Lists are ordered collections of data. To create a list simply wrap other data, separated by commas, in square brackets [ and ]:

You can put any value into a list that you like, even another list. Just like the characters in a string, the elements in a list are ordered and can be accessed by index with bracket notation.

inventory = ["beans", "coin", "tome"]

inventory.append("magic sword")

You can also use list() on another iterable such as a string or a range (we'll cover ranges later) to generate lists:

>>> list("Hello")

['H', 'e', 'l', 'l', 'o']

>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> letters = ['a', 'b', 'c', 'd']

>>> letters.append('e')

>>> print(letters)

['a', 'b', 'c', 'd', 'e']

### .insert(), POP(), .index(),. Sort()

>>> names = ["Bethany", "Alex", "Grae"]

>>> **for** name **in** names:

>>> greeting = "Hello " + name

>>> print(greeting)

### What are objects?

Of course, saying "everything in Python is an object" doesn't tell you much about what an object actually is. Python **objects** are collections of attributes. Each attribute has a name and a value. You might be thinking that attributes of an object seem a lot like the items of a dictionary. You'd be right.

In addition to attributes, each object has a unique id, which you can access with the built-in [id() function](https://docs.python.org/3/library/functions.html#id), and a **type**, which you can access with the built-in [type() function](https://docs.python.org/3/library/functions.html#type).

We can get a list of the names for each attribute of an object by passing the object into the built-in dir() function. Let's try that with the integer 1 to see what 1 really looks like.

That's... a lot of attributes for something you might think was just the simple number "one". What are these mysterious attributes?

### Accessing and using attributes

Well, let's find out what these attributes are by accessing them. You can access object attributes using dot notation: follow the object with a . and then the name of the attribute you want to access. Let's try this with a string. If we call dir() on a string we see one of the attributes is upper. Let's access that.

>>> "hello".upper

<built-**in** method upper of str object at 0x1015715e0>

Let's think about abstraction for a moment. Consider the statement 1 + 2and the statement 1 + 'pizza'. We expect the first statement to work because both operands are numbers, and we (rightly) expect the second statement to fail because you can't add a number and a string. The operands in the second example aren't the same type of thing.

What do we mean when we say a number and a string aren't "the same type"? Similarly, what do we mean when we say something like "integers are a type of number"?

### Abstraction and classes

In both cases above we're making abstractions. Each integer has a whole bunch of attributes it shares with other integers, attributes it doesn't necessarily share with a string. For example numbers can be even or odd, composite or prime, but those concepts don't even make sense when talking about strings.

Similarly, different types of numbers share some attributes but not others. All integers are divisible by one, but other types of real numbers (like floats) don't have that attribute. "Number" is an abstract class that both integers and floats belong to, even though they're different from one another.

In all of these cases we can intellectually abstract away from specific instances of something (like 1 and 10 and 42) to create a "class" (in this case, integers) where we list out all the attributes that the instances share with one another. Whenever we chose a specific instance of the class you know that the instance inherits all of the properties of the class it belongs to.

All of this talk about abstraction is a bit heady, so let's look at the way Python creates different types of numbers using **classes** as a concrete example.

Integers and floats are both a type of number. Python has a [numbers class](https://docs.python.org/3/library/numbers.html#numbers.Number)that ints and floats belong to. They inherit the attributes of that class. Rational numbers are another type of number. Integers are rational numbers but floats aren't, so integers inherit the attributes of the [rational class](https://docs.python.org/3/library/numbers.html#numbers.Rational) but floats don't. You can read more detail about this and related numeric classes [here](https://docs.python.org/3/library/numbers.html), but the takeaway is that Python classes give us a way to define a type of object and allow all objects of a certain type to inherit the attributes of their class. All the unexpected attributes you saw on the object 1 above are inherited from the classes it belongs to.

### Custom objects

As a data scientist you can keep on using all of Python's built-in objects and the objects you import from the standard library and other modules (more on importing modules next) without needing to worry too much about making your own custom objects. But seeing how custom objects work will help you understand objects in general, so we'll look at an example of setting up your own custom objects.

Let's say you want to model a bunch of quarks. Each of your quarks will have its own color (red, green, or blue) and flavor (up, down, strange, charm, top, or bottom). All quarks will have the same baryon number (1 / 3) and will have a method to interact with another quark by exchanging colors (modeling the strong interaction).

Before you can start churning out quarks you first define what you mean by quark, or, in Python terms, you need to define the class. Let's see what that class might look like

**class** **Quark**(object):

**def** **\_\_init\_\_**(self, color, flavor):

self.color = color

self.flavor = flavor

baryon\_number = 1 / 3

**def** **interact**(self, other\_quark):

self.color, other\_quark.color = other\_quark.color, self.color

**def** **\_\_repr\_\_**(self):

**return** "{} {} quark".format(self.color, self.flavor)

Let's break down this example piece by piece. In the first line we use the class keyword to start the definition of a new class. This works just like the def keyword when defining new functions. After that comes the name of our new class (Quark). It's customary to use "[CapWords](https://www.python.org/dev/peps/pep-0008/#class-names)" capitalization with custom classes. We follow the class name with parentheses containing the class we want "subclass" from. If you don't have a more specific class you'd like to subclass, the built-in object object has the most basic default attributes you want to inherit.

Inside the class definition we define three methods (\_\_init\_\_(), interact() and \_\_repr\_\_()) and the baryon\_number attribute. Each object that we create from this class will inherit these attributes. The interact()method implements the color exchange we set out to do, while \_\_init\_\_()and \_\_repr\_\_() are special double-underscore or "dunder" methods.

The \_\_init\_\_() method is special. Python automatically calls an object's \_\_init\_\_() method when it's created. That means all of the code inside runs as a part of setting up each new object. In our example we're setting two attributes: the quark's color and the flavor.

"But wait", you say, "what about the first self argument?" Good question. In an object, the self variable is used to refer to the object itself. Every method expects self as the first argument, and whenever you call a method self is passed in behind the scenes without you having to explicitly include it in your method call. When you're reading code that includes self you can mentally replace that with "the particular object I'm dealing with", so in English self.color = color would read as:

*The color of the particular quark I'm dealing with is the value of color passed into \_\_init\_\_().*

Let's create a couple of quarks and play around with them.

### his method models the way quarks interact with one another by

### # exchanging color.

### def interact(self, other\_quark):

### self.color, other\_quark.color = other\_quark.color, self.color

### # The repr method controls how the object is represented by the

### # print() function and other representations of the object.

### def \_\_repr\_\_(self):

### return "{} {} quark".format(self.color, self.flavor)

### # Now that we have the class set up, let's call Quark() to create two

### # actual instances of quark objects.

### q1 = Quark("red", "up")

### q2 = Quark("blue", "down")

### # Print each object to see what they look like.

### print("q1 is a {}".format(q1))

### print("q2 is a {}".format(q2))

### # Test our interact() method by having q1 and q2 interact.

### q1.interact(q2)

### # Print them out again to see how they changed.

### print("Now q1 is a {}".format(q1))

### print("Now q2 is a {}".format(q2))

nlike color and flavor, each quark has the same baryon number, so we can set that outside the \_\_init\_\_ function just like we do for the method attributes.

The interact() method is a straightforward function that manipulates both the object calling it and the object passed in as an argument. The \_\_repr\_\_() method is another useful dunder method that tells your object how to play nice with print() and other cases that require a representation of your object.

Take a minute and go back to the interactive example above to see if you can add a spin attribute to the class definition, then run your code and see if it works, or, if there's an exception, if you can run down the cause. In physics quarks can have a spin of either 1 / 2 or -1 / 2.

# Modules

In addition to the standard library, there is a vibrant ecosystem of open-source Python modules that you can download, install, and import into your programs. The best centralized reference for these is the [Python Package Index](https://pypi.python.org/pypi), or "PyPI". For example, here is the [PyPI page for Pandas](https://pypi.python.org/pypi/pandas/0.19.2), a module you'll use heavily as a data scientist. We'll install and dig into Pandas and other packages soon.

### Importing modules

All you need to start using modules is an import statement at the top of your program. Here's how you'd import the built-in math and random modules

**import** math

It's customary to put your import statements at the [top of your files](https://www.python.org/dev/peps/pep-0008/#imports).

Once imported, you'll have access to a math module object and a randommodule object. Just like every object, these contain attributes that you can work with. Some interesting attributes are the [math.pi constant](https://docs.python.org/3/library/math.html#math.pi) and the [random.choice() method](https://docs.python.org/3/library/random.html#random.choice). Let's tinker with these

### Custom modules

Creating your own modules is easy once you're working with files on your local machine. In fact, every Python file (file with a .py extension) is also a module. The module name is the name of the file minus the .py part, so if you have a file called demo.py you can import it with import demo. All of the variables and functions in your file are available as attributes of the module.

**Dictionaries**

*Dictionaries* are another way to collect several pieces of data together. While a list stores that data as an ordered sequence, dictionaries store that data as an *unordered* collection of *key-value pairs*.

To create a dictionary, wrap key-value pairs in curly braces:

adventurer = {"name": "grae", "profession": "magician"}

### Function definitions

Let's look at a function definition:

**def** **times\_two**(num):

**return** num \* 2

# Install modules

When you installed Python 3 you also got **pip**, the [package manager for Python](https://en.wikipedia.org/wiki/Pip_%28package_manager%29), for free. You'll use pip frequently in your career to download and manage Python tools.

###### If you somehow didn't install Python 3 or your Python 3 version is lower than 3.4 you may not have pip. You can install it manually with [these instructions](http://stackoverflow.com/a/12476379), but you're better off just installing the most recent version of Python.

These instructions should work across Mac, Windows, and Linux platforms.

Let's confirm your install:

$ pip --version

pip 9.0.1 from /usr/local/lib/python3.6/site-packages (python 3.6)

If that gave you an error you don't have pip installed. Your path and versions may look slightly different. The important thing is that your Python version is at least 3.5. If instead that shows Python 2 then you may have the Python 3 version on your system as pip3. Try $ pip3 --version. If that worked then replace all the examples of pip with pip3 in the coming material.

Before moving on make sure you have the most recent version of pip by running:

$ python -m pip install -U pip

###### The instructions that follow use python and pip for consistency instead of python3 or pip3. Replace python with python3 and pipwith pip3 if appropriate for your system.

