',
 'odd']
```

Let's nest some if statements. We'll look at whether a number if a divisible by two and by five, in a list of 1-100.

In []:

```
twofive = [i for i in range(1, 101) if i % 2 == 0 if i % 5 == 0]
twofive
```

Out[]:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

List comprehensions cut down on the amount of code you have to write, but it's recommended to avoid using list comprehensions once they get too complicated. For example, nesting too many if statements or if-else statements can make the code hard to read, especially later on.

It's also important to note that any list comprehension can be written as a for loop, but not every for loop can be a list comprehension. If you're scratching your head about how to turn a for-loop into a comprehension, it may be too complex for anyone reading your code later on (probably future-you!).

Introduction to Numpy

Numpy is a math library for python that allows us to take on a number of amazing math tasks easily, without having to write our own functions. It's also an important precursor to understanding Pandas, which leverages a lot of Numpy capabilities. Let's begin by importing numpy. We'll as "as np" to create an alias for numpy, to avoid having to type the full library name when we reference it.

Importing a library as is simple as typing:

In []:

```
import numpy as np
```

Numpy stores data, similar to Python's lists, in *arrays*. In fact, arrays and lists look very similar, but declaring an array as a numpy array allows us to interact with it in unique ways.

In []:

```
array1 = np.array([0,2,4,6,8,10])
array1
```

Out[]:

```
array([ 0,  2,  4,  6,  8, 10])
```

We can explore the data type as we did before by using the type() function:

In []:

```
type(array1)
```

Out[]:

```
numpy.ndarray
```

Acting on Arrays

Numpy arrays allow us to interact with the data stored within them. For example, let's create another array, and we'll jumble the numbers a little. We can then call the .sort() function to sort the values numerically.

In []:

```
array2 = np.array([4,2,3,6])
array2
```

Out[]:

```
array([4, 2, 3, 6])
```

In []:

```
array2.sort()
array2
```

Out[]:

```
array([2, 3, 4, 6])
```

We can also act on the values more directly, such as by multiplying them.

In []:

```
array2 = array2*2
array2
```

Out[]:

```
array([ 4,  6,  8, 12])
```

Other functions such as adding, subtracting, etc. are available as well.

Array Shapes and Multi-Dimensional Arrays

Arrays are also commonly referred to as *matrix* or *vector*. A matrix, for those familiar with linear algebra, can take different dimensions. For now,

let's focus on 1-dimensional arrays like we just generated. Let's find out what the *shape* of this array is, using the .shape method.

In []:

```
array1.shape
```

Out[]:

```
(6,)
```

We may have expected this to return a value of (6,1), rather than (6,). In fact, right now, the data *isn't* a 6x1 matrix, but we can reassign it to one.

In []:

```
array1.reshape(1,6)
```

Out[]:

```
array([[ 0,  2,  4,  6,  8, 10]])
```

Notice now, that the array we returned has two sets of square brackets. In essence, this created a *list of lists*. In fact, a multi-dimensional matrix is simply a list of lists. Let's now create a 3x3 matrix by declaring it directly as a numpy array.

In []:

```
matrix33 = np.array([
   [1,2,3],
   [4,5,6],
   [7,8,9]
])
matrix33
```

Out[]:

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

We can double-check the shape of the array by using the .shape method again.

In []:

```
matrix33.shape
```

Out[]:

```
(3, 3)
```

Let's close our discussion on Numpy here. Numpy has many applications outside of this, but we want to focus on Pandas as it has many utilities that simplify data science work.

Introduction to Pandas

Pandas is a powerhouse for data scientists using Python. It offers up unique ways to explore data, clean data, gain insights, and present analyses. Let's begin by importing Pandas. This time we'll use the convention pd as an alias.

In []:

```
import pandas as pd
```

The magic of Pandas begins in its data structures, *series* and *dataframes*.

A **series** is a 1-dimensional array, similar to those we explored in the Numpy section. The key difference here is that each element is labeled and the individual elements of the array can be accessed via its label.

An **array** is similar to an Excel worksheet, in that there exist rows and columns, and elements can be accessed by the column and row labels. Column names are referred to as *column index* and row names are referred to as *index* or *indices*.

These two structures represent the core models of Pandas and are the first step in getting started with data analysis.

Dataframes can be generated directly with Pandas or can be read in from a variety of formats, including .csv and .xlsx files. Let's use a freely available dataset and read it in. By convention, a dataframe is given the name *df*.

Generating a Dataframe

Let's begin by creating a dataframe. datagy has created a dataset on a number of athletes, playing for various cities in North America. To import the dataset, we'll use the pd.read_excel() function. By default, your only requirement will be a link to the data, but you can specify other parameters such as separator, or ignoring headers.

You can load data directly from a URL or store the file somewhere else. If the file is stored in the same directory as the notebook, simply call the file name.

The dataset can be found on our Github page here: https://github.com/datagy/Intro-to-Python/raw/master/sportsdata.xls .

In []:

```
df =
pd.read_excel('https://github.com/datagy/Intro-to-Python/raw/ma
ster/sportsdata.xls')
```

Exploring the Data

We have now generated a dataframe named *df*, which contains all the data in the Excel file at that URL. Pandas provides a number of useful functions that allow us to gain insight into the data.

Let's begin exploring the data. We can print the first five rows of the data by using the .head() function.

In []:

```
df.head()
```

Out[]:

	ID	First_Name	Last_Name	Height in Feet Inches	...	City	League
0	1	Jacob	Miller	6'3"	...	Atlanta	National
1	2	Santiago	Parker	6'1"	...	Edmonton	Championship
2	3	Nolan	Morris	6'1"	...	Ottawa	Championship
3	4	Michael	Brooks	5'10"	...	St. Louis	National
4	5	Andrew	Watson	5'11"	...	San Diego	Championship

The .head() function by default returns the first five rows. Say we wanted to return the first three rows - we could use df.head(3).

In []:

```
df.head(3)
```

Out[]:

	ID	First_Name	Last_Name	Height in Feet Inches	...	City	League
0	1	Jacob	Miller	6'3"	...	Atlanta	National
1	2	Santiago	Parker	6'1"	...	Edmonton	Championship
2	3	Nolan	Morris	6'1"	...	Ottawa	Championship

Similarly, we can explore the last few rows using the .tail() method:

In []:

```
df.tail()
```

Out[]:

	ID	First_Name	Last_Name	Height in Feet Inches	...	City	League
4995	4996	Maverick	Phillips	5'10"	...	Salt Lake City	National
4996	4997	Axel	Thomas	6'1"	...	San Francisco Bay Area	Championship
4997	4998	Jonathan	Garcia	6'3"	...	San Francisco Bay Area	Championship
4998	4999	Jose	Lee	5'11"	...	Vancouver	National
4999	5000	Henry	Foster	6'1"	...	Syracuse	Championship

Working with Columns

Let's take a quick look at how columns work in a Pandas dataframe. We can use the .columns method on a dataframe to quickly generate a list of the column names.

In []:

```
df.columns
```

Out[]:

```
Index(['ID', 'First_Name', 'Last_Name', 'Height in Feet
Inches',
       'Weight in lbs', 'City', 'State', 'Country', 'Foot',
'Age', 'Photo',
       'Wage', 'Value', 'Website', 'League'],
      dtype='object')
```

Say we wanted to clean up the column names. We could use the .rename() method to easily rename columns using a dictionary. Let's rename only a few.

In []:

```
columns_rename = {
   'Height in Feet Inches': 'height',
   'Last_Name': 'lastname'}

df.rename(columns = columns_rename, inplace=True)
#we use inplace=True to tell Python to make these changes
directly

df.columns
```

Out[]:

```
Index(['ID', 'First_Name', 'lastname', 'height', 'Weight in
lbs', 'City',
       'State', 'Country', 'Foot', 'Age', 'Photo', 'Wage',
'Value', 'Website',
       'League'],
      dtype='object')
```

Conventionally, dataframe columns aren't written in uppercases and don't contain spaces. We could simply create a dictionary to replace all character names, or we could apply some fancier Python to make this easier on ourselves. We'll begin by creating a list comprehension that does the hard work for us. As we explained earlier, list comprehensions give easy syntax for for-loops, when working with lists.

In []:

```
df.columns = [col.lower() for col in df]
df.columns
```

Out[]:

```
Index(['id', 'first_name', 'lastname', 'height', 'weight in
lbs', 'city',
       'state', 'country', 'foot', 'age', 'photo', 'wage',
'value', 'website',
       'league'],
      dtype='object')
```

We can also strip out the spaces and replace them with underscores using the .replace() function:

In []:

```
df.columns = [col.replace(' ', '_') for col in df]
df.columns
```

Out[]:

```
Index(['id', 'first_name', 'lastname', 'height',
'weight_in_lbs', 'city',
       'state', 'country', 'foot', 'age', 'photo', 'wage',
'value', 'website',
       'league'],
      dtype='object')
```

When we looked at the header of the dataframe earlier, it was easily noticeable that the id column was a replication of the index column. Let's use Pandas to drop this column. We can also drop some columns that we know we won't use for data analysis, such as the photo and website.

In []:

```
df.drop(columns = ['photo', 'website'], axis = 1, inplace=True)
df.head()
```

Out[]:

	id	first_name	lastname	height	...	value	league
0	1	Jacob	Miller	6'3"	...	$1.2M	National
1	2	Santiago	Parker	6'1"	...	$828.8K	Championship
2	3	Nolan	Morris	6'1"	...	$55.7M	Championship
3	4	Michael	Brooks	5'10"	...	$1M	National
4	5	Andrew	Watson	5'11"	...	$447.7K	Championship

Gaining Insight into the Dataframe

In bold across the top row are the *column names* or column indices. Each of these can be referenced when slicing into the data, which we'll explore a little later.

Along the left-most column are numbers 0, 1, 2... These represent the *indices* or row numbers. Note that these numbers didn't exist in the actual dataset, but were created when we generated the dataframe. Unless we force Pandas to reassign them, they will remain in place, even if we filter the data.

We can ask Pandas to provide high-level insight into the data by using a number of important functions. Let's begin getting info on the columns by using the .info() function.

In []:

```
df.info()
```

Out[]:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 13 columns):
```

```
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   id              5000 non-null   int64
 1   first_name      5000 non-null   object
 2   lastname        5000 non-null   object
 3   height          4977 non-null   object
 4   weight_in_lbs   5000 non-null   object
 5   city            5000 non-null   object
 6   state           4990 non-null   object
 7   country         4993 non-null   object
 8   foot            5000 non-null   object
 9   age             5000 non-null   int64
 10  wage            5000 non-null   object
 11  value           4982 non-null   object
 12  league          5000 non-null   object
dtypes: int64(2), object(11)
memory usage: 507.9+ KB
```

The .info() function tells us a number of things:

- the number of entries (and their indices),
- the number of columns
- a list of the column name, # of non-null objects, and the data type within the column,
- a list of data types, and
- the memory usage of the dataframe

This function allows us to explore whether or not there is missing data, how data is structured, etc. This allows us to take steps towards data cleaning, allowing us to ensure that the data is as clean as possible before we begin analyzing it.