## NumPy

NumPy is the fundamental scientific computing package. Most other Python data science tools are built on top of NumPy and treat it as a dependency. You should be able to install NumPy with:

$ python -m pip install numpy

Numpy is big and may take a while to install. As of mid 2016 NumPy should also install successfully on Windows machines with the above command. Confirm your successful install by running pip freeze at your command prompt. This will list out all of the Python packages you've installed and should include NumPy.

## Pandas

Pandas is the Python library for data manipulation. It gives you custom objects (particularly the data frame) that you'll use every day.

$ python -m pip install pandas

Run pip freeze to confirm your install.

## Matplotlib

Matplotlib is the 2D plotting library you'll use to produce many of your visualizations. You'll use it both for data exploration and for presentation.

python -m pip install matplotlib

Confirm your install with pip freeze.

## SciPy

SciPy adds algorithms, convenience functions, and is the basis for many other packages you'll install in the future.

python -m pip install scipy

Brrr, it's getting cold in here from all this pip freeze.

## Install Jupyter Notebooks

Jupyter notebooks allow you to write, execute, and visualize Python interactively. You'll use Jupyter notebooks as your primary IDE for writing Python in the bootcamp.

python -m pip install jupyter

One last pip freeze for good measure.

To use Jupyter notebooks, run $ jupyter notebook from your command line. This will start the Jupyter server and open a browser tab that hits the server. You can kill the server by closing the terminal window it's running in or with control + c on a mac. If you close your browser but the server's still running you can open a new tab to the server at the address listed in your terminal that starts with http://localhost:8888/?token=.

# Jupyter notebooks

Before going forward you should make sure Jupyter is installed on your machine. You can do this with jupyter --version from your command line.

This particular notebook is hosted in a server far away from your local machine, but you'll want to be able to run notebooks locally for development and reporting purposes, and to copy and run notebooks other people create.

## Launching Jupyter

Jupyter is launched from the command line. All you have to do is use your command line to navigate to the directory in which you want to work and run:

jupyter notebook

If you want to open an already existing notebook run:

jupyter notebook [your/particular/filepath]

Launching a notebook creates a **kernel** that will run your work on your machine. If you opened a directory you'll see the directory, while if you opened a file you'll see the new file.

Also of note, Jupyter notebooks can run in many different languages. It was originally developed for Python, but now can support R and many others. You select the language when you create a new notebook.

## Cells

Notebooks are divided into cells. Below is an empty cell

Cells will be the home to all of the content you put into notebooks. Below we have a cell with some simple code.

In [1]:



1

message = 'Hello! This string is stored in the variable "message"'

Now, when you just write code into a cell nothing really happens. You have to 'run' or execute it. To execute a cell you can press the run cell button from the menu bar at the top of the notebook (it's the triangle pointing to the right with a line next to it) or you can press shift and enter when the cell is highlighted.

Cells can be selected in two ways. First, a cell is highlighted but inactive if there is a box around the cell (usually in green) but no cursor. If there is a cursor then the cell is active and you can input into it.

A notebook can have any number of cells you need. There are several ways to add cells. When you run the last cell in your notebook a new cell is automatically added below. Also, if you have a cell highlighted but inactive, you can add a cell above or below by typing a or b respectively. To delete a cell double-tap d.

Now, every notebook runs in a single Python environment.

That means a couple of things. Firstly, the code is executed in the order you execute in. Variables and functions and the like are stored for the whole notebook. So message still means something even here:

In [2]:



1

print(message)

Hello! This string is stored in the variable "message"

You can also get into trouble with this. If you change code in your cell and run it multiple times you can change the variables or outputs in a way you don't intend. Same thing if you run cells out of order. That's why it's **VERY IMPORTANT** that you keep all of your code ordered and runable.

Basically what this means is that you should be able to restart your notebook and run every cell in order to get to the result you want. If you have to run things out of order or rerun cells, that's likely going to cause problems both for you and anyone else reading your code. If you get confused about what was executed in what order it can be useful to re-run everything from scratch by selecting the "Kernel" menu above and choosing "Restart and run all". This will basically reboot and rerun your entire notebook from top to bottom.

Use cells intelligently throughout your notebook. Many people think of a cell as a step. You should use a cell to write a function or run a model, but don't try to do too much in a single cell. Keep it simple and your code will be more modular and easier to read and reuse.

## Markdown

So far we've only talked about code cells, but there are other kinds of cells as well. The main other kind of cell you should know about is **markdown**. This cell is actually a markdown cell. You can double click it to see and edit the raw markdown, then run the cell again to see the markdown rendered.

When you're using notebooks to write reports, you want to be able to include prose that isn't very appropriate for code comments. Markdown is useful for this.

By default every new cell is a code cell. To convert a cell to markdown, highlight the cell, make sure it's inactive, and type 'm'. It's that simple.

Jupyter supports full markdown so you can do all sorts of things, like

Italics

**Bold**

# Big Headers

#### And small headers

both inline code

*# And syntax-highlighted*

multiline()

codeblock = "samples"

Embedded HTML like key tags and tags with custom CSS.

And LaTeX style equations: eiπ+1=0eiπ+1=0

Probably most usefully, you have [links](http://jupyter-notebook.readthedocs.io/en/latest/notebook.html).

That link above is the nice and readable documentation for Jupyter. It's worth looking through if you have any further questions about notebooks. For more detail on markdown specifically, [see here](http://jupyter-notebook.readthedocs.io/en/latest/examples/Notebook/Working%20With%20Markdown%20Cells.html).

And look, our code still works, and message is still defined :)

In [3]:



1

print(message)

Hello! This string is stored in the variable "message"

That should be enough to get you started with Jupyter. If you have any questions after reading [the docs](http://jupyter-notebook.readthedocs.io/en/latest/notebook.html), follow up with your mentor!

We're going to start with a package called [NumPy](https://docs.scipy.org/doc/numpy/reference/). NumPy is the basic package for doing slightly more advanced math and storing data in an analytics-friendly form. We'll make use of NumPy throughout this prep course and the bootcamp.

You should have installed NumPy on your local environment in the previous Unit. If you don't yet have NumPy installed, install it now with pip install numpy.

## Element-wise and Aggregator functions

NumPy allows you to do these computations in two ways: with element-wise functions that process array elements one at a time and then return a new array, and with aggregator functions that process the array into a single value the function returns.

Note that these methods return arrays of the same length as the input array, just like the built-in Python function map(). Use element-wise functions when you want to transform each individual element in an array and get back a collection of all the results.

# Pandas Data Frames

## The Data Frame

The data frame is like a NumPy array, with a few additional features like column names and row indexing. It is probably the primary way data scientists handle data. You can create a data frame in many different ways, either from csv files, by querying databases, or explicitly. For your first data frame, let's recall the 2-dimensional array we created in the previous assignment. To create a data frame use the pd.DataFrame() function and pass in a NumPy array:

my\_array = np.array([[0, 1, 2, 3], [4, 5, 6, 7]])

df = pd.DataFrame(my\_array)

df

Columns are labeled with column names, rows with an index number (starting with zero by default). You can set both column names and indexes explicitly during the creation of the data frame or after the fact. Let's set both for df from above.

df.columns = ['first', 'this', 'that', 'last']

df.index = ['row\_1', 'row\_2']

df

ou can also set column and index names through the column= or index= keyword arguments when you call the pd.DataFrame() function to initially construct the data frame.

df2 = pd.DataFrame(

my\_array,

columns=['first', 'this', 'that', 'last'],

index=['row\_1', 'row\_2'])

df2

#### Adding More Data

df['COLUMN\_NAME'] = [LIST\_OF\_VALUES]

# Pandas - Selecting and Grouping

## Basic Selects with .loc and .iloc

[.loc](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.loc.html) is a selector that indexes over rows and columns. It selects over the row index first, then the column name (if included). [.iloc](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.iloc.html) does the same thing but over indices. For example, to select the row for 'George' in our purchases data frame, we just pass the string 'George' in to purchases.loc with bracket notation:

To select the column 'country' we would use:

purchases.loc[:, 'country']

As we mentioned above, you can also do integer indexing, as done on lists, over both rows and columns using .iloc. For example:

purchases.iloc[1:3, 1]

## Conditional Selection

You can also use .loc for conditional selection, or selecting all the entries that meet a given criteria. This will use **lambda**, which is a construction that allows for defining anonymous, unnamed functions at runtime. We use the lambda function to create a condition on the row or column.

let's say we want all the columns for individuals who made more than one purchase. That ends up being a relatively simple line of code.

purchases.loc[lambda df: purchases['items\_purchased'] > 1, :]

purchases[purchases['items\_purchased'] > 1]

Groups

purchases.groupby('country')

But wait, when you run that line, it doesn't return your data any more. It returns a line that references a grouped object, but not the object.

That's because if we want it to return something we have to do something on those groups. There are several methods that you can use here. Some are built in like .sum() or .count(). For even greater possibilities you can use .aggregate(numpy\_function). Let's use this to find out which group has more page views and purchases.

purchases.groupby('country').aggregate(np.mean)

# Don't want to take the mean of all columns? Try this:

# purchases.groupby('country')['column\_name'].mean()

# Working with Files

In the examples so far we built Pandas data frames by manually typing or pasting in text. Doing that with actual data sets would be tedious or impossible, so you'll almost always be loading your data into Pandas from files (like CSV files), from the web using APIs, or from databases and other large data stores.

We'll cover APIs and databases in the bootcamp. In this assignment we'll show you how to load CSV, JSON, and XML files you've generated or downloaded into Pandas. We'll also briefly discuss working with files using vanilla Python.

When your files are well-encoded and correctly formatted and your data is clean, loading files is pretty straightforward. But that's not always the case. Dealing with malformed data is one of the largest challenges of being a data scientist, and there's no one-size-fits-all solution. Here we're going to cover the easy cases and point you towards further reading if you need it for your capstone. Your goal in this assignment is not to memorize the process for dealing with every use case, but to get an overview of the process, be comfortable working with simple files, and have a reference that you can come back to later if and when you need it.

We'll give you several example files to play with in this assignment. Go ahead and create a directory on your computer now where you'll store these files. You can download a zip with each example [here](https://gist.github.com/Grae-Drake/89c19c54077b818ea69d314e74bb6fbf), or download each file one by one below.

## Opening files with Pandas

### CSV files

**CSV** files, or "Comma Separated Values" files, are a common file format for **structured** tabular data. They're easy to produce from Microsoft Excel, Google Sheets, and Python scripts. They're also simple to work with: you can open and read them in any text editor. If you do that you'll see that a CSV is just a text file where the values in each column are separated by commas, and the rows are separated from one another with newlines. Because of their convenient structure and simplicity CSV's are extremely common to work with.

Here's a sample CSV file with the same [purchase data](https://gist.githubusercontent.com/Grae-Drake/89c19c54077b818ea69d314e74bb6fbf/raw/b259f6241c5ca9c5a3b6235bdfb260eb0d99314f/purchases.csv) you saw in the previous assignment. If you don't have it already, save a copy of this file by right-clicking in your browser and choosing "Save As". Name it purchases.csv and place it in the same directory as your Python files.

The basics of loading a CSV into Pandas are simple. You can load the purchases.csv file you just saved with this one-liner:

>>> **import** pandas **as** pd

>>> df = pd.read\_csv('purchases.csv')

he Pandas read\_csv() method takes a string representing the path to the file you want to read and returns a data frame object. In the example above the CSV file we're working with is in the same directory as the script we're running. If you saved your file elsewhere or with a different name you'll have a different path than 'purchases.csv'.

Try loading purchases.csv in your own Jupyter notebook now. If your environment isn't set up for that yet or you're running into trouble you can play with this [interactive example](https://trinket.io/library/trinkets/27033bfb61).

The only required argument for read\_csv() is the file path, but there are dozens of optional keyword arguments available that are useful in different contexts. You can read about those in [the documentation](http://pandas.pydata.org/pandas-docs/stable/io.html#csv-text-files).

Working as a data scientist you'll frequently want to output data to a file. CSV's are likely your best bet for that. To create a CSV file and write your data frame to it use .to\_csv():

>>> df.to\_csv('my\_data.csv')

The only argument passed in to .to\_csv()above is the path for the file you'd like to output, but there are a number of [optional keyword arguments](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html) you can use to tweak your output file. You can also omit the path entirely to skip the file creation and return a string that you could paste or pipe elsewhere.

Time for you to give this a shot. Choose a file name and location where you'd like to output a CSV. Look over the .to\_csv() options linked above and choose some options you'd like to tinker with. Then, using the appropriate path, your data frame in your Jupyter notebook, and the options you've chosen, create a CSV file. Find and open your file using your favorite text editor. Did it work? Does everything look like you'd expect?

### JSON

While CSV files give you a nice (but rigid), built-in tabular structure with rows and columns, other common formats like JSON and XML allow for more customizable and flexible data storage. Unlike raw unstructured text, JSON and XML do have some structure requirements and are known as **semi-structured** files.

This flexibility of semi-structured data often comes at the cost of additional complexity. JSON data can be deeply nested and may take substantial processing before you can get it into the form you want to work with. We'll cover that in more detail in the bootcamp. For now we'll give you the same purchases data you've been using in JSON format. If you don't already have it, download this [purchases.json](https://gist.githubusercontent.com/Grae-Drake/89c19c54077b818ea69d314e74bb6fbf/raw/27aba85593a688b47ec141d0d6e7f60a9e9d33a9/purchases.json) file.

**JSON** stands for "JavaScript Object Notation" and is a way to represent a JavaScript object as a string. "Objects" in JavaScript are just collections of key-value pairs, exactly like Python dictionaries. Almost everything you know about Python dictionaries translates directly to JavaScript objects, so you can imagine that "JSON" stands for "Python Dictionary Notation" and treat JSON as if it's a string representation of a Python dictionary.

Like CSV files, JSON files are human readable and you should be able to open and explore them with your favorite text editor. This is particularly useful when you work with new JSON and you aren't sure how it's structured. Opening files in your text editor can get tricky with very large JSON files which are too big to fit in your computer's memory, but we won't worry about that for now.

You can create a data frame from a JSON file with read\_json():

>>> **import** pandas **as** pd

>>> df = pd.read\_json('purchases.json')

Again, the only required argument is the file path, but you can pass in other keyword arguments to run read\_json() with different options. [See the docs](http://pandas.pydata.org/pandas-docs/stable/io.html#json)for more details, and look into pandas.io.json.json\_normalize() for [normalizing](http://pandas.pydata.org/pandas-docs/stable/io.html#normalization) nested JSON.

Try loading purchases.json in your own Jupyter notebook, or see this [interactive example](https://trinket.io/library/trinkets/c8aef0fa91) if needed.

You should generally prefer CSV for outputting your data frames into files, but you can use .to\_json to output a data frame as a JSON file:

>>> df.to\_json('my\_data.json')

The more common use case is to send JSON data over the web. For that we'd call to\_json() without a path argument to create a JSON string that we'd later process and send:

>>> serialized\_purchases = df.**to\_json**()

We'll cover networking and using JSON like this in the bootcamp. If you'd like a preview of that check out the **Requests** module, [the only Non-GMO HTTP library for Python](http://docs.python-requests.org/en/master/).

### XML

**XML**, or "eXtensible Markup Language", like JSON, is a hierarchical semi-structured data format. They are both widely used to transfer data over the web, but the newer JSON format is more common these days than the older and clunkier XML format. If you've worked with HTML then the XML syntax of opening and closing tags wrapping, or "marking up", data will be familiar. Here's an XML file with our purchases data: [purchases.xml](https://gist.githubusercontent.com/Grae-Drake/89c19c54077b818ea69d314e74bb6fbf/raw/a0b46f7f996dd0c5e6a7bfdd9af192108f3f3060/purchases.xml).

Pandas doesn't have an XML equivalent to read\_csv() and read\_json, so we'll use the xml module from the Python standard library to read in XML files and convert them to an element tree. Once we have an element tree we'll manually process it into a list that we can feed into Pandas.

# Import Pandas and a part of the xml module.

import pandas

import xml.etree.ElementTree as ET

# Load and parse the XML file into a tree.

tree = ET.parse('purchases.xml')

# Find the root of the tree. This is the node of the tree where we'll

# start our iteration.

root = tree.getroot()

# Define a custom function to loop over our tree, extract values, and

# return a two-dimensional list.

def xml\_to\_list(root):

result = []

for row in root:

row\_list = []

for column in row:

row\_list.append(column.text)

result.append(row\_list)

return result

# Feed our two-dimensional list into Pandas.

df = pandas.DataFrame(xml\_to\_list(root))

print(df)

The custom xml\_to\_list() function above is designed for the specific XML file we're working with. If you're working with differently structured XML then you'll need to iterate over your XML tree differently.

We aren't going to go into detail here about element trees and iterating over tree objects. You can read more at the official [xml.etree.ElementTree tutorial](https://docs.python.org/3.6/library/xml.etree.elementtree.html)if you like. If you find yourself working with XML in any detail you'll want to check out the [lxml package](http://lxml.de/index.html), a fast and useful collection of tools for working with XML.

As a data scientist you frequently don't get to choose the format of data files you're given, which is why we're covering XML here. When generating data files you should probably avoid XML because it can take more work to process. CSV's are the top choice for writing data to a file. If you need a semi-structured format, prefer JSON over XML.

## Python open()

Each of the file loading techniques we covered above are package-specific ways to open and load files of one particular type (CSV, JSON, XML). Python offers a more general-purpose way to open any files you like with the built-in open() function. Here's a simple text file for you to play with if you don't already have it: [poem.txt](https://gist.githubusercontent.com/Grae-Drake/89c19c54077b818ea69d314e74bb6fbf/raw/27aba85593a688b47ec141d0d6e7f60a9e9d33a9/poem.txt)

Let's open the poem.txt file, create a file object, and print out the

# file text line by line.

# Note the poem.txt tab above.

with open('poem.txt') as poem\_file:

text = poem\_file.readlines()

print("This file is {} lines long".format(len(text)))

for line in text:

print(line)

# Try loading and printing the purchases.csv file line by line just

# just like we did for poem.txt.

In the example above we use the [open()](https://docs.python.org/3/library/functions.html#open) function to create a **file object** that we can then work with. We then use the .readlines() method of the file object to create list of strings, where each element of the list is a line of text from our input file. You can read more about .readlines() and other file object methods in the [I/O documentation](https://docs.python.org/3/library/io.html#i-o-base-classes).

Opening a file with open() will leave it open until you close it. The .close()file object method will close a file. It's super easy to forget to manually close files, which can keep resources tied up and cause unexpected trouble. Luckily, Python gives us the [with](https://docs.python.org/3/reference/compound_stmts.html#with) statement you see used above so we don't have to remember to use .close(), because files opened in a withstatement will automatically be closed once the with statement exits. Using with when manually opening files is best practice and you should plan on doing that each time you use open() unless you have a compelling reason not to.

Almost all of the data files you produce as a data scientist will be CSV or JSON files, so we won't cover using built-in Python I/O to write to files. If you'd like additional resources on any of these topics check out the [reading and writing files](https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files) section of the official Python tutorial.

## A word about encoding

All of the files you've worked with seem to be made of human-readable characters, but when you get down to it everything is stored as bits, collections of ones and zeros. Which of course raises the questions: how do you know which characters you can use, and how do you translate the ones and zeros into those particular characters?

Today there is a clear answer: Unicode and UTF-8. All strings in Python 3 are **Unicode** strings, and [UTF-8](https://en.wikipedia.org/wiki/UTF-8) is the default encoding Python uses whenever possible. However, you're certain to run across files created with different encodings, whether that's because the files are old, the software used to create them is old, or because the [proprietary software used to create your CSV files](http://stackoverflow.com/questions/508558/what-charset-does-microsoft-excel-use-when-saving-files) is mired in legacy cruft. Unfortunately, it's not possible to automatically determine a file's encoding and then decode it correctly. When encountering encoding errors you'll need to make educated guesses about the likely encoding and use trial and error to test.

As hinted above, Microsoft Windows is a big culprit here. English-language versions of Windows use the [cp1252](https://en.wikipedia.org/wiki/Windows-1252) encoding, Cyrillic versions of Windows use [cp1251](https://en.wikipedia.org/wiki/Windows-1251), and so on. Here is a full reference list of [Windows "Code Pages"](https://en.wikipedia.org/wiki/Windows_code_page), and here is a larger list of [historical encodings](http://unicodebook.readthedocs.io/historical_encodings.html). If you're unable to manually determine the encoding of your file you can attempt a statistical detection with [Chardet](https://pypi.python.org/pypi/chardet).