While the .info() method provides information onto the number of non-null objects, this can be confusing to read sometimes. What if we wanted a quick representation of the number of null objects? We could chain a statement that computes, per row, the number of null values:

In []:

```
df.isnull().sum()
```

Out[]:

```
id                0
first_name        0
lastname          0
height           23
weight_in_lbs     0
city              0
state            10
country           7
foot              0
age               0
wage              0
value            18
league            0
dtype: int64
```

As you can imagine, the .isnull() evaluates all null values, while .sum() adds these up. .isnull() returns a dataframe of boolean types (True for null, False for non-null). A True is assigned a value of 1, and a False is assigned a value of 0.

We'll explore how to deal with these missing values later on, when we cover Conditionally Selecting Data with Pandas.

We can gain even more insight into the actual data by using the .describe() function.

In []:

```
df.describe()
```

Out[]:

	id	age
count	5000	5000
mean	2500.5	25.0122
std	1443.520003	5.013849
min	1	4
25%	1250.75	22
50%	2500.5	25

75%	3750.25	28
max	5000	43

For each *numerical*-type column, the .describe() function returns a number of valuable statistics.

Indexing Data with Pandas

Similar to how we indexed lists and other objects earlier, we can also index a dataframe. Recall that a dataframe has both a column and a row index. As such, when we wish to retrieve a particular value, we need to reference both of these indices.

Let's say we want to return the first five rows of the first column.

In []:

```
df.iloc[0:5, 0]
```

Out[]:

```
0    1
1    2
2    3
3    4
4    5
Name: id, dtype: int64
```

In the example above, we use square brackets as we did earlier, and we specify first the rows we wish to retrieve, then the columns. We prefix the reference with .iloc, which returns an *integer-location*. This is handy when we know the location of the objects we want to return.

One of the powerful pieces available in Pandas is the ability to have *named indices*. We can access these using the .loc method. Say we wanted to return items 1-5 of the age column. We don't need to know what position that column is in, but rather just its name.

In []:

```
df.loc[:5, 'age']
```

Out[]:

```
0    27
1    31
2    27
3    28
4    27
5    22
Name: age, dtype: int64
```

Pandas Series Objects

Another method of querying Pandas dataframes is to use the series object, which allows us to retrieve an entire column much more easily. We can call the entire column by placing its name into square brackets, similar to querying a dictionary.

In []:

```
df[['age']]
#Note: this would work with a single set of square brackets,
#but the double bracket makes output nicer in Jupyter
```

Out[]:

	age
0	27
1	31
2	27
3	28
4	27
...	...
4995	29
4996	18
4997	32
4998	22
4999	28

We can also select columns by using df.age. Let's stay away from this method as it can cause issues when:

1. Columns have spaces
2. Columns use reserved keywords (like info)

Using this, you can even query lists of columns, regardless of the order in which they exist in the dataframe:

In []:

```
df[['id', 'age']]
```

Out[]:

	id	age
0	1	27
1	2	31
2	3	27
3	4	28
4	5	27
...	...	...
4995	4996	29
4996	4997	18
4997	4998	32
4998	4999	22
4999	5000	28

Pandas Methods

There are a number of other helpful methods. Similar to the .describe() method, we can apply some useful methods, either to a series or to an entire dataframe. Applying some methods to a dataframe will display results for all numerical columns.

In []:

```
df['age'].mean()
```

Out[]:

```
25.0122
```

In []:

```
df.mean()
```

Out[]:

```
id      2500.5000
age       25.0122
dtype: float64
```

Other helpful methods include:

- df.count()
- df.max()
- df.min()
- df.median()
- df.std()

Selecting and Dealing with Null Values

We can also filter our data frames using Pandas, by applying conditions (or filters) onto the data. This can be helpful if we want to select null data, for example, to see where data issues might lie.

Null values cause problems in analysis and taking a look at these during your exploratory data analysis is helpful to figure out how to deal with them. Without going into the statistics behind it, we have two main options (1) impute data for missing values, or (2) removing those records.

Let's begin by selecting out only values that have null values. We'll regenerate the summary on columns that have null values.

In []:

```
df.isnull().sum()
```

Out[]:

```
id                0
first_name        0
lastname          0
height           23
weight_in_lbs     0
```

```
city              0
state            10
country           7
foot              0
age               0
wage              0
value            18
league            0
dtype: int64
```

Typically, if we wanted to deal with null values, we have two options: (1) impute the value using some measure, or (2) delete the records. We don't really know what influences the height of a player, or their value, so it may be prudent to delete those records. Let's delete the records for height and wage, as they represent on a small fraction of our dataset.

In []:

```
df.dropna(subset = ['height', 'value'], axis = 0, how = 'any',
inplace = True)
```

We've introduced a bit of new terminology with the .dropna() method. We use:

- **subset** denotes which columns we want to drop
- **axis = 0** to denote we want to drop rows (1 would indicate columns)
- **how = 'any'** to denote that any cells are NaN
- **inplace = True** to make the change to the data frame itself

With State and Country, however, we know which state and country a city is in, and we are able to impute those records based on the City column. Let's explore the records that have blank states first.

In []:

```
statenull = df[df['state'].isnull()]
statenull
```

Out[]:

	id	first_name	lastname	height	weight_in_lbs	city	state	country	...
4251	4252	Hunter	Hernandez	6'1"	192lbs	Atlanta	NaN	USA	...
4332	4333	Alexander	Rodriguez	5'11"	192lbs	Baltimore	NaN	USA	...
4349	4350	Benjamin	Long	6'0"	192lbs	Detroit	NaN	USA	...
4358	4359	Mason	Young	6'1"	192lbs	Atlanta	NaN	USA	...
4381	4382	Ian	Harris	5'9"	192lbs	Denver	NaN	USA	...
4394	4395	Asher	Reed	5'7"	192lbs	Winnipeg	NaN	Canada	...
4470	4471	Jason	Martinez	6'2"	192lbs	Newark	NaN	USA	...
4524	4525	Logan	Brooks	6'2"	192lbs	Raleigh	NaN	USA	...
4527	4528	Carson	Lee	6'0"	192lbs	Green Bay	NaN	USA	...
4547	4548	Nolan	Flores	5'11"	192lbs	Salt Lake City	NaN	USA	...

We now know which records have missing values for the state variable. We can either manually map in those values, or we can create a dictionary that does the heavy lifting for us. In this case, as there are only ten records, it may not be a lot of work to do this manually, but to do this for hundreds (or thousands) of records could cause a big headache.

We can automate this mapping using a for loop that iterates over complete records. Let's take this on step by step.

First, we'll want to create a smaller dataframe that includes only complete records of city, state, and country and removes all duplicates. We apply the .copy() method to create a *deep* copy of the dataframe, that removes any tie to the original dataframe. We then apply the .dropna() method that will remove the missing values. Finally, we apply the .drop_duplicates() method that, as you may be able to guess, drops any duplicate records.

In []:

```
location = df[['city', 'state', 'country']].copy()
location.dropna(inplace=True)
location.drop_duplicates(inplace=True)
```

```
location.info()
```

Out[]:

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 50 entries, 0 to 412
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   city     50 non-null     object
 1   state    50 non-null     object
 2   country  50 non-null     object
dtypes: object(3)
memory usage: 1.6+ KB
```

We can see above that we now have a new dataframe that has fifty unique records. We can now turn this into a dictionary to allow us to map this over more readily. We'll chain a number of arguments here to create our dictionaries. Let's explore these in a bit of detail:

1. *pd.Series* generates a list
2. The list is filled with values from either the state or the country column
3. The index of each list is identified as the corresponding city
4. *.to_dict()* converts the list to a dictionary

In []:

```
state_dict =
pd.Series(location['state'].values,index=location['city']).to_d
ict()
country_dict = pd.Series(location['country'].values, index =
location['city']).to_dict()
```

We can now extract values directly from the dictionaries by accessing their keys:

In []:

```
print(state_dict['Atlanta'])
print(country_dict['Atlanta'])
```

Out[]:

```
Georgia
USA
```

Now we're in a position to map these values into the missing values for both the *state* and *country* columns. We'll use the .fillna() method available in Pandas to fill these values. Let's see what that looks like:

In []:

```
df['state'] = df['state'].fillna(df['city'].map(state_dict))
```

Let's see if this worked before we explain what we did. We'll pull the records for one of the cells we saw was missing data earlier.

In []:

```
df.loc[4253]
```

Out[]:

```
id                        4254
first_name               Wyatt
lastname                 Rivera
height                     5'8"
weight_in_lbs           198lbs
city                   Orlando
state                  Florida
country                    USA
foot                     Right
age                         26
wage                      $121K
value                     $1.4M
league           Championship
Name: 4253, dtype: object
```

It looks like this worked quite well! Let's explore the work we did above in a bit of detail:

1. We use the .fillna() method to ask Pandas to fill any missing values in the state column
2. We use the .map() method to map in any values based on the city column

This, in many ways, is similar to mapping missing values using a VLOOKUP or INDEX-MATCH in Excel.

Let's apply this to the *country* column as well.

In []:

```
df['country'] =
df['country'].fillna(df['city'].map(state_dict))
```

Let's re-run our code from earlier now to make sure we no longer have any missing data.

In []:

```
df.isnull().sum()
```

Out[]:

```
id              0
first_name      0
lastname        0
height          0
weight_in_lbs   0
city            0
state           0
country         0
foot            0
age             0
wage            0
value           0
league          0
dtype: int64
```

Conditionally Selecting Data with Pandas

We may want to select certain portions of our dataframe in Pandas to investigate a particular subsection of data.

For example, we may want to select only players from Atlanta.

In []:

```
atlanta = df[df['city'] == 'Atlanta']
atlanta.head()
```

Out[]:

	id	first_name	lastname	height	weight_in_lbs	city	state	country	foot	age	wage	value	league
0	1	Jacob	Miller	6'3"	192lbs	Atlanta	Georgia	USA	Right	27	$97K	$1.2M	National
8	9	Greyson	Martin	6'1"	204lbs	Atlanta	Georgia	USA	Left	27	$83K	$666.4K	National
24	25	Isaac	Brown	6'2"	192lbs	Atlanta	Georgia	USA	Right	22	$137K	$1.4M	National
29	30	Greyson	Thomas	6'0"	186lbs	Atlanta	Georgia	USA	Left	20	$112K	$1.2M	National
64	65	Caleb	Green	6'1"	222lbs	Atlanta	Georgia	USA	Right	31	$122K	$976K	National

Similar to the .isnull() method, this sets a boolean statement to the entire dataframe and returns only the records to which the value equates to True.

You may notice in the head that we printed, that the index remains constant with what exists in our original dataframe. This is because what we have done here is to create a representation of that dataframe. If we make changes to the data here, it will also be reflected in the other dataframe.

We can also select based on multiple conditions. Say we wanted to select only right footed players from Atlanta:

In []:

```
atlanta_right = df[(df['city'] == 'Atlanta') & (df['foot'] ==
'Right')]
atlanta_right.head()
```

Out[]:

	id	first_name	lastname	height	weight_in_lbs	city	state	country	foot	age	wage	value	league
0	1	Jacob	Miller	6'3"	192lbs	Atlanta	Georgia	USA	Right	27	$97K	$1.2M	National
24	25	Isaac	Brown	6'2"	192lbs	Atlanta	Georgia	USA	Right	22	$137K	$1.4M	National
64	65	Caleb	Green	6'1"	222lbs	Atlanta	Georgia	USA	Right	31	$122K	$976K	National
91	92	Angel	Lopez	5'8"	198lbs	Atlanta	Georgia	USA	Right	32	$113K	$904.8K	National
130	131	Joshua	Gonzales	6'2"	201lbs	Atlanta	Georgia	USA	Right	22	$77K	$766K	National

We can also use comparators like greater than or less than to filter dataframes. For example, if we wanted to count the number of players over the age of 30, we could write the following:

In []:

```
df[df['age'] > 30]['id'].count()
```

Out[]:

```
705
```

Cleaning up Data

It's often said that a data scientist will spend 80% of their time cleaning data. Based on what we've seen, it might appear that the dataset is quite clean, but there are a number of improvements we need to make to the data. Let's begin this journey by looking at the data types of each of the columns.

In []:

```
df.info()
```

Out[]:

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4959 entries, 0 to 4999
Data columns (total 13 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   id             4959 non-null   int64 
 1   first_name     4959 non-null   object
 2   lastname       4959 non-null   object
 3   height         4959 non-null   object
 4   weight_in_lbs  4959 non-null   object
 5   city           4959 non-null   object
 6   state          4959 non-null   object
 7   country        4959 non-null   object
 8   foot           4959 non-null   object
 9   age            4959 non-null   int64 
 10  wage           4959 non-null   object
 11  value          4959 non-null   object
 12  league         4959 non-null   object
dtypes: int64(2), object(11)
memory usage: 702.4+ KB
```

Cleaning Wage Data

What this tells us is that some of the numeric columns are labeled as objects. Because of this, we'll run into issues if we try to do things like calculate averages. If we were to calculate the average wage, by using **df['wage'].mean()**, Jupyter would return an error message.

Let's see what the data in that column looks like.

In []:

```
df['wage'].head()
```

Out[]:

```
0     $97K
1    $104K
```

```
2    $6194K
3     $114K
4      $41K
Name: wage, dtype: object
```

Hopefully, each wage is listed in $. This will make our work significantly easier, than having to convert between currencies. Let's assume for a moment that the currency is the first character in each cell. We can identify the unique values using some chaining.

In []:

```
df['wage'].str[0].unique()
```

Out[]:

```
array(['$'], dtype=object)
```

Let's break down what we've done here:

- **df['wage']** selects the wage column
- **.str[0]** selects the column as type string and then only selects the first value
- **.unique()** returns the unique values found in the resulting subset of data

As the only unique currency that exists in the wage column is the Euro, we can remove that character.

In []:

```
df['wage'] = df['wage'].str.replace('$', '')
```

Let's see what the data looks like now.

In []:

```
df['wage'].head()
```

Out[]:

```
0      97K
1     104K
2    6194K
3     114K
4      41K
Name: wage, dtype: object
```

Ok, so we've made progress, but we also want to make sure that all the values are in thousands. Let's assume that that value is always listed last, and redo what we did earlier.

In []:

```
df['wage'].str[-1].unique()
```

Out[]:

```
array(['K'], dtype=object)
```

Similar to above, we can remove all mentions of K.

In []:

```
df['wage'] = df['wage'].str.replace('K','000')
df['wage'].head()
```

Out[]:

```
0      97000
1     104000
2    6194000
3     114000
4      41000
Name: wage, dtype: object
```

Now we'll change the data type to be an integer.

In []:

```
df['wage'] = df['wage'].astype(int)
df['wage'].head()
```

Out[]:

```
0      97000
1     104000
2    6194000
3     114000
4      41000
Name: wage, dtype: int32
```

Now, when we try to calculate the mean wage, we won't run into an error:

In []:

```
df['wage'].mean()
```

Out[]:

```
1654078.0399274046
```

Let's see what other columns may need to be cleaned. Let's re-print the .head() and .info() views.

In []:

```
df.head()
```

Out[]:

	id	first_name	lastname	height	weight_in_lbs	city	state	country	foot	age	wage	value	league
0	1	Jacob	Miller	6'3"	192lbs	Atlanta	Georgia	USA	Right	27	97000	$1.2M	National
1	2	Santiago	Parker	6'1"	210lbs	Edmonton	Alberta	Canada	Left	31	104000	$828.8K	Championship
2	3	Nolan	Morris	6'1"	198lbs	Ottawa	Ontario	Canada	Right	27	6194000	$55.7M	Championship
3	4	Michael	Brooks	5'10"	198lbs	St. Louis	Missouri	USA	Right	28	114000	$1M	National
4	5	Andrew	Watson	5'11"	207lbs	San Diego	California	USA	Right	27	41000	$447.7K	Championship

In []:

```
df.info()
```

Out[]:

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4959 entries, 0 to 4999
Data columns (total 13 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   id             4959 non-null   int64 
 1   first_name     4959 non-null   object
 2   lastname       4959 non-null   object
 3   height         4959 non-null   object
 4   weight_in_lbs  4959 non-null   object
 5   city           4959 non-null   object
 6   state          4959 non-null   object
 7   country        4959 non-null   object
 8   foot           4959 non-null   object
 9   age            4959 non-null   int64 
 10  wage           4959 non-null   int32 
 11  value          4959 non-null   object
 12  league         4959 non-null   object
dtypes: int32(1), int64(2), object(10)
memory usage: 683.0+ KB
```

We can see that value, height, and weight all need some cleaning. Weight allows us to follow the same process as above, but the other two require a bit more complex cleaning.

Cleaning Weight Data

In []:

```
df['weight_in_lbs'].head()
```

Out[]:

```
0    192lbs
1    210lbs
2    198lbs
3    198lbs
```

```
4    207lbs
Name: weight_in_lbs, dtype: object
```

In []:

```
df['weight_in_lbs'].str[-3:].unique()
```

Out[]:

```
array(['lbs'], dtype=object)
```

We can also chain the commands we used above to make the code shorter. The important thing to realize here is the order: we first take away the strings 'lbs', and then cast the column as integers.

In []:

```
df['weight_in_lbs'] = df['weight_in_lbs'].str.replace('lbs',
'').astype(int)
```

Cleaning Height Data

In []:

```
df['height'].head()
```

Out[]:

```
0     6'3"
1     6'1"
2     6'1"
3    5'10"
4    5'11"
Name: height, dtype: object
```

We can see that the heights are recorded in feet and inches. This isn't great if we wanted to calculate measures of central tendency. For the time being, let's change the data into inches. We know that there are 12 inches per foot.

What we'll do is split the text of the column at the ′ sign into two columns. We could also do this using regular expressions, but that exists outside the scope of this course.

In []:

```
df[['height_ft', 'height_in']] = df['height'].str.split("'", n
= 1, expand = True)
```

What we've done is used the .split() function to split the text at the ′ character. The function works as follows:

- the first argument is the character to split on
- the second argument tells the function how often to split (the default is -1, which is unlimited)
- the last argument is whether or not to create new columns

To make our work a bit easier to explain, we've created two separate columns.

Let's see what these columns look like now:

In []:

```
df[['height_ft', 'height_in']].head()
```

Out[]:

	height_ft	height_in
0	6	3"
1	6	1"
2	6	1"
3	5	10"
4	5	11"

We no longer have the ' character in height_feet, but we still have the " character in height_in. Let's remove that we did earlier:

In []:

```
df['height_in'] = df['height_in'].str.replace('"','')
df[['height_ft', 'height_in']].head()
```

Out[]:

	height_ft	height_in
0	6	3
1	6	1
2	6	1
3	5	10
4	5	11

We'll need to convert each of these two integers, to be able to correctly calculate the height in inches.

In []:

```
df['height_ft'] = df['height_ft'].astype(int)
df['height_in'] = df['height_in'].astype(int)
```

We can now store these values back in the height column, following some math. We will want to:

- multiply the height_ft column by 12
- add the height_in column

In []:

```
df['height'] = df['height_ft'] * 12 + df['height_in']
df.head()
```

Out[]:

	id	first_name	lastname	height	weight_in_lbs	city	state	country	foot	age	wage	value	league	height_ft	height_in
0	1	Jacob	Miller	75	192	Atlanta	Georgia	USA	Right	27	97000	$1.2M	National	6	3
1	2	Santiago	Parker	73	210	Edmonton	Alberta	Canada	Left	31	104000	$828.8K	Championship	6	1
2	3	Nolan	Morris	73	198	Ottawa	Ontario	Canada	Right	27	6194000	$55.7M	Champio	6	1

													nship		
3	4	Michael	Brooks	70	198	St. Louis	Missouri	USA	Right	28	114000	$1M	National	5	10
4	5	Andrew	Watson	71	207	San Diego	California	USA	Right	27	41000	$447.7K	Championship	5	11

Great! That worked. Now we can delete ("drop") the last two columns, as we no longer need them. We will also want to rename the height column to be explicit that it's in inches.

In []:

```
df.drop(columns = ['height_in', 'height_ft'], inplace=True)
df.rename(columns = {'height':'height_in'}, inplace=True)
```

Cleaning Value Data

Cleaning the value data will be a little trickier. We can see from the print out above that we have different measures (K, M, etc.), but also decimal points. Let's explore how we can take this on. We'll begin by (1) verifying and dropping currency (if it's all Dollars), and identifying how many suffixes we're dealing with.

In []:

```
df['value'].str[0].unique()
```

Out[]:

```
array(['$'], dtype=object)
```

Everything is in Dollars, so we can drop the dollar sign altogether.

In []:

```
df['value'] = df['value'].str.replace('$', '')
```

Let's take a look at the suffixes.

In []:

```
df['value'].str[-1].unique()
```

Out[]:

```
array(['M', 'K'], dtype=object)
```

We can see here that we have some players valued in the thousands, others in millions. This might be a little easier with regular expressions, but as that sits outside the scope of this course, let's explore how else we can take on this work.

One approach we could take would be to extract the M and K classifiers into another column and work from there.

In []:

```
df['value_km'] = [row[-1] for row in df['value']]
df.value_km.unique()
```

Out[]:

```
array(['M', 'K'], dtype=object)
```

In []:

```
df.head()
```

Out[]:

	id	first_name	lastname	height_in	weight_in_lbs	city	state	country	foot	age	wage	value	league	value_km
0	1	Jacob	Miller	75	192	Atlanta	Georgia	USA	Right	27	97000	1.2M	National	M
1	2	Santiago	Parker	73	210	Edmonton	Alberta	Canada	Left	31	104000	828.8K	Championship	K
2	3	Nolan	Morris	73	198	Ottawa	Ontario	Canada	Right	27	6194000	55.7M	Championship	M

3	4	Michael	Brooks	70	198	St. Louis	Missouri	USA	Right	28	114000	1M	National	M
4	5	Andrew	Watson	71	207	San Diego	California	USA	Right	27	41000	447.7K	Championship	K

We extracted the last value of the *value* column into a new column. From here, we can assign the correct number of zeroes to either a K or M.

In []:

```
df['value_km'] = df['value_km'].str.replace('K','000')
df['value_km'] = df['value_km'].str.replace('M', '000000')
```

To prepare the *value* column, let's now remove any "K" and "M" characters. We'll also want to make sure the largest number of decimals points in the data.

In []:

```
df['value'] = df['value'].str.replace('K','')
df['value'] = df['value'].str.replace('M','')
```

We'll now split the string at the '.' symbol into another column.

In []:

```
df[['value_original', 'value_decimal']] =
df['value'].str.split('.', expand=True)
df['value_decimal'].unique()
```

Out[]:

```
array(['2', '8', '7', None, '1', '9', '4', '5', '3', '6'],
dtype=object)
```

What we've done above is split the value column into two columns: the first for anything before the decimal, and anything after. We then printed the unique list of all decimal values to make sure they only had one decimal place (since they all are only one value long, we can be sure they

are only one decimal in length). We can now prepend the value to the value_km column to ensure we don't add redundant 0s.

In []:

```
df['value_decimal'] = df['value_decimal'].fillna(value=0)
df['value_decimal'].unique()
```

Out[]:

```
array(['2', '8', '7', 0, '1', '9', '4', '5', '3', '6'],
dtype=object)
```

In []:

```
df['value_end'] = df['value_decimal'].map(str) +
df['value_km'].map(str)
```

Now we need to drop the zero at the end.

In []:

```
df['value_end'] = df['value_end'].str[:-1]
```

Now, let's combine our original_value column with the value_end column, into a new value_final column and recast it as an integer.

In []:

```
df['value_final'] = df['value_original'].map(str) +
df['value_end'].map(str)
df['value_final'] = df['value_final'].astype(int)
```

Now let's drop all the columns that we created that we no longer need:

In []:

```
df.drop(columns = ['value', 'value_km', 'value_original',
'value_decimal', 'value_end'], inplace=True)
df.rename(columns = {'value_final': 'value'}, inplace = True)
```

Let's see what this dataset looks like now:

In []:

```
df.head()
```

Out[]:

	id	first_name	lastname	height_in	weight_in_lbs	city	state	country	foot	age	wage	league	value
0	1	Jacob	Miller	75	192	Atlanta	Georgia	USA	Right	27	97000	National	1200000
1	2	Santiago	Parker	73	210	Edmonton	Alberta	Canada	Left	31	104000	Championship	828800
2	3	Nolan	Morris	73	198	Ottawa	Ontario	Canada	Right	27	619400	Championship	55700000
3	4	Michael	Brooks	70	198	St. Louis	Missouri	USA	Right	28	114000	National	1000000
4	5	Andrew	Watson	71	207	San Diego	California	USA	Right	27	41000	Championship	447700

Styling Data with Pandas

We can see in the dataframe head above that the value and wage are listed as integers, but could also be represented as dollars to make them easier to read. However, we don't want to change the format back to a string, as then we can't run analysis on it. Thankfully, we can apply some formatting directly with Pandas:

In []:

```
df.head().style.format({'wage': '${0:,}'})
```

Out[]:

	id	first_name	lastname	height_in	weight_in_lbs	city	state	country	foot	age	wage	league	value
0	1	Jacob	Miller	75	192	Atlanta	Georgia	USA	Right	27	$97,000	National	1200000
1	2	Santiago	Parker	73	210	Edmonton	Alberta	Canada	Left	31	$104,000	Championship	828800
2	3	Nolan	Morris	73	198	Ottawa	Ontario	Canada	Right	27	$6,194,000	Championship	55700000
3	4	Michael	Brooks	70	198	St. Louis	Missouri	USA	Right	28	$114,000	National	1000000
4	5	Andrew	Watson	71	207	San Diego	California	USA	Right	27	$41,000	Championship	447700

Similarly, we can apply this to multiple columns:

In []:

```
df.head().style.format({
   'wage': '${0:,}',
   'value': '${0:,}'
   })
```

Out[]:

id	first_name	lastname	height_in	weight_in_lbs	city	state	country	foot	age	wage	league	value	
0	1	Jacob	Miller	75	192	Atlanta	Georgia	USA	Right	27	$97,000	National	$1,200,000
1	2	Santiago	Parker	73	210	Edmonton	Alberta	Canada	Left	31	$104,000	Championship	$828,800
2	3	Nolan	Morris	73	198	Ottawa	Ontario	Canada	Right	27	$6,194,000	Championship	$55,700,000
3	4	Michael	Brooks	70	198	St. Louis	Missouri	USA	Right	28	$114,000	National	$1,000,000
4	5	Andrew	Watson	71	207	San Diego	California	USA	Right	27	$41,000	Championship	$447,700

This allows us to more easily read the data that's stored in the cell while maintaining its integrity as an integer.

However, it's a bit cumbersome to write out and is best saved for the final presentation.

There are a number of other helpful formatting codes that we'll display here for use later on:

Code	Example	Description
${0:,.2f}'	$2.00	Dollars with 2 decimals
${0:,.0f}'	$2	Dollars with 0 decimals
{:,.2%}'	2.00%	Percent with 2 decimals
{:,.0%}'	2%	Percent with 0 decimals
{:.0f}'	2	Rounds to 0 decimals

Grouping Data in Pandas using Group By

We may find ourselves wanting to group together data by a certain attribute and creating summary statistics. We can do this easily using the built-in Group By function in Pandas.

Let's begin by grouping our data by a player's preferred foot, and counting how many players prefer each foot.

In []:

```
df.groupby('foot')['id'].count()
```

Out[]:

```
foot
Left     1354
Right    3605
Name: id, dtype: int64
```

What we've asked Pandas to do here is:

1. Group by preferred_foot
2. Display only the id column (so that it doesn't display the same count for each column)
3. Count each instance of right or left foot

This can be immensely helpful in returning crosstabs for different columns. Say, for example, we wanted to return the ten most valuable teams, as measured by its players' values. We can also do this by changing the .count() method to a .agg() method which accepts a list of aggregation functions:

In []:

```
df.groupby('city')['value'].agg(['sum']).sort_values(by =
'sum', ascending = False).head(10)
```

Out[]:

	sum
city	
San Diego	2.67E+09
Anaheim	2.42E+09
New Orleans	2.28E+09
Oklahoma City	2.28E+09
Seattle	2.26E+09
Phoenix	2.22E+09
Orlando	2.18E+09
Indianapolis	2.14E+09
Chicago	1.98E+09
Portland	1.97E+09

We've chained a number of commands here. Let's go through them in order:

- **df.groupby('club')['value_final'].sum()** groups by the club and generates a sum for the value_final
- **.sort_values(ascending = False)** sorts the object in descending order
- **.head(10)** displays the top ten teams

We could have written this as three separate lines for the same result, but this is much faster.

Let's make this a little prettier by applying the formatting we learned earlier:

In []:

```
df.groupby('city')['value'].agg(['sum']).sort_values(by = 'sum',
ascending = False).head(10).style.format('${0:,.0f}')
```

Out[]:

sum	
city	
San Diego	$2,671,410,900
Anaheim	$2,424,731,600
New Orleans	$2,282,601,700
Oklahoma City	$2,277,591,400
Seattle	$2,262,016,900
Phoenix	$2,215,048,200
Orlando	$2,176,438,600
Indianapolis	$2,142,666,500
Chicago	$1,976,995,300
Portland	$1,968,534,800

Similarly, we can calculate the top ten teams with the average oldest players:

In []:

```
df.groupby('city')['age'].agg(['mean']).sort_values(by =
'mean', ascending=False).head(10).style.format('{:.2f}')
```

Out[]:

	mean
city	
Miami	26.05
Tampa	25.95
Anaheim	25.91
Minneapolis/St. Paul	25.83
Pittsburgh	25.69
Ottawa	25.66
Phoenix	25.62
Montreal	25.61
Cleveland	25.59
Green Bay	25.59

The .groupby method brings us to a natural segway into pivot tables.

Generating Summaries with Pandas Pivot Tables

You may be familiar with generating pivot tables in Excel. They provide helpful insight into the data by allowing you to quickly generate summaries of the data you're working with. Pandas also provides access to pivot tables. Let's generate some summary statistics using pivot tables.

One of the main benefits here is that we can define our own functions by which to summarize data. That's a bit outside the scope the course, but keep this in mind as you progress.

In []:

```
wages = pd.pivot_table(data=df, index = 'city', values =
'wage', aggfunc='sum')
wages.sort_values(by='wage', ascending=False).head(10)
```

Out[]:

	wage
city	
San Diego	280280000
Anaheim	240535000
Oklahoma City	231703000
Phoenix	223029000
Indianapolis	215189000
Seattle	215069000
New Orleans	211621000
Orlando	207052000
Portland	201201000
Green Bay	199510000

On the surface, this looks a lot like the group-by statements we created earlier. Where this becomes much more powerful, is when we combine it with multiple columns or aggregation functions. For example, if we wanted to show both the total club value by player, but also the total sum of salaries, we could write the following code:

In []:

```
wages = pd.pivot_table(data = df, index = 'city', values =
'wage', aggfunc=[sum, np.mean, max, min])
wages.head()
```

Out[]:

	sum	mean	max	min
	wage	wage	wage	wage
city				
Anaheim	240535000	2.45E+06	16641000	54000
Atlanta	184116000	1.66E+06	16926000	42000
Baltimore	108270000	1.16E+06	12432000	53000
Boston	127508000	1.47E+06	17556000	47000
Buffalo	113383000	1.30E+06	15801000	37000

We can see that the columns look a little different than what we've seen before. Let's see what the column names are.

In []:

```
wages.columns
```

Out[]:

```
MultiIndex([( 'sum', 'wage'),
            ('mean', 'wage'),
            ( 'max', 'wage'),
            ( 'min', 'wage')],
           )
```

The resulting columns are a list of tuples. This is a byproduct of the fact that the resulting table is now a multi-index dataframe. This is a bit out of

the scope of this course. To get back to a single index, we can run the reset_index() function and assign column names.

In []:

```
wages.reset_index()
wages.columns = ['sum', 'mean', 'max', 'min']
wages.columns
```

Out[]:

```
Index(['sum', 'mean', 'max', 'min'], dtype='object')
```

This chapter introduced Pandas pivot tables. While they can do much more than what we've explored here, let's begin diving deeper into data visualization.

Visualizing Data with Pandas

Pandas provides some rudimentary charts that you can create, built on top of the topic of our next chapter: matplotlib. We'll explore these first, as they're easy to put together when working with a dataframe. The next chapter will explore how to customize these graphs.

In order to make this work, let's import the correct library. We'll also run a Jupyter *magic* command, that will allow the output to be displayed inline.

In []:

```
from matplotlib import pyplot as plt
%matplotlib inline
```

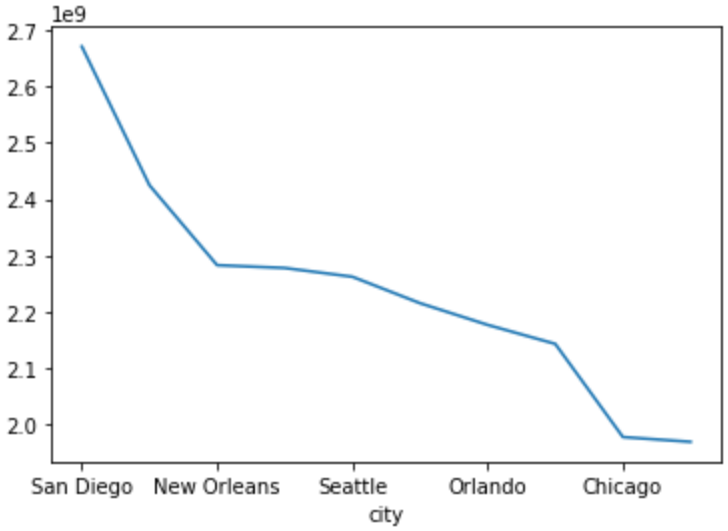
Once we have imported the matplotlib, plotting in Pandas is as easy as adding .plot to the end of a dataframe. Let's take the summary table we created earlier of the most valuable clubs and add .plot() to it.

In []:

```
df.groupby('city')['value'].sum().sort_values(ascending=False).
head(10).plot()
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a414a879c8>
```



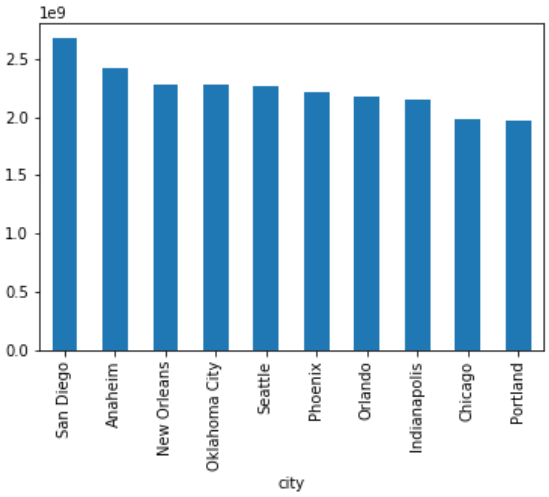
Right away, we can tell that this isn't a very pretty graph. The data shouldn't be displayed as a line chart and the text is impossible to read. Let's add some arguments to .plot() to change some of this.

In []:

```
df.groupby('city')['value'].sum().sort_values(ascending=False).
head(10).plot(kind = 'bar')
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a413cea3c8>
```



By adding the argument of *kind = 'bar'*, we've changed the data into discrete categories.

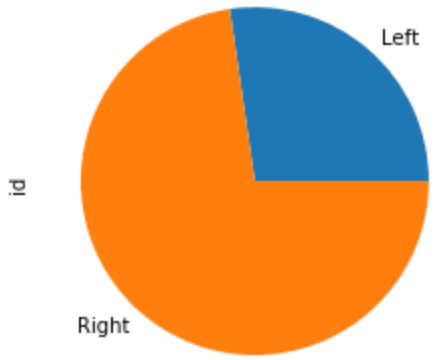
Let's now use our right/left foot analysis to create a pie chart.

In []:

```
df.groupby('foot')['id'].count().plot(kind = 'pie')
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a414ab46c8>
```



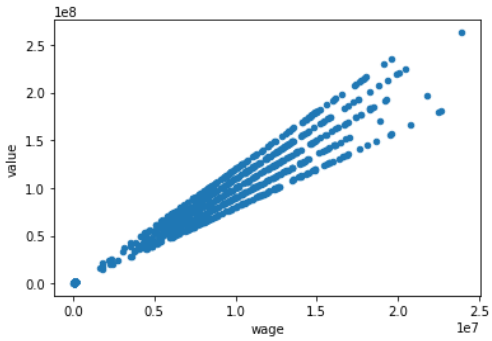
We can also create a scatter plot that plots wage against value.

In []:

```
df.plot(kind='scatter', x='wage', y='value')
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a414ae6d48>
```



These charts are by no means pretty, but they offer easy insight into patterns that we are able to more readily identify visually. Let's move onto matplotlib and seaborn now, to see how we can customize these graphs.

Plotting with Python: Matplotlib and Seaborn

We saw earlier how easy Pandas and matplotlib make plotting data. Let's take this to another level by customizing the outputs of the graphs. Seaborn is a library that is built on top of matplotlib and seeks to remove a lot of the verbose code required to make graphs look better.

Let's begin by importing the required libraries. We've already imported matplotlib, but for the sake of the chapter, we'll re-write the code here.

In []:

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

It may seem a bit verbose, but let's take a quick look at the anatomy of a plot that's based on matplotlib. As this is true as well for seaborn and many other libraries derived from matplotlib, it's worthwhile exploring.

A **figure** is the overall window that everything is drawn on. It is the top-level component for everything else we'll consider. You can make several figures, and each figure can contain multiple elements (such as plots, titles, etc.)

An **axes** is the area on which you plot data using functions. Each axes can contain ticks and labels associated with data. Each **axes** contains both a y-axis and an x-axis, which contain ticks, ticklines, and ticklabels. They also contain labels, a title, a legend, and gridlines. They also contain *spines*, which connect the tick marks and mark boundaries of the data.

Matplotlib offers two difference interfaces: object-oriented and state-machine. Let's begin by exploring the object-oriented interface.

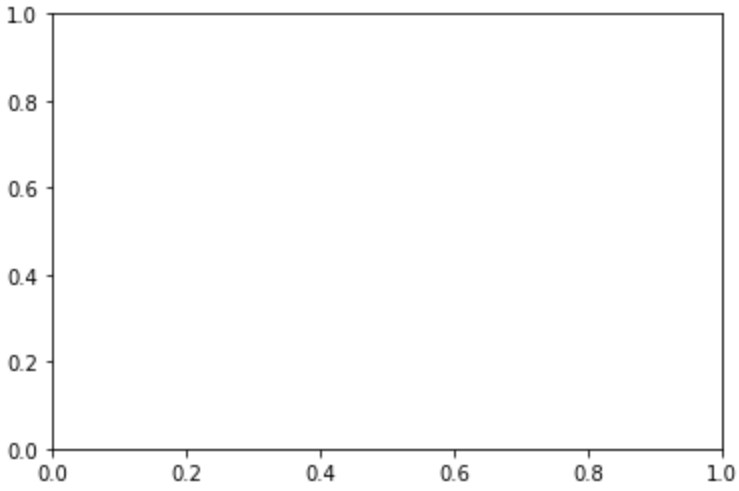
Object-Oriented

Let's start building some plots.

In []:

```
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
```

Out[]:



Let's break this code down a bit:

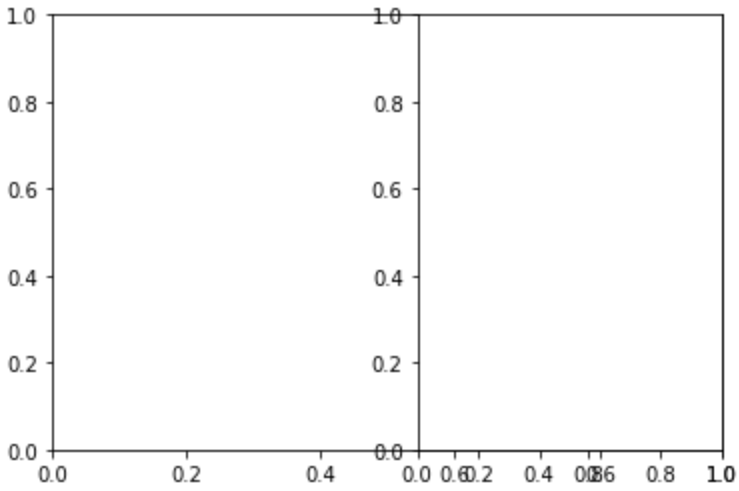
1. Line 1: create a figure object and assigns it to the variable fig
2. Line 2: adds a subplot to the figure object

Note the (1,1,1) parameters. These take the form of (# of rows, # of columns, plot number). So we if wanted to create two plots in two columns and one row, we could write:

In []:

```
fig = plt.figure()
ax1 = fig.add_subplot(1,1,1)
ax2 = fig.add_subplot(1,2,2)
```

Out[]:



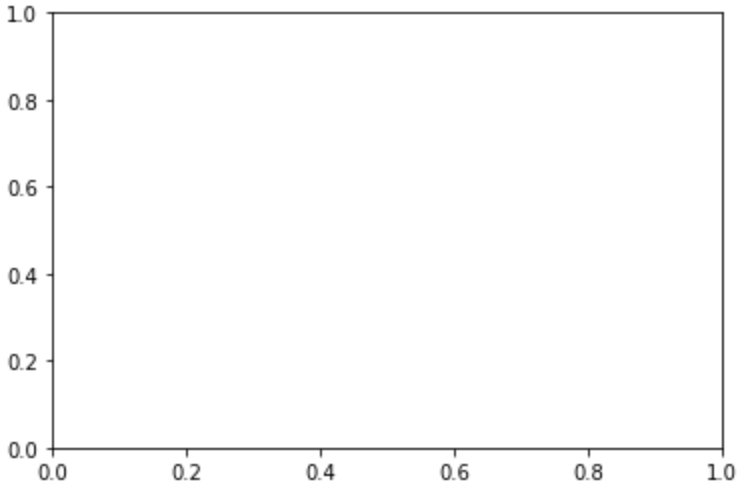
The commas in the arguments are actually optional - we could have achieved the same result earlier using (111) and (122).

Say we wanted to delete a subplot: we could achieve this using .delaxes():

In []:

```
fig.delaxes(ax2)
fig
```

Out[]:



We can also specify the size of our figure by using the following commands, where the size is added as a tuple in inches in the format of

(width, height). DPI refers to dots per inch, and is akin to the resolution of the plot.

In []:

```
fig = plt.figure(figsize=(4,4), dpi=100)
```

Out[]:

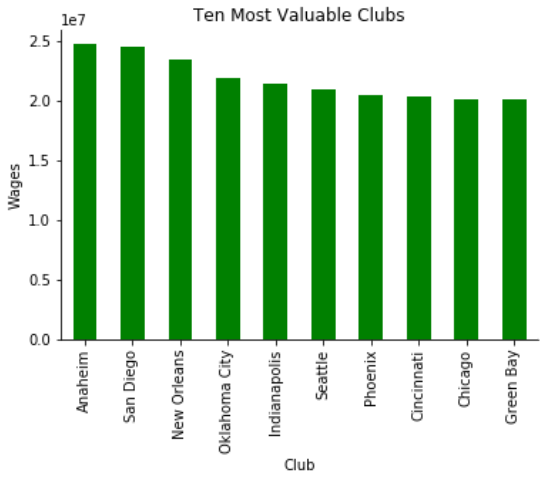
```
<Figure size 400x400 with 0 Axes>
```

Now let's create a plot similar to the ones we'd created above. We'll generate a summary table and assign it to a variable to be able to call it more readily.

In []:

```
wages =
df.groupby('city')['value'].mean().sort_values(ascending=False)
.head(10)
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax = wages.plot(kind='bar', color = 'green')
plt.ylabel('Wages')
plt.xlabel('Club')
plt.title('Ten Most Valuable Clubs')
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
```

Out[]:



While we've been able to make the graph look a little better, the numbers on the y-axis are a little hard to decipher. It'd be better if we were able to format them as dollars. Thankfully, we've already learned about string notation earlier.

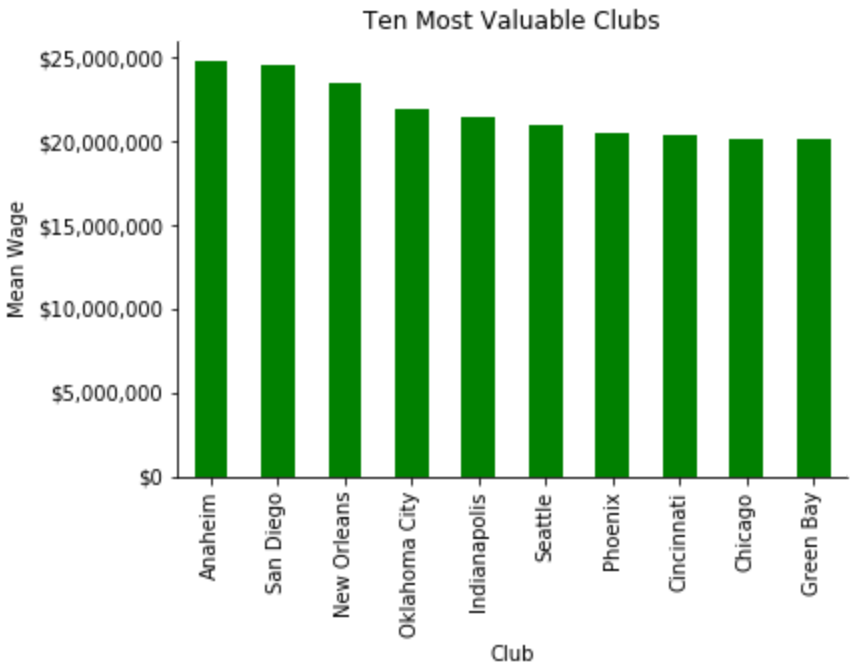
We'll need to import another matplotlib module, *ticker*, to be able to make this happen.

In []:

```
import matplotlib.ticker as ticker
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax = wages.plot(kind='bar', color = 'green')
plt.ylabel('Mean Wage')
plt.xlabel('Club')
plt.title('Ten Most Valuable Clubs')
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

formatter = '${x:,.0f}'
formatter = ticker.StrMethodFormatter(formatter)
ax.yaxis.set_major_formatter(formatter)
```

Out[]:



State Machine

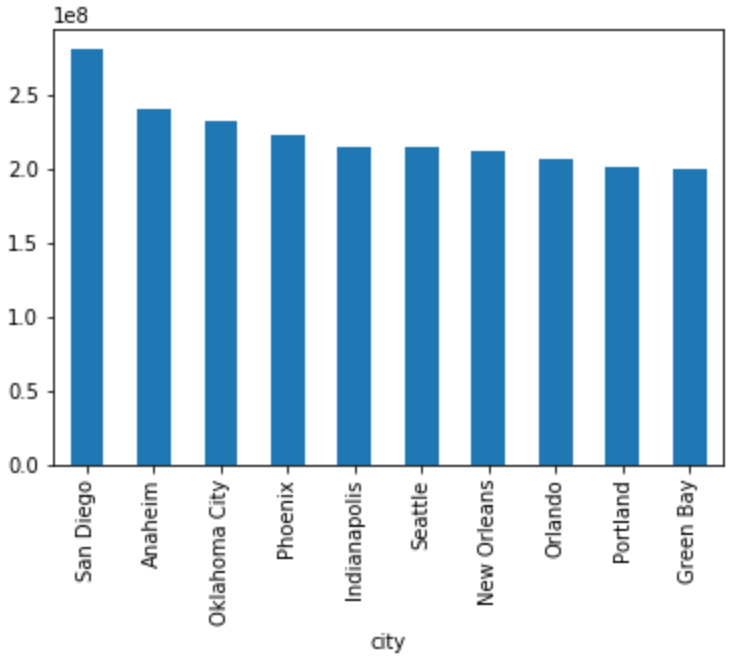
Let's now explore matplotlib in the state-machine method, rather than object-oriented, as it's a few less things to wrap our minds around. Working in jupyter notebooks eliminates some of the guesswork required to identify which plot you're working.

In []:

```
valuable =
df.groupby('city')['wage'].sum().sort_values(ascending=False).h
ead(10)
plt.figure()
valuable.plot(kind = 'bar')
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a417559348>
```



Now that we have the shell of our plot, we can add customizations to it using the plt. and ax. prefixes, depending on what we want to change.

Since we're working within the state-machine method, we want to be explicit and keep repeating our code.

In []:

```
top10 =
df.groupby('city')['value'].sum().sort_values(ascending=False).
head(10).reset_index()
top10 = top10.rename(columns={'city': 'City', 'value':'Total
Value of Players'})
top10
```

Out[]:

City	Total Value of Players	
0	San Diego	2.67E+09
1	Anaheim	2.42E+09
2	New Orleans	2.28E+09
3	Oklahoma City	2.28E+09
4	Seattle	2.26E+09
5	Phoenix	2.22E+09
6	Orlando	2.18E+09
7	Indianapolis	2.14E+09
8	Chicago	1.98E+09
9	Portland	1.97E+09

You may notice the .reset_index() function we called at the end of our chain. What this does is impact the multi-index we previously created and creates a little dataframe of its own that is no longer connected the source data.

Let's add a column that pretties up the value of players a little, by rounding it to the nearest million.

In []:

```
top10['Value Millions'] = top10['Total Value of
Players']/1000000
top10
```

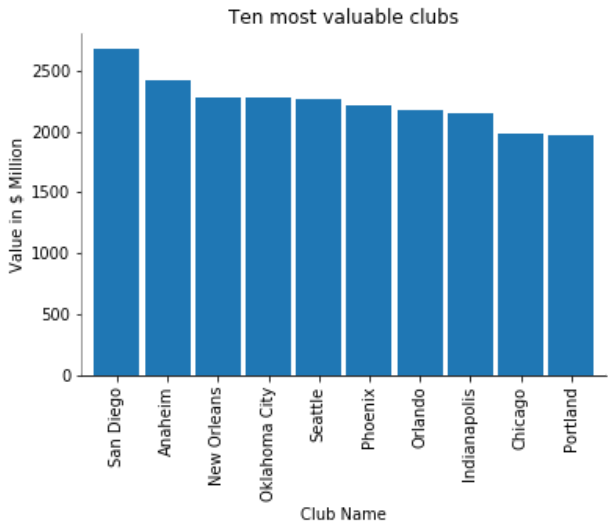
Out[]:

	City	Total Value of Players	Value Millions	
0	San Diego	2.67E+09	2671.4109
1	Anaheim	2.42E+09	2424.7316
2	New Orleans	2.28E+09	2282.6017
3	Oklahoma City	2.28E+09	2277.5914
4	Seattle	2.26E+09	2262.0169
5	Phoenix	2.22E+09	2215.0482
6	Orlando	2.18E+09	2176.4386
7	Indianapolis	2.14E+09	2142.6665
8	Chicago	1.98E+09	1976.9953
9	Portland	1.97E+09	1968.5348

In []:

```
fig, ax = plt.subplots()
top10.plot(kind = 'bar', x='City', y='Value Millions', ax=ax,
width=0.9)
ax.set(title='Ten most valuable clubs', xlabel = 'Club Name',
ylabel = 'Value in $ Million')
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_color('gray')
ax.legend().set_visible(False)
```

Out[]:



This is quite a bit of typing to get a halfway decent looking plot. Thankfully, matplotlib also has styles built into it. We can access these by writing the code below:

In []:

```
plt.style.available
```

Out[]:

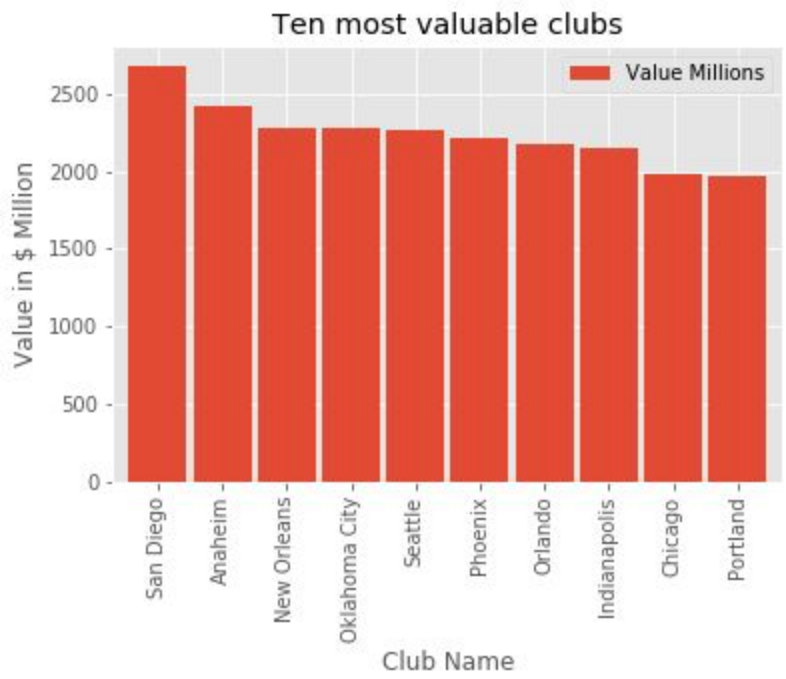
```
['bmh',
 'classic',
 'dark_background',
 'fast',
 'fivethirtyeight',
 'ggplot',
 'grayscale',
 'seaborn-bright',
 'seaborn-colorblind',
 'seaborn-dark-palette',
 'seaborn-dark',
 'seaborn-darkgrid',
 'seaborn-deep',
 'seaborn-muted',
 'seaborn-notebook',
 'seaborn-paper',
 'seaborn-pastel',
 'seaborn-poster',
 'seaborn-talk',
 'seaborn-ticks',
 'seaborn-white',
 'seaborn-whitegrid',
 'seaborn',
 'Solarize_Light2',
 'tableau-colorblind10',
 '_classic_test']
```

To change to a particular style, use:

In []:

```
plt.style.use('ggplot')
fig, ax = plt.subplots()
top10.plot(kind = 'bar', x='City', y='Value Millions', ax=ax,
width=0.9)
ax.set(title='Ten most valuable clubs', xlabel = 'Club Name',
ylabel = 'Value in $ Million');
```

Out[]:

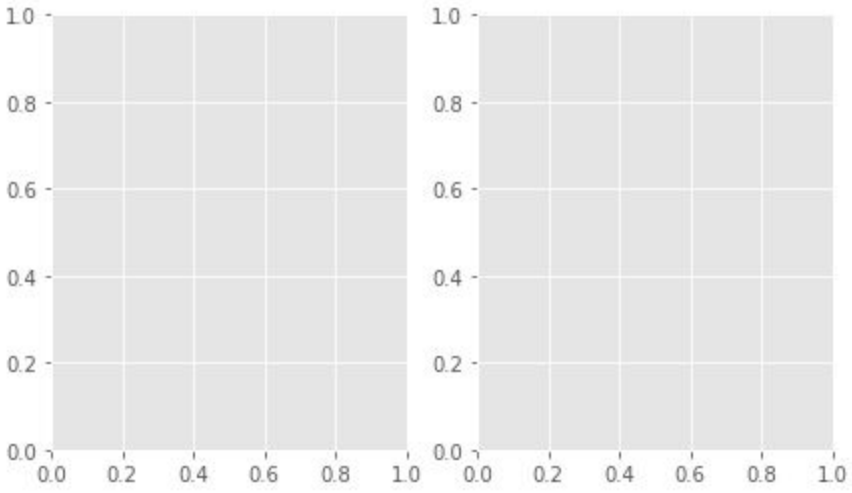


One of the big benefits of writing in object-oriented modes is that we can easily apply changes in styles to multiple plots at once. Let's begin by creating a new figure that contains two plots.

In []:

```
plt.style.use('ggplot')
fig, (ax0, ax1) = plt.subplots(nrows=1,ncols=2, figsize=(7, 4))
```

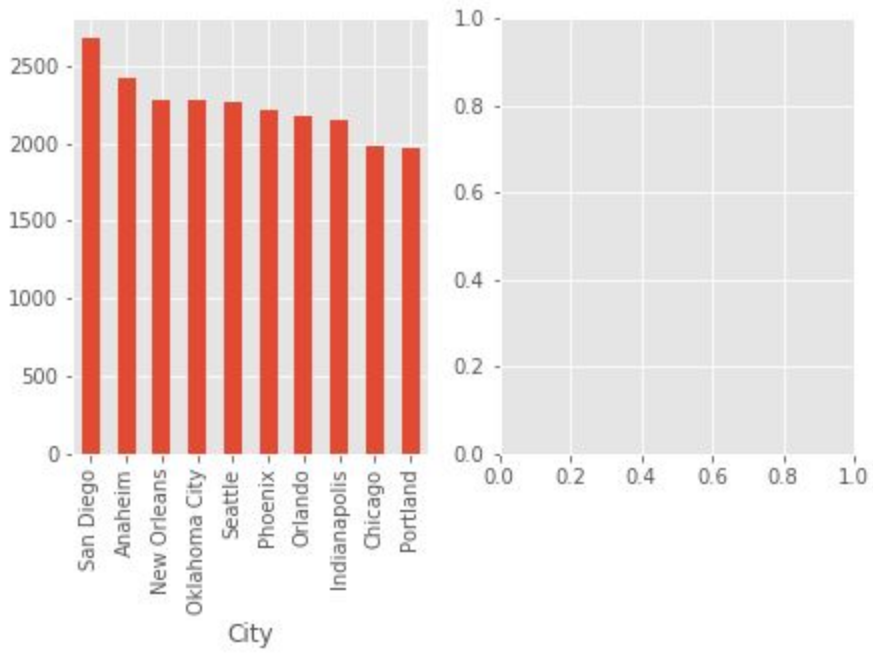
Out[ ]:



In [ ]:

```
fig, (ax0, ax1) = plt.subplots(nrows=1,ncols=2, figsize=(7,
4))
top10.plot(kind = 'bar', x='City', y='Value Millions', ax=ax0)
ax0.legend().set_visible(False)
```

Out[ ]:

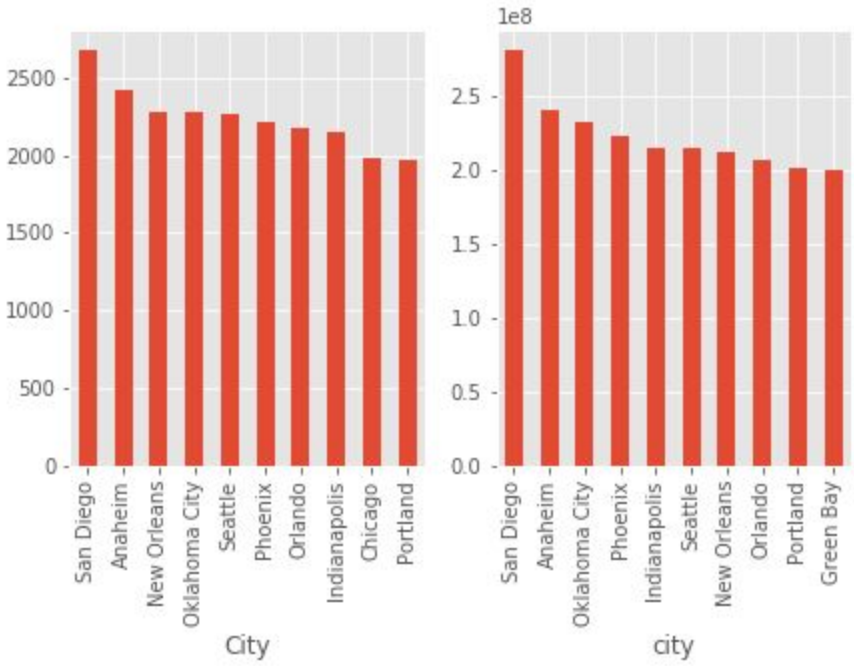


We've only told matplotlib one axes on which to plot data, and as such, the one on the right (ax1) remains empty. We could add in another chart easily.

In []:

```
fig, (ax0, ax1) = plt.subplots(nrows=1,ncols=2, figsize=(7, 4))
top10.plot(kind = 'bar', x='City', y='Value Millions', ax=ax0)
df.groupby('city')['wage'].sum().sort_values(ascending=False).h
ead(10).plot(kind='bar', ax=ax1)
ax0.legend().set_visible(False)
```

Out[]:



Similar to before, we can add in a number of customizations, such as titles for each axes and for the figure itself.

You'll notice as well on the right axes, the y-axis has scientific notation. We can remove this by importing another matplotlib package. We'll use a lambda function, which is a bit outside of the scope of this course, but you can copy and paste the code to re-use it.

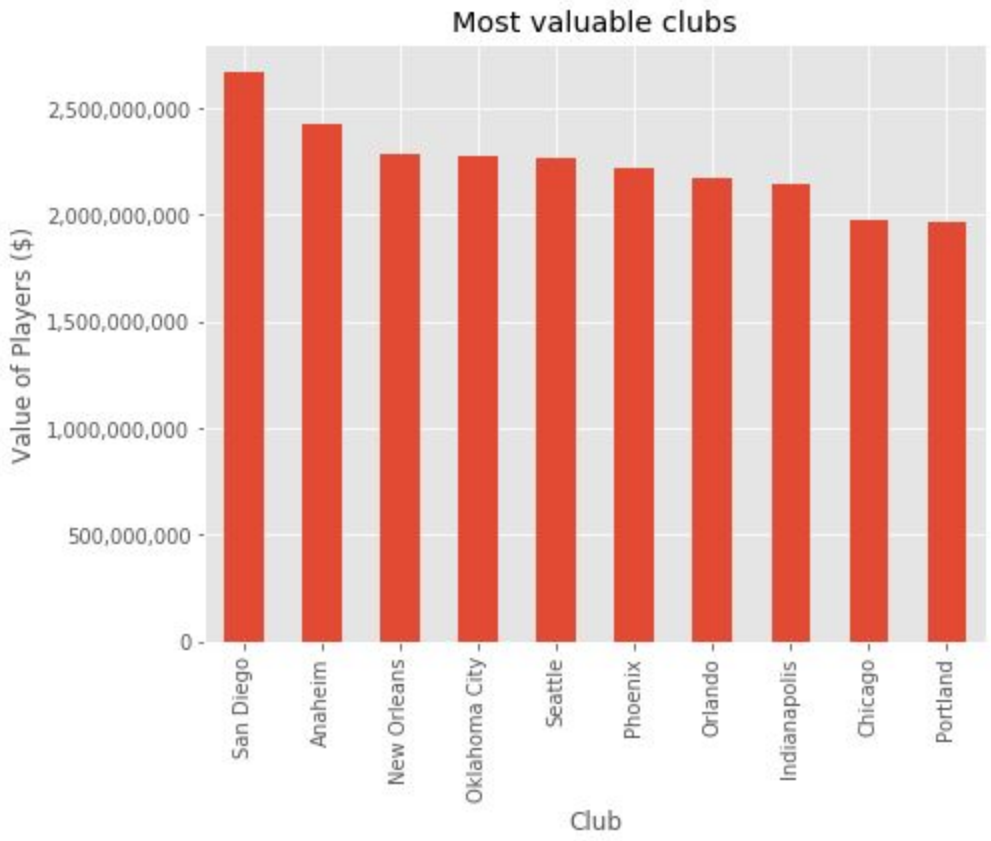
In []:

```
#Import Additional Package
import matplotlib.ticker as ticker

#Initiate Figure and Axes
plt.style.use('ggplot')
fig, (ax0) = plt.subplots(nrows=1, ncols=1, figsize=(7, 5.5))