Unicode is a deep topic that's closely tied to the history of computing. If you're interested in a quick practical and historical overview the [Unicode HOWTO in the Python documentation](https://docs.python.org/3/howto/unicode.html) makes for excellent reading.

# Data visualizations with matplotlib

# Basic Plot and Scatter

Matplotlib is the fundamental plotting and visualization library for Python. It can create a myriad of plots (a plot is a way of visually representing data), and it is actually the foundation for many of the other plotting packages in Python. For this section, we're going to primarily focus on the **pyplot** portion of the library. The convention to load pyplot is:

import matplotlib.pyplot as plt

## Line Plots

Let's start by just putting a list of values into the plot function and see what it does.

If you're using Jupyter notebooks you'll want to use the magic %matplotlib inline to get your plots rendering well. It's good practice to always include that line in your first cell when working with Jupyter notebooks, along with loading your packages. This statement makes sure your images will always generate in the notebook instead of possibly as popups and prevents a slow matplotlib compilation from giving you rendering errors.

Also, you won't always need plt.show() at the end of a cell in a notebook, but it's good practice to keep it there. It will be useful to ensure that all the options and specifications you add to your plot are properly rendered. We'll explain more later.

plt.plot(df['rand'], color='purple')

plt.ylim([-0.1, 1.1])

plt.ylabel('Values')

plt.title('Random Series')

plt.show()

## Scatter

Let's look at one other common kind of plot first, the scatter. This works much the same as .plot() and is called with .scatter() instead.

# Subplots

Subplots are matplotlib's way of generating multiple plots in a single figure. You break up your figure into multiple distinct areas and generate different plots in each area. This can be extremely useful for generating a series of plots and presenting them in a clearly associated way. It also can make it easy to transfer multiple visuals as a single unit. Note that the visuals below are a single image.

Now let's put these two plots together in one object as two sub plots. For that we'll use the subplot functionality of matplotlib. To do that, we need to define the subplot before generating each plot. That can be done using plt.subplot(), which takes three arguments. The first two parameters define the dimensions of the plot, while the third identifies the subplot you're creating. Like so:

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.plot(df['rand'], color='purple')

plt.ylabel('Values')

plt.title('Random Series')

plt.subplot(1, 2, 2)

plt.plot(df['rand\_shift'], color='green')

plt.ylabel('Shifted Values')

plt.title('Shifted Series')

plt.show()

So we created a 1 high, 2 wide subplot with the random series on top and the shifted series below. Great! But how did it really work?

Well, the plt.figure() function sets the size of the figure we're about to create, taking a tuple with the x and y dimensions we want. Then each plot gets created using the plt.subplot()function. This specifies which plot we're going to work on. You generate a grid with the first two parameters, with rows followed by columns like the pandas selecting logic, then pick which subplot within the grid to make via the third parameter. That parameter increases numerically throughout the grid, so this only works for subplots with less than 10 figures (though there are other ways to handle such cases).

You could also add different types of plots as subplots, so we could have one plot in this figure be a scatter while the other is a line plot. Let's also add one other feature, tight\_layout. This creates additional spacing around the subplots. It can be extremely helpful when dealing with many subplots or when the figure starts to look a little claustrophobic.

# Statistical Plots

## Histogram

The **histogram** is a great tool for showing the possible values of a variable, as well as how common those values are.

Let's make one now. First we'll generate some random variables from a normal distribution (don't worry if you aren't familiar with this, we'll talk about more in the next unit), and then generate the plot.

Now that we have a histogram let's go into a little more detail about how it works. It starts by dividing the values in the input into bins. These bins are evenly sized ranges that have an upper and lower bound. The histogram then counts how many values are in each bin and plots a bar for that count for each bin. That gives you a sense of the density of the variable across its range. Histograms are fantastic for visualizing the distribution of a variable.

**Bin count and placement**

By default, matplotlib distributes all values into ten bins and chooses the best placement for those bins. That might not be the number you want and is totally adjustable with the binskeyword argument.

In [3]:



1

*# Random data.*

2

x = np.random.normal(10, 5, 1000)

3

​

4

*# Build our histogram. Let's go ahead and set the color too.*

5

plt.hist(x, bins=40, color='red')

6

plt.title('Default Bin Placement Demo')

7

plt.xlabel('Random Values')

8

plt.show()We can specify bin placement by passing in an array of specific bounds for your bins rather than just the total number you want. Let's do that and create a histogram with bins at exactly the integer marks from -10 to 40. While we're at it we'll add a second histogram to our figure.

### Normalizing histograms

Did you notice it's difficult to compare the distributions of the two variables with these two histograms? The scales of each variable are incompatible, so their heights are also incompatible, with the red histogram dominating the visualization. Normalization allows you to solve this problem by rescaling the height of the bins so that the total area under each curve sums to 1.

This is useful when you have variables of different sizes (one with tens of thousands of observations, one with hundreds, for example) and you wish to compare their histograms.

Here is a plot with the same histograms, but this time they're normed.

# Same data, this time normed.

plt.hist(x, normed=True, color='blue', bins=np.arange(-10, 40), alpha=.5)

plt.hist(y, normed=True, color='red', bins=np.arange(-10, 40), alpha=.5)

plt.title('Normed histograms')

plt.xlabel('Random Values')

plt.show()

## Boxplot

Another useful visualization is the boxplot. You can create one with the plt.boxplot() method, which takes an array and a series of other options as parameters, as well as the standard pyplot options for things like color.

A boxplot is a relatively simple visualization. Let's generate one and then go through its features. All we need is an array or variables. We'll use the same distribution as before.

## Data Science Workflow

Visualizations are often the first step in investigating how variables are behaving. You can easily see things about the distribution of data (does it match certain known behaviors or patterns) and if there are any anomalous or otherwise extreme data points that need specific investigation. These will be key factors in determining how to approach the problem. Throughout this course, almost every major project will start with visualizations.

Visuals will also play a key role in the evaluation of any other data science work, particularly model building. As we progress through this course we will go over a variety of techniques for building and evaluating models, but one of the techniques we will consistently come back to is visualizing projections or results.

## Choosing the Right Visualization

Another key part of this process is choosing the right visualization. Every type of chart of graph or visual is best suited to a specific context. Line graphs are great for tracing out a trajectory over time. Scatter plots show the relationship between two variables. There are several other visualization forms we'll introduce throughout the course, each with their own particular benefits. You'll learn more about these as you work through the course. As you do, always ask yourself if the visualization you're using is as simple and clean as it could be, and make sure you're explaining the factors based on the kinds of relationships that your chosen type is best suited to explain.

This unit is only the beginning of your adventures in visualization, but it is without a doubt one of the key skills to master in order to become a great data scientist.

Units 3

# Statistics for Data Science

Statistics, or the collection, analysis, and interpretation of data, is an integral part of the data scientist's knowledge base. In this Unit you'll learn the fundamentals of statistics and probability and begin applying those fundamentals using Python.

Mastering the topics covered in this Unit will give you an entry into the language and thought processes data scientists use to solve complex problems. You will build on this foundation in logic and mathematics to master more advanced statistical methods later on.

# Summarizing Data

 Estimated Time: **5-6 hours**

We'll start this introduction to statistics by describing the process of going from large datasets to compact summaries that people can understand and use. We will talk about how data scientists use statistical summaries to describe larger groups of interest, called populations, and how they choose the best statistics to summarize different aspects of a dataset.

# Population vs Sample

 Estimated Time: **15 minutes**

A major purpose of data science is to give us information about some group, known as a population. This population can be all the people living in a country, all the purchases made at a store, or any other unit from which information can be drawn. Often, it is difficult, prohibitively expensive, or simply impossible to get data from all members of a population. Imagine trying to get a questionnaire to every person in a country to learn about their product preferences- it can't be done!

Instead, we randomly extract a subset from the population (a random group of people, a random selection of purchases), called a sample, that we can study in detail to learn about the population as a whole.

For example, imagine we have a 100 pound bag of M&Ms and we want to know the percentage of green M&Ms in the bag. The bag of M&Ms would be our population. While it would certainly be possible to count every single M&M, it would take a very long time, not to mention being potentially quite messy. As an alternative, we could shake up the bag, pour out half a pound of M&Ms, and count the M&Ms in that sample. If our half pound of M&Ms was 8% green, then it is pretty safe to say that the whole 100-pound bag is also 8% green M&Ms.

Statisticians take data about a sample and reduce the complexity of that data into understandable and accurate summaries, known as statistics. Statisticians use the **sample** statistics to infer information about the entire **population** from which the sample is taken.

# Measures of Central Tendency

 Estimated Time: **1 hour**

Statistics can describe either an individual variable or the relationships among two or more variables. A variable represents information about a particular measurable concept (temperature, price, size, etc). Each measurement within a variable is called a datapoint. Let's make a dataframe in Python with one variable, age, that we can play with later on.

**import** pandas **as** pd

*# Make a blank data frame.*

df = pd.DataFrame()

*# Populate it with data.*

df['age'] = [28, 42, 27, 24, 35, 54, 35, 37]

When describing individual variables, the two characteristics of most interest are the central tendency and the variance. We'll cover central tendency in this assignment and variance in the next.

## Central Tendency

The central tendency describes a point around which datapoints in a variable cluster. Central tendency can be measured in a number of ways. The most common measures are the mean the median, and the mode.

### Mean

The mean represents the average value within a variable, and is computed as the sum of the individual datapoints in a variable x divided by the total number of values in a variable n. It is sometimes also referred to as the "expected value" of a variable.

**mean** = sum(x) / **n**

Here are two ways you can compute the mean of our age data, first with built-in Python functionality and then with NumPy.

*# Using built-in Python functionality.*

sum(df['age']) / len(df['age'])

*# Using NumPy*

import numpy **as** np

np.mean(df['age'])

The mean is easy to understand and commonly used, but it's sensitive to extreme values: one abnormally large value in a set of otherwise small values will cause the mean to become much larger.

### Median

The median represents the middle value in a variable when the values are ordered from least to greatest. If there are an odd number of values in a variable, then the median is the middle value, and if there are an even number of values in a variable, the median represents the average of the two middlemost values.

Here's how you can compute the median of our age data using the statistics module of the Python standard library or NumPy.

*# Vanilla Python, using the built-in statistics module.*

import statistics

statistics.median(df['age'])

*# Using NumPy.*

import numpy **as** np

np.median(df['age'])

The median, like the mean, easy to understand, and has the added benefit that it isn't sensitive to extreme values. However, the median has fewer useful mathematical properties than the mean as we'll see later.

### Mode

The mode represents the value in a variable that occurs the most frequently.

*# Return the mode using the statistics module.*

**import** statistics

statistics.mode(df['age'])

If two or more values in a variable occur with equal frequency, there will be multiple modes. Note the code above will raise a StatisticsError if you run it on data containing multiple modes. Receiving this error, or generating and inspecting a list of counts beforehand, will show whether there is more than one mode to look for.

*# Generate a list of unique elements along with how often they occur.*

(**values**, counts) = np.unique(df['age'], return\_counts=True)

*# The location in the values list of the most-frequently-occurring element.*

ind = np.argmax(counts)

*# The most frequent element.*

**values**[ind]

The code above will handle data with multiple modes without raising an exception, but you'll get back just the first mode. If you want to push your understanding of Python you can challenge yourself to revise it to give you all of the modes.

### Quick note about bias

The mean, median and mode calculated from a **sample** are considered unbiased estimates of the **population** mean, median and mode. An estimate is "unbiased" if, across multiple representative samples, the sample estimates converge on the population value. A "biased" estimate would converge on a value that was either higher or lower than the population value.

Unbiased estimates are useful because they let us use a small group of observations to make generalizations about a much larger group.

# Measures of Variance

 Estimated Time: **1 hour**

## Variance

While measures of central tendency are important, on their own they are not enough to describe a variable. Another piece of information is equally vital: variance.

The variance of a variable describes how much values differ from the central tendency, and how much they differ from each other. If all the values in a variable are close to the central tendency, then variance is said to be low. If values in a variable vary widely, with some far away from the central tendency, variance is said to be high.

Another way to think of variance is that it gives a clue to how valuable each individual datapoint is within a variable. If variance is low and most datapoints are similar to the central tendency, then each individual datapoint provides little new information about the concept being measured. If variance is high, then each individual datapoint is more likely to provide unique information about the concept being measured.

While some people tend to be more interested in measures of central tendency like the mean, data scientists are usually just as excited about the variance. This is because data scientists generally want to answer questions about why things are different from each other: Why is this store's profit margin so much higher than the others? Why is this medicine's rate of side effects so much lower than others in the same trial? Why do some customers spend so much more time on the company website? A variable with lots of variance provides information about differences between observations that data scientists can use to understand and predict future outcomes.

Variance v is measured as the sum of the squared difference of each individual datapoint x from the mean, divided by the number of datapoints n minus 1.

v = sum((x - mean) \*\* 2) / (n - 1)

There are two peculiarities about how variance is calculated. First, why is the difference between x and the mean squared? And second, why divide by n - 1 and not n?

One point to note is that the average of the differences between each value and the mean is zero (approximately half the differences would be negative and half positive, and cancel each other out), which isn't very useful. Squaring the differences makes all values positive so that the negative values no longer cancel out the positive ones. Of course, we could just take the absolute value of each difference, which would be another way to solve the problem of negative and positive values canceling each other out. It turns out that squaring the differences has other mathematical advantages over taking the absolute value, however, that we will discuss later.

Estimates of **sample** variance divide by n - 1 because dividing by n would underestimate the **population** variance, creating bias. We'll cover bias in more depth later.

We can calculate the variance of an array with [numpy.var()](https://docs.scipy.org/doc/numpy/reference/generated/numpy.var.html) or with pandas syntax df['column'].var.

df['age'].var()

np.var(df.age)

## Standard Deviation

The most common estimate of variability used by statisticians is the square root of the variance, called the standard deviation. The standard deviation has some useful mathematical properties that we will review in the Central Limit Theory lesson later in this Unit.

s = v **\*\*** 0.5

NumPy gives us the useful [np.std()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.std.html) function for working with standard deviations. A tricky default in numpy is to calculate the population standard deviation, dividing by n, rather than the sample standard deviation, dividing by n - 1. To calculate the sample instead of the population standard deviation we need to manually set the "delta degrees of freedom" with the ddof named parameter:

np.std(df['age'], ddof=1)

## Standard Error

Another useful estimate of variance is the standard error, which quantifies uncertainty in the estimate of the sample mean. While the standard deviation tells us about variance in the population, the standard error tells us about the precision of our sample mean estimate. One example of standard errors at work is poll results, where they are called the "margin of error". For example, a poll might report that 44% of respondents were in favor of measure X, with a margin of error (standard error) of 3%. In other words, if the poll were run over and over again with new samples of respondents, the average response would fall between 41% (44-3) and 47% (44+3). Smaller standard errors mean more precise estimates.

The formula for the standard error se of the mean is the standard deviation of the sample s divided by the square root of the sample size n.

se = s / (n \*\* 0.5)

Using Python and NumPy, this is:

np.std(df['age'] ,ddof=1) / np.sqrt(len(df['age']))

Let's examine sampling from different distributions of low and high variance. We'll create two variables, one with low variability and one with high variability, and see how they differ.

*# First, create an empty dataframe to store your variables-to-be.*

pop=pd.DataFrame()

*# Then create two variables with mean = 60, one with a low standard*

*# deviation (sd=10) and one with a high standard deviation (sd=100).*

pop['low\_var']=np.random.normal(60, 10, 10000)

pop['high\_var']=np.random.normal(60, 100, 10000)

*# Finally, create histograms of the two variables.*

pop.hist(layout=(2, 1), sharex=**True**)

plt.show()

*# Calculate and print the maximum and minimum values for each variable.*

print(pop.max())

print(pop.min())

The variable with high variance has a much wider range of possible values than the variable with low variance. If these variables represented two populations we wanted to study, we would take samples from each, then generalize from those samples to get information about the populations. Let's try that next.

*# Take a random sample of 100 observations from each variable*

*# and store it in a new dataframe.*

sample=pd.DataFrame()

sample['low\_var'] = np.random.choice(pop['low\_var'], 100)

sample['high\_var']=np.random.choice(pop['high\_var'], 100)

*# Again, visualize the data. Note that here we're using a pandas method to*

*# create the histogram.*

sample.hist()

plt.show()

*# Check how well the sample replicates the population.*

sample.mean()

sample.std(ddof=1)

Since the sample is randomly drawn from the population, you can re-run the code as many times as you like and always get a new sample. Try this a few times. You will notice that the low variability samples are closer to the population mean and standard deviation than the high variability samples. Each time a sample is drawn from each population, there is a chance to draw values from the tail ends of the distribution – extremely high values, or extremely low values. Having extreme values in the sample can pull the sample mean away from the population mean. Since the high variability variable has values that are much more extreme than the low variability variable, the estimates have the potential to fall farther from the mean.

Happily, since the extreme values are spread equally across "extremely high" and "extremely low," even multiple samples from a high variability population will eventually converge on the true mean… it will just take a bit longer.

Food for thought: what would happen if you increased the sample size to 1000?

# Describing Data with Pandas

# Describing data with Pandas

So far in this lesson, we've discussed the various ways we can use statistics to describe a given dataset. Now, we're going to discuss how we can leverage the tools of data science, specifically the pandas package, to quickly and easily describe our data. This is what you'll actually be using day to day when you have to describe or summarize the data you're working with. Rather than draw out formulas or perform calculations you'll use the tools of programming to get the answers you want easily and efficiently.

## What we've seen before

We've already shown some of the basic tools. We have NumPy methods like .mean() or .std()to calculate the mean and standard deviation of our data.

In [4]:



1

data.height.mean()

Out[4]:

65.921110470114073

In [5]:



1

data.height.std()

Out[5]:

7.125274567757093

Now, there are many more methods in pandas to describe data in simple aggregative forms. Things like median and variance all have associated pandas methods. As a general rule of thumb, if you're trying to compute a standard statistical measure (the kinds of measures you could find in a statistics book somewhere) Python probably has a coded up method for it somewhere already. Usually that method will be in NumPy and pandas, but not always. It is, however, always worth a quick Google and check of Stack Overflow to see if the work has already been done before you go off and create your own functions.

## The .describe() method

So far we've mostly talked about methods with two kinds of output: it either stays the same shape with modified values (the iterative kinds of methods) or it condenses the data into a single value output (aggregative methods). There is another group of methods in Pandas, and they happen to be supremely useful for quickly and coherently summarizing data in a numeric rather than visual way.

In statistics, there are a lot of descriptive values that are often used in concert with each other. The most classic example is probably mean and standard deviation. Using the two of them together gets you a lot of information about how the data is distributed across values.

Pandas understands this. Sometimes you want more than one value, but less than all of them. You want a set of summary statistics that give you a good, standardized view into the data and its variables. Enter .describe().

In [6]:



1

data.describe()

Out[6]:

|  | **height** | **weight** |
| --- | --- | --- |
| **count** | 200.000000 | 200.000000 |
| **mean** | 65.921110 | 181.416225 |
| **std** | 7.125275 | 25.996945 |
| **min** | 49.268448 | 126.352513 |
| **25%** | 61.194613 | 162.815738 |
| **50%** | 65.567748 | 176.152202 |
| **75%** | 69.586985 | 198.285979 |
| **max** | 91.566155 | 253.611685 |

Let's look at what that did. Firstly, it returned a data frame, but not one of the same size or shape that we gave it. Instead it iterated over the columns and created these standard statistical measures for each column possible. We say each column possible because one is missing: Gender. That's because gender is a string, rather than a numeric value. We can't compute the means of strings.

Now, as for the values themselves. Count should be relatively self evident, as should min and max. Mean and std (standard deviation) we've also talked about before. The three percent values are percentiles. These values represent cutoff points, below which a certain percentage of the data lies. So, 25% of weights are below 162.82 and so on.

Together, these values give us a decent image of what each of the variables included looks like. We can get a numerical sense of what we might call their "shape". However, this is only one part of .describe()'s capabilities. As we covered in the toolkit unit, we can also group our data. This allows us to be even more insightful with our describe, letting us compare the summary statistics for two different groups of our data.