#Axes 1
top10.plot(kind = 'bar', x = 'City', y = 'Total Value of
Players', ax = ax0)
ax0.set(title='Most valuable clubs', xlabel = 'Club', ylabel =
'Value of Players ($)')
ax0.legend().set_visible(False)
ax0.get_yaxis().set_major_formatter(
   ticker.FuncFormatter(lambda x, p: format(int(x), ',')))
```

Out[]:



Introducing Seaborn

We can see that matplotlib is incredibly flexible, but it's also extremely verbose. Of course, that's a double-edged sword - we get more control but we also have to type more to get something that looks decent.

That's where seaborn comes in. It's built on top of matplotlib, but it removes a lot of the boilerplate text that matplotlib uses. Of course, this also reduces its flexibility but may be enough to make decent looking graphs.

Seaborn is also closely integrated with Pandas' dataframes and allows for easier chart creation.

Let's begin by importing the library. It's often abbreviated to sns.

In []:

```
import seaborn as sns
```

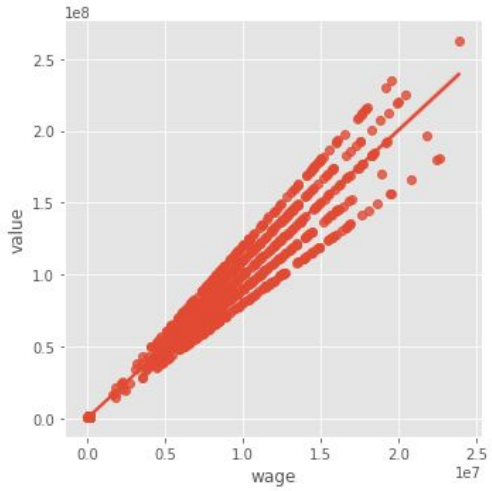
Let's start by creating the scatterplot we created earlier.

In []:

```
sns.lmplot(data=df, x='wage', y='value')
```

Out[]:

```
<seaborn.axisgrid.FacetGrid at 0x1a417460e08>
```



An interesting thing you'll notice is the line. Seaborn is tailored towards statistics and immediately implements a regression line. We can turn this off by using the fit_reg = False parameter.

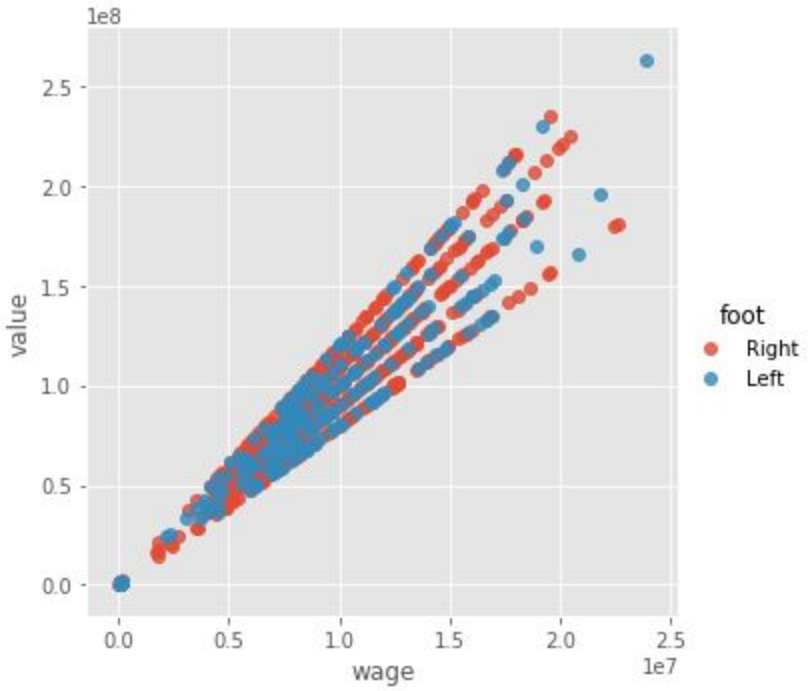
We can also add a third variable to the graph by passing a variable to the hue parameter. Let's do both of these things.

In []:

```
sns.lmplot(data=df, x='wage', y='value', fit_reg = False, hue =
'foot')
```

Out[]:

```
<seaborn.axisgrid.FacetGrid at 0x1a4188a2188>
```



Seaborn comes with five different themes installed: darkgrid, whitegrid, dark, white, and ticks. We can set these by following the syntax below.

In []:

```
sns.set_style('dark')
```

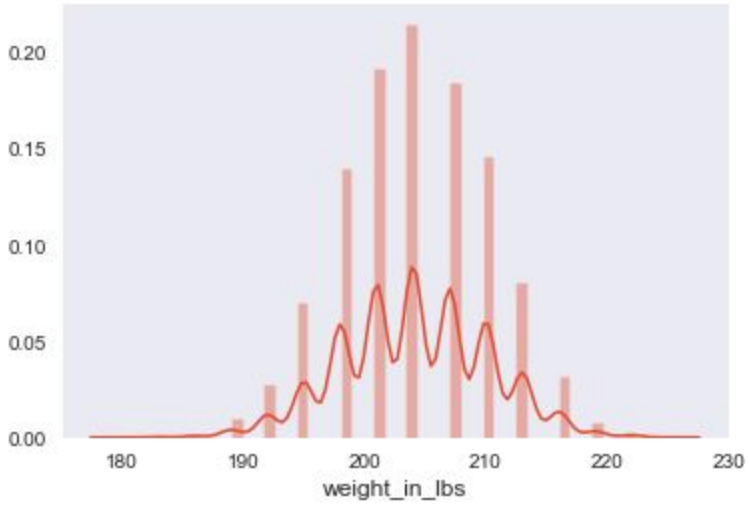
We can also create a histogram by using the .displot() function (for distribution plot):

In []:

```
sns.distplot(df['weight_in_lbs'])
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a4188c0988>
```



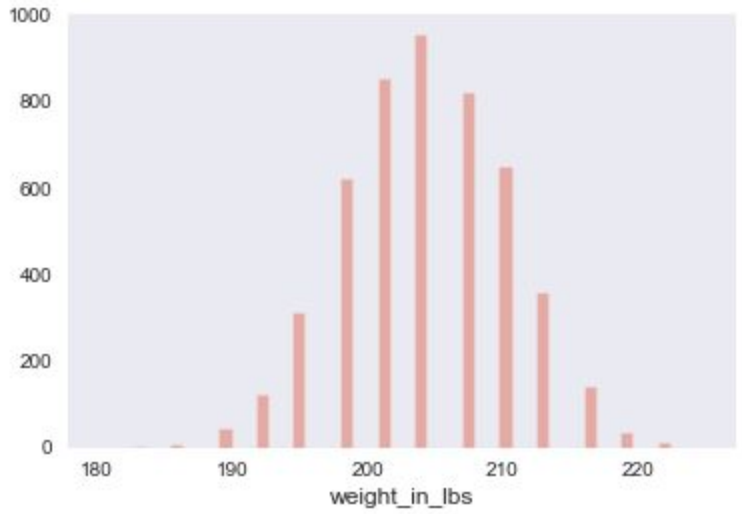
To remove the density line, we can pass the kde = False argument.

In []:

```
sns.distplot(df['weight_in_lbs'], kde = False)
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a4189a8cc8>
```



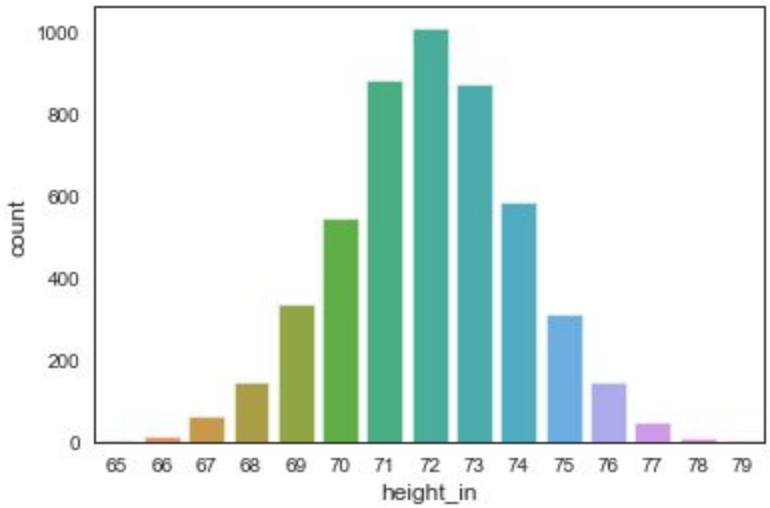
Now, let's explore making a bar graph, which in Seaborn is called a countplot.

In []:

```
sns.set_style('white')
sns.countplot(data=df, x='height_in')
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a418ac9248>
```

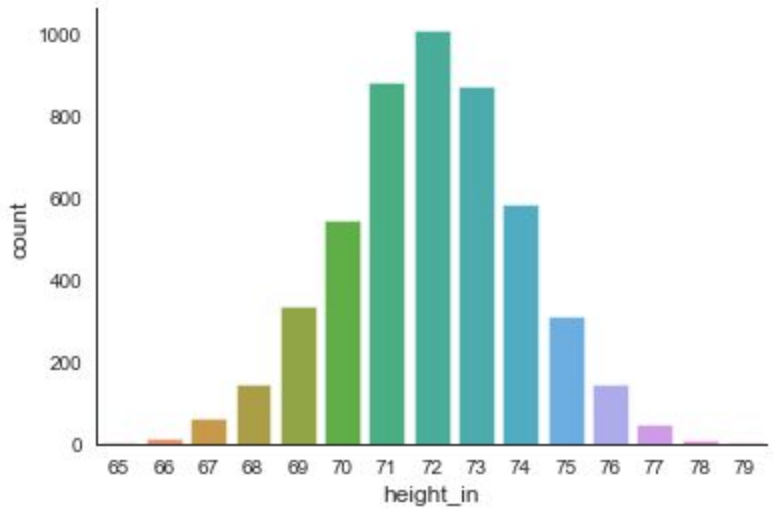


One of the issues with the white theme is the black spines along the right and top sides. In matplotlib, we would have had to pass two arguments to despine these spines. In Seaborn, this can be done with a single .despine() method.

In []:

```
sns.countplot(data=df, x='height_in')
sns.despine()
```

Out[]:



The key take-away from this isn't that Seaborn is *better* than matplotlib: they serve very different functions. Seaborn is a statistics focused plotting engine that's based on matplotlib. It seeks to make *some* difficult things easy to do, but has more limited use cases than matplotlib.

Conclusion

You've know learned some key pieces in getting started with data science. This is where the fun really starts! It might all seem a little daunting, but keep at it!

The best way to learn is to keep practicing. Use the dataset provided here or find some of your own to play around with.

The Python community is fantastic and incredibly helpful. Check out our posts on https://www.datagy.io or my Medium posts: https://medium.com/@nik.piepenbreier

StackOverlow and the r/learnpython subreddit are also valuable resources.