In [7]:



1

data.groupby('gender').describe()

Out[7]:

|  |  | **height** | **weight** |
| --- | --- | --- | --- |
| **gender** |  |  |  |
| **female** | **count** | 100.000000 | 100.000000 |
| **mean** | 63.195078 | 164.643079 |
| **std** | 4.786304 | 12.775653 |
| **min** | 50.591490 | 128.763644 |
| **25%** | 59.987204 | 153.933783 |
| **50%** | 63.674985 | 166.122162 |
| **75%** | 66.576880 | 173.270107 |
| **max** | 74.507797 | 192.725068 |
| **male** | **count** | 100.000000 | 100.000000 |
| **mean** | 68.647143 | 198.189371 |
| **std** | 8.008156 | 25.038595 |
| **min** | 49.268448 | 126.352513 |
| **25%** | 62.886700 | 184.832311 |
| **50%** | 68.617447 | 198.401302 |
| **75%** | 74.520715 | 217.906158 |
| **max** | 91.566155 | 253.611685 |

Now we have twice the output. This may not be the easiest form to read it, but it does give us a sense of the difference between the two groups, male and female. In this case we can see that the distributions for height and weight are higher for men than for women, which is what we'd expect. This kind of grouping can give us another layer of insight to our analysis.

## Value Counts

Sometimes, you aren't dealing with data that is best summarized in this form. The most common example of this is strings, where these kinds of methods do not apply. In that case what you're probably interested in is counts. Python gives you an easy way to go over a column of data and return the distinct values as well as the counts of each.

In [8]:



1

data.gender.value\_counts()

Out[8]:

female 100

male 100

Name: gender, dtype: int64

Now, the first thing to note is that this method is working on data.gender, which is a series object rather than a data frame object. This .value\_counts() method cannot iterate over a whole data frame. Luckily each column and row in a data frame is a series and you can use this method simply by selecting a column as we did above.

There are several reasons to use this method. Firstly, it gives you another way to make sense of your data. In this case it shows us that our data is evenly balanced between males and females, with one hundred samples of each.

There are plenty of other ways this function could be useful. It can show outliers or possible malformed data. For example, if we were to see something like 'Mal' with a single entry, we'd have found a typo in the data. This method works over both numerical and object data, though it is not valuable to run over the numeric columns in this example. Can you think of why?

In [9]:



1

data.weight.value\_counts().head()

Out[9]:

175.143849 1

208.507849 1

169.185494 1

165.450584 1

165.087602 1

Name: weight, dtype: int64

As you can see, it's not useful because we're dealing with truly continuous random data, so no value is exactly repeated. We simply get a list of all the values with a count of 1 for each.

However, these two methods, .describe() and .value\_counts(), do often provide incredibly easy and valuable insights into your dataset. You'll want to use them throughout the course as one of the ways to get a first, quick sense of the data before digging in more specifically on points of interest.

# Drill - Describing Data

 Estimated Time: **1-2 hours**

Now that we have introduced some tools for describing populations, let's try them out. First do these drills by hand, then use the Python code we've provided in the previous assignments to check your work. Keep track of your work in a Google document or markdown file that you can submit below and share with your mentor.

1. Greg was 14, Marcia was 12, Peter was 11, Jan was 10, Bobby was 8, and Cindy was 6 when they started playing the Brady kids on The Brady Bunch. Cousin Oliver was 8 years old when he joined the show. What are the mean, median, and mode of the kids' ages when they first appeared on the show? What are the variance, standard deviation, and standard error?
2. Using these estimates, if you had to choose only one estimate of central tendency and one estimate of variance to describe the data, which would you pick and why?
3. Next, Cindy has a birthday. Update your estimates- what changed, and what didn't?
4. Nobody likes Cousin Oliver. Maybe the network should have used an even younger actor. Replace Cousin Oliver with 1-year-old Jessica, then recalculate again. Does this change your choice of central tendency or variance estimation methods?
5. On the 50th anniversary of The Brady Bunch, four different magazines asked their readers whether they were fans of the show. The answers were: TV Guide 20% fans Entertainment Weekly 23% fans Pop Culture Today 17% fans SciPhi Phanatic 5% fans

Based on these numbers, what percentage of adult Americans would you estimate were Brady Bunch fans on the 50th anniversary of the show?

Discuss your answer to each of these questions, along with your code, with your mentor.

When you've given it a try, you can find a solution [here](https://github.com/Thinkful-Ed/data-201-resources/blob/master/solutions/Prep%20course/3.1.4.ipynb).

# Wrap Up

 Estimated Time: **5 minutes**

In this lesson, you were introduced to the basics of statistics and sampling.

Moving forward, you should feel comfortable with the following:

* Explaining the relationship between a population and a sample
* Calculating and interpreting the mean, median, and mode of a sample
* Calculating and interpreting the variance, standard deviation, and standard error of a sample

If you're unclear on any of the above, be sure to follow up with your mentor.

# Wrap Up

 Estimated Time: **5 minutes**

In this lesson, you were introduced to the basics of statistics and sampling.

Moving forward, you should feel comfortable with the following:

* Explaining the relationship between a population and a sample
* Calculating and interpreting the mean, median, and mode of a sample
* Calculating and interpreting the variance, standard deviation, and standard error of a sample

If you're unclear on any of the above, be sure to follow up with your mentor.

# Perspectives on Probability

 Estimated Time: **15 minutes**

Probability is a way of quantifying the likelihood of a future outcome. People use probabilities to make decisions all the time. Knowing the probability of rain tomorrow helps to decide whether to put the rainboots and umbrella in the car tonight.

There are two broad schools of thought about probability in statistics. The frequentist school of thought defines probability as describing how often a particular outcome would occur in an experiment if that experiment were repeated over and over. For example, each time a coin is flipped, the outcome is either 1 (heads) or 0 (tails). Each coin flip is an "experiment" with an outcome. Over many coin flips, the coin will come up heads about 50% of the time. For a frequentist, saying an outcome has a 50% probability is equivalent to saying that if the experiment were repeated many times, that outcome would occur 50% of the time.

The Bayesian school of thought defines probability as describing how likely an observer expects a particular outcome to be in the future, based on previous experience and expert knowledge. Each time the experiment is run, the probability is updated if the new data changes the belief about the likelihood of the outcome. The probability based on previous experiences is called the "prior probability," or the "prior," while the updated probability based on the newest experiment is called the "posterior probability."

Both Bayesian and frequentist approaches to probability are used to model data, depending on the question being asked. Disagreements between the two camps can [get](https://xkcd.com/1132/) [heated](http://www.smbc-comics.com/index.php?id=4127), but there's no need to take sides: Most of the time, the Bayesian and frequentist approaches arrive at the same answer. A good rule of thumb is that frequentists are trying to calculate the likelihood of getting the data you have in the context of a fixed, if unknown, "true" population value. Bayesians are trying to calculate the most likely population value, given the data you have. As the data changes, Bayesians beliefs about the population value change as well.

For a more in-depth discussion with complex examples of how frequentist and Bayesian approaches can be used to answer a question, including code in Python, see this [series of articles by Jake Vanderplas](http://jakevdp.github.io/blog/2014/03/11/frequentism-and-bayesianism-a-practical-intro/)

# Randomness, Sampling and Selection Bias

 Estimated Time: **1 hour**

The concept of probability is fundamental to data science. A lot of the value a data scientist brings comes from the ability to quantify uncertainty – to go from a vague and woolly "maybe it will rain" to a clear and precise "65% chance of rain with a margin of error of 3%." Quantified uncertainty not only defines what is known, but how confident we can be about that knowledge.

Probability is also the fundamental basis of another critical element of the data scientist's toolkit: Randomness. Randomness can be defined as a situation where all possible outcomes are equally likely. The flip of a balanced coin is random: 50% likely to be heads, 50% likely to be tails. The roll of a six-sided dice is also random, with a 1 in 6 chance of getting each number between 1 and 6.

As we mentioned in an earlier assignment, it is rarely possible to gather data from all elements of a population of interest (can't get questionnaires to all potential customers around the world, can't get information about purchasing decisions that haven't happened yet). Instead, data scientists leverage the concept of randomness to gather a representative sample from the population. Using randomness, each element of a population has an equal chance of being chosen for the sample.

In an ideal world, **random sampling** creates a smaller group (a sample) that is sufficiently similar to the population that anything we learn about the sample also applies to the population. Larger samples are more likely to resemble the population, because they are less swayed by the occasional extreme value. On the other hand, smaller samples may be cheaper and faster to collect. Deciding how large a sample to collect depends in part on how variable we think the population is: the more variability in the population, the larger sample we need to get to be confident that our sample is a good stand-in for the population. However, it is important to keep in mind that the sample is random: It may, through sheer dumb luck, sample enough unusual individuals to be unrepresentative.

In practice, random sampling depends on perfect access to the population, which is very rarely possible. When studying customers, for example, not all customers may be interested in or willing to participate in data collection. The sample, in that case, differs from the population of all customers: The customers in the sample are all people who are willing to be studied. Systematic differences between the sample and the population are known as selection bias. If the sampled individuals differ from the others in important ways, such as spending rates or purchasing behavior, then knowledge gained from their data might not apply to all customers.

## Selection bias: A practical example

An example from US history illustrates the pitfalls of generalizing from a biased sample. The 1936 US Presidential election pitted Alfred Landon against Franklin D. Roosevelt. The largest pre-election poll of the time, conducted by the respected magazine Literary Digest, projected that Landon would beat Roosevelt by 14%. This projection was based on ballots sent to 10 million US citizens, nearly a quarter of all eligible voters at the time. Yet the Literary Digest turned out to be impressively wrong, with Roosevelt beating Landon by 24%. The poll was off by an astounding 38%. How did this happen?

The Literary Digest's sampling efforts, though broad, were flawed: they sampled people by drawing from the telephone directory. In the middle of the Great Depression, phones were luxuries many could not afford. The Digest's methods led to a sample that was considerably older and richer than the general population… with predictable results for their election forecast. When it comes to sampling, it is better to have a small but representative sample than a large and biased one.

For more details on the Literary Digest poll, see [this case study](https://www.math.upenn.edu/~deturck/m170/wk4/lecture/case1.html).

# Independence and Dependence

 Estimated Time: **1 hour**

When talking about probabilities of events, whether Bayesian or frequentist, one important consideration is whether the events are independent or dependent of one another. An event is independent of other events in the sample space if the outcome of that event is not affected by the outcome of other events in the sample space. For example, imagine a bag of 10 marbles, containing 5 blue marbles and 5 red marbles. Without looking, you reach into the bag and draw out a red marble. Then you put the marble back in the bag, and draw a blue marble. The probability of drawing a red marble first is 5/10, or 50%. The probability that the second marble will be blue is also 5/10, or 50%. Because you put the first marble back, the probability of drawing the second marble is independent of the first. Neither event affects the other.

The probability of drawing and replacing a red marble:

**P**(red) = 1/2

P in the formula above means "probability".

The probability of drawing and replacing a blue marble after drawing and replacing the red marble):

**P**(red) = **P**(blue) = 1/2

The probability of two or more independent events can be calculated by multiplying the probabilities of each individual event. The probability of drawing a red marble and then a blue marble:

P(red ∩ blue) = P(red) \* P(blue) = (1/2) \* (1/2) = 1/4

When the probability of event B changes based on the outcome of event A, the probability of event B is said to be dependent on, or conditional on, event A. Returning to the bag of marbles, again you reach into the bag and take out a red marble. Again, the probability that the first marble will be red is 5/10 or 50%.

**P**(red) = 1/2

Next, without replacing the red marble, you draw a blue marble. Now, the probability of drawing a blue marble depends on the color of the first marble drawn. We use | to denote a condition on the probability we're talking about. This is where the information we already have can be disclosed.

If the first marble was blue, then the probability of drawing a red marble the second time is 5/9 (because one blue marble is missing from the bag).

**P**(red | blue) = 5/9

You can read the | symbol as "given", so P(red | blue) would read "the probability of red given blue".

If the first marble was blue, then the probability of drawing a blue marble the second time is 4/9 (because one blue marble is missing from the bag).

**P**(blue | blue) = 4/9

The probability of drawing a blue marble the second time is conditional on the outcome of the first draw.

The probability of two conditional events can be calculated by multiplying the probability of event A by the probability of event B conditioned on A.

P(A ∩ B) = P(A) \* P(B | A)

The probability of drawing two blue marbles in a row without replacement:

P(blue) \* P(blue | blue) = (1/2) \* (4/9) = 2/9

The probability of drawing a red marble and then a blue marble without replacement:

P(red) \* P(blue | red) = (1/2) \* (5/9) = 5/18

In data, conditional variables (conditional and dependent are synonymous, however, because a 'dependent variable' has a specific meaning in experimental design, we will use conditional when referring to variables) in a dataset contribute less information than independent variables, because some information is duplicated among conditional variables. For example, a survey may ask four questions: 1) a customer's age, 2) their income, 3) whether they bought any widgets that month, and 4) how much money they spent on widgets that month. The age and income variables are independent in the sample space: While knowing someone's age may give some hints about their income, there is enough variability in incomes between people of the same age that every datapoint is giving new information. Similarly, while the income variable is probably related to how much money people have available to spend on widgets, people with the same income may buy different amounts of widgets (or no widgets at all), so each datapoint provides new information for both variables.  
Questions three and four, on the other hand, are conditional. If someone says that they didn't buy any widgets in the last month, we already know they spent $0 on widgets. Conversely, if someone says they spent $0 on widgets last month, we already know they didn't buy any widgets. The two variables aren't perfect duplicates, since knowing someone did buy one or more widgets isn't the same as knowing how much money they spent, but for at least some cases, knowing the answer to question 3 means we can be 100% certain that we know the answer to question 4 (and vice versa).

# DRILL - Exercises in Probability

 Estimated Time: **1-2 hours**

Now it’s time to compute some probabilities. Keep track of your work in a Google document or markdown file that you can share with your mentor.

1. Calculate the probability of flipping a balanced coin four times and getting each pattern: HTTH, HHHH and TTHH.
2. If a list of people has 24 women and 21 men, then the probability of choosing a man from the list is 21/45. What is the probability of not choosing a man?
3. The probability that Bernice will travel by plane sometime in the next year is 10%. The probability of a plane crash at any time is .005%. What is the probability that Bernice will be in a plane crash sometime in the next year?
4. A data scientist wants to study the behavior of users on the company website. Each time a user clicks on a link on the website, there is a 5% chance that the user will be asked to complete a short survey about their behavior on the website. The data scientist uses the survey data to conclude that, on average, users spend 15 minutes surfing the company website before moving on to other things. What is wrong with this conclusion?

Discuss your answer to each of these questions with your mentor.

# Bayes' Rule

 Estimated Time: **1 hour**

# Bayes' Rule

On a random day you see a pop up clinic for an instant test for a new disease you’ve heard of: Weapon X. It’s an incredibly rare disease with almost no symptoms for months and then you suddenly die. It’s affecting about one in a million people, from what you’ve heard. This test says it’s 99.99% accurate in both directions (false positives and false negatives), so you decide it’s worth taking the test.

It comes back positive.

Should you panic?

Bayes' Rule explains that, in actuality, you’re probably just fine.

## Conditional Probability

In this scenario we’re not focused on the probability of an independent event. It’s not the probability that you are infected with Weapon X. It’s the probability that you’re infected given the condition that you get a positive test.

That positive test provides additional information about what’s going on, and we can use that information to improve our probability estimate.

For this example, let’s say we have a million people. Chances are one of them is infected, since the disease affects one in a million, and that person will likely get a positive test result. However, if we tested every one of those other 999,999 people, we’d expect about 100 people to get positive results.

Once you see that positive test, what do we know about your likelihood of being infected? Since we know you have a positive test, we can consider only the people in that group who have seen a positive test result. In that group are 101 people, only 1 of whom actually has the infection. This works out to roughly 1% chance that you’re infected.

Now, these counts are a bit of a simplification, using expected counts rather than probabilities. Bayes' rule gives you this relationship in terms of probabilities.

## Bayes' Rule

Bayes' Rule can be put in its simplest and most abstract terms as follows:

P*(A | B)* = P*(B | A)* \* P*(A)* / P*(B)*

**OR**

P*(A | B)* = P*(B | A)* \* P*(A)* / [P*(A)*\*P*(B|A)* + P*(A~)*\*P*(B|A~)*]

In English, this formula says the probability of A given B equals the probability of B given A, times the probability of A, divided by the probability of B. We expand the probability of B in the second formula where A~ is the inverse of A, so in our case not being infected. The numerator is our scenario of interest, while the denominator represents the realm of scenarios that could give our condition.

We can put that in terms of our example pretty simply.

P(Infected| Positive Test) = P(Positive Test| Infected) \* P(Infected) / P(Positive Test)

= .9999 \* .000001/(.000001\*.9999 + .999999\*.0001) = .0099 or .99%

So that says it’s less than 1% that you’re actually infected, which is still reasonably unlikely.

This may seem like a detached example, but this really happens. In general, people can be very bad at this kind of probabilistic reasoning. In one example in New York and San Francisco, [clinics stopped using a rapid HIV test because it was scaring healthy people](http://www.nytimes.com/2005/12/10/health/false-positives-from-hiv-test.html?_r=0). It’s why a lot of these tests get run multiple times before their results are given. The first response to a test suggesting an individual has a rare disease is usually to run it again.

# Drill - Exercises in Bayes' Rule

 Estimated Time: **1 hour**

Now it's time to use Bayes' rule to compute some conditional probabilities. First look over the numbers and estimate each of the four probabilities, using your intuition. Then, calculate the probabilities using Bayes' rule. Keep track of your work in a Google document or markdown file that you can share with your mentor.

A diagnostic test has a 98% probability of giving a positive result when applied to a person suffering from Thripshaw's Disease, and 10% probability of giving a (false) positive when applied to a non-sufferer. It is estimated that 0.5 % of the population are sufferers. Suppose that the test is now administered to a person whose disease status is unknown. Calculate the probability that the test will:

1. Be positive
2. Correctly diagnose a sufferer of Thripshaw's
3. Correctly identify a non-sufferer of Thripshaw's
4. Misclassify the person

Were your intuitions on the mark, or way off? If your statistical intuition is leading you astray, you aren't alone. According to Nobel-prize winning Daniel Kahneman, [humans simply are not good intuitive statisticians](http://www.burns-stat.com/review-thinking-fast-slow-daniel-kahneman/). That fact has two strong implications for you as you prepare for a career in data science:

1. Just because your statistical intuition is wrong does not mean you're a bad statistician. We're all in this boat. Don't give up on yourself. And,
2. Just because you easily intuit the answer to a statistical question does not mean you're right. Don't trust your intuition. Check your work.

Discuss your answer to each of these questions with your mentor. If you want more Bayes practice, try [these exercises](http://stony.me/statistics-for-beginners-bayes-rule-4/).

After giving it a try yourself, you can find a solution [here](https://github.com/Thinkful-Ed/data-201-resources/blob/master/solutions/Prep%20course/3.2.6.md).

# Evaluating data sources

 Estimated Time: **1 hour**

Not all data sources are created equal. It is tempting to look at a dataset and assume it is correct, or representative, or that because it's about the subject you're interested in it can answer the questions you have about the topic. That is not necessarily the case. Evaluating data both for quality and bias is an important skill of data science.

## Bias

We've mentioned the notion of samples versus populations before. It is very common to look at a problem and not be able to get data on every relevant individual, so we take a sample. Whenever we're dealing with a sample, it is essential to ask if that sample is truly random. If you do the sampling yourself, that process can be pretty easy. You know the process you engaged in to sample from the population and you can thoroughly evaluate its credibility.

For example, if you wanted information about the popular kinds of cereal, you could go to your local grocery and make a list of the cereals that they have on the shelf. This will give you a dataset, but not a great one. It's a single sample, biased by things like geographic location. The same principle applies when scraping a website, it's a single sample with the biases associated with that website.

This process becomes much more difficult, however, when dealing with data collected by someone else, and particularly datasets available on the internet. Pay close attention to the method used to collect the data and evaluate what if any biases may be present because of that.

## Quality

It's important when approaching a new data source to try to judge its objective quality. Are there are a lot of unexplained blanks? Is it unevenly distributed across time? Does the volume fluctuate seemingly without cause? All of these things, and many more, can be signals that the quality of the dataset is questionable. Engaging in this questioning early and explicitly will help prevent situations where the conclusions of your analysis are compromised or when poor data quality becomes your conclusion.

## Exceptional Circumstance

What was the situation under which data was collected? This can greatly impact the data and therefore the quality of conclusions you can draw from it. Returning to the grocery example, if you were trying to analyze the eating habits of Americans, you could look at the shopping behaviors at grocery stores across the country. Let's say you obtained a large sample that accurately covered the whole of the country with data from a variety of cities and neighborhoods.

But what if that data was collected on the day before the Super Bowl? Your dataset would then be skewed because the data was collected in an exceptional circumstance. The shopping behaviors on that day are possibly quite different from normal. Thinking about that aspect of the dataset can again help catch its limitations early on in the process.

## What to do?

So what can you do with data that has issues with its quality or source? There are many options.

Firstly if you can quantify the bias, you can **adjust** to it. For example, let's say you were interested in fishing numbers, but only had data from San Francisco for the given year. You can look at other databases and figure out how other cities typically compare to where you have your data and try to impute larger trends. We'll cover imputing data in more detail later in the bootcamp.

You can also **limit your conclusions** to the scope of the data. Be very clear about where your conclusions are applicable. If you scraped a single online retailer, understand and state that your conclusions only apply to that store. When using data from Amazon, you're talking about Amazon shopping behaviors, not retail consumption as a whole. That analysis can still be highly valuable.

## Drill

In each of the scenarios, find possible shortcomings of the theoretical or actual data sources to answer the given question. What could be done to either adjust the analysis or reframe the question so that you can answer it accurately?

* **Data Source**: Amsterdam availability data scraped from AirBnB on December 24th. **Question**: What are the popular neighborhoods in Amsterdam?
* **Data Source**: Mental health services use on September 12, 2001 in San Francisco, CA and New York City, NY. Question: How do patterns of mental health service use vary between cities?
* **Data Source**: [Armenian Pub Survey](https://www.kaggle.com/erikhambardzumyan/pubs). **Question**: What are the most common reasons Armenians visit local pubs?

Write up your answers and submit a link below.

After trying it yourself, you can find a solution [here](https://github.com/Thinkful-Ed/data-201-resources/blob/master/solutions/Prep%20course/3.2.7.md).

# Challenge - Beware of Monty Hall

 Estimated Time: **1 hour**

One of the classic problems in this space is referred to as the Monty Hall Problem. Some people even use this as an interview question! It is deceptively simple, and really digging into it reveals a myriad of approaches and some serious applications of conditional probability. The story goes like this:

You are on a game show and given the choice of whatever is behind three doors. Behind one door is a fantastic prize (some examples use a car, others use cash) while behind the other two doors is a dud (some examples say a goat, others say it's just empty). You pick a door. Then the host opens one of the other two doors to reveal a dud. But here's the wrinkle: the host now gives you the opportunity to switch your door. What should you do?

Write up some notes on this problem, including how you think Bayes' Rule might apply. Drop a link to your notes below and discuss it with your mentor.

After that check out the Wikipedia page for this problem. [It's quite thorough](https://en.wikipedia.org/wiki/Monty_Hall_problem).

# Wrap Up

 Estimated Time: **5 minutes**

In this lesson, you were introduced to basic concepts of probability as they are used by data scientists.

Moving forward, you should feel comfortable with the following:

* Defining probability
* Explaining the role of randomness in sampling
* Differentiating between independent and conditional events, and calculating the probabilities of each
* Applying Bayes’ Rule

If you're unclear on any of the above, be sure to follow up with your mentor.

# The Normal Distribution and the Central Limit Theorem

 Estimated Time: **8-9 hours**

A whole host of commonly-used statistical methods are built on the assumption that data is "normal". In this lesson, we will discuss what normality is, what it does, and how to tell if the data you currently are working with are normal using histograms in Python.

# Define Normality

# Thinkful Data Science Prep course 3.3.1

## Define Normality

Data is described as "normal," or "normally distributed," if most values cluster in the center of the range, with the rest tapering off symmetrically to the left and the right. The mean and median of a normally distributed variable are equal. The information in a normal distribution can be summarized by the mean μ ("mu") and standard deviation σ ("sigma"). The probability density function for a normally distributed variable is:

f(x|μ,σ2)=12σ2π⎯⎯⎯⎯⎯⎯⎯⎯√e−(x−μ)22σ2f(x|μ,σ2)=12σ2πe−(x−μ)22σ2

e is [Euler’s number](http://mathforum.org/dr.math/faq/faq.e.html) (e=2.71828…), a mathematical constant.

While you don’t need to memorize the probability density function to work with normally distributed variables, it is good to be able to recognize it if you come across it while reading about other statistical concepts.

Approximately 68% of the values in a normally-distributed variable fall within 1 standard deviation above or below the mean, 95% of values fall within two standard deviations of the mean, and 99.7% of values fall within three standard deviations of the mean.

We can use Python to generate a normally distributed variable by providing a mean and standard deviation, which we graph as a histogram.

In [1]:



1

**import** numpy **as** np

2

**import** pandas **as** pd

3

**import** matplotlib.pyplot **as** plt

4

**%**matplotlib inline

In [2]:



1

*# Making a standard normally distributed variable with 1000 observations, a mean of 0, and*

2

*# a standard deviation of 1, and putting it in a data frame.*

3

mean = 0

4

sd = 1

5

n = 1000

6

​

7

df = pd.DataFrame({'rand': np.random.normal(mean, sd, 1000)})

8

​

9

*# Plotting the variables in the data frame (here, just the variable "rand") as a histogram.*

10

df.hist()

11

*# Inline printing the histogram*

12

plt.show()

The normal distribution is useful for data scientists because:

* It is easily summarized using just two statistics (mean and standard deviation).
* The area under the curve is 1, making it easy to calculate the probability of individual outcomes within the distribution.
* It describes many natural phenomena, such as blood pressure, height, weight, etc.
* In general, any variable that measures an outcome produced by many small effects acting additively and independently will have a close to normal distribution.
* Lots of common scores (percentiles, z-scores) and statistical tests (t-tests, ANOVAs, bell-curve grading) assume a normal distribution.

# WDIB - Deviations from Normality and Descriptive Statistics

# Thinkful Data Science Prep course 3.3.2

## When does it break: deviations from normality and descriptive statistics

Unfortunately, the usefulness of the normal distribution means that it oftentimes becomes the "default" distribution in people’s minds. This isn’t helped by the fact that it is called "normal"! Real data (as opposed to idealized mathematical concepts) is never perfectly normal, but some data is more normal than others. When statistics that assume normality are used on non-normal data, the mis-match between statistics and data can result in inaccurate conclusions.

While there are statistical tests of non-normality, they are sensitive to sample size, meaning that whether or not the test says your data is normal has more to do with how much data you have than the distribution of your data. Instead, the best method of deciding if your data is normal is to inspect the data visually using histograms and quantile-quantile (QQ) plots.

QQ plots graph a variable with an unknown distribution against a variable with a known distribution. Values for each variable are sorted into ascending order, then plotted against each other with the known variable as the x-axis and the unknown variable as the y-axis. If the mystery variable shares the same distribution as the known variable, the result should be a straight line running from the lower left-hand corner to the upper right-hand corner of the plot. Deviations from the straight line indicate that the data does not fit the distribution.

Let’s try a QQ plot to check if data is normally distributed:

In [2]:



1

**import** numpy **as** np

2

**import** pandas **as** pd

3

**import** matplotlib.pyplot **as** plt

4

**%**matplotlib inline

In [3]:



1

*# Making two variables.*

2

rand1 = np.random.normal(50, 300, 1000)

3

rand2 = np.random.poisson(1, 1000)

4

​

5

*# Sorting the values in ascending order.*

6

rand1.sort()

7

rand2.sort()

8

​

9

*# Making a standard normally distributed variable with 1000 observations,*

10

*# a mean of 0, and standard deviation of 1 that we will use as our “comparison.”*

11

norm = np.random.normal(0, 1, 1000)

12

​

13

*# Sorting the values in ascending order.*

14

norm.sort()

In [4]:



1

*# Plotting the variable rand1 against norm in qqplots.*

2

plt.plot(norm, rand1, "o")

3

plt.show()

Looking at the QQ plot, it is clear that the values of "rand1" are normally distributed, while the values of "rand2" are not normally distributed. (In fact, "rand2" reflects a different probability distribution, "Poisson," which will be discussed in a later assignment.)

You may notice that with a QQ plot, the scales of the known and unknown variables do not have to match: What matters is the relationships between datapoints within each variable.

When data are not normal, the mean and standard deviation are no longer accurate or informative summaries. Let's make histograms of rand1 and rand2, then compute descriptive statistics to see how well they match up.

# Other Distributions

# Thinkful Data Science Prep course 3.3.3

## Other distributions

So far, we’ve categorized data as either “normal” or “non-normal,” but there are many other probability distributions that also have useful characteristics for addressing particular statistical problems. We won’t review all of them (see here for a [more comprehensive list](https://www.causascientia.org/math_stat/Dists/Compendium.pdf)) but here are brief introductions to some of the most common.

### Bernoulli

The **Bernoulli distribution** represents two possible outcomes of an event (such as a coin flip). Summarized by p, the probability of the outcome k.

The probability mass function for the Bernoulli distribution is:

f(k|p)={p,1−p,if k=1if k=0f(k|p)={p,if k=11−p,if k=0

Note that when a distribution is discrete (only takes integers), it has a probability mass function, while a continuous distribution has a probability density function.

In [1]:



1

**import** numpy **as** np

2

**import** pandas **as** pd

3

**import** matplotlib.pyplot **as** plt

4

**%**matplotlib inline

In [2]:



1

*# Generate a bernoulli distribution with p =0.5.*

2

bernoulli= np.random.binomial(1, .5, 1000)

3

​

4

*#Plot a histogram.*

5

plt.hist(bernoulli)

6

​

7

*# Print the histogram*

8

plt.show()

**Binomial:**

A **binomial distribution** counts the number of successes when an event with two possible outcomes is repeated many times (such as many coin flips). Summarized by *p*, the probability of getting *k* successes during *n* repetitions of the event. The probability mass function is:

f(k|n,p)=(nk)pk(1−p)(n−k)f(k|n,p)=(nk)pk(1−p)(n−k)

**Gamma**

The **gamma distribution** represents the time until an event (such as lifespan until death), when the event starts out unlikely (few people die in youth), becomes more likely (more people die in old age), then becomes less likely again (few people die in extreme old age because most have already died). Summarized by a shape parameter (αα) and an inverse-scale parameter (ββ). The probability density function is:

f(x|α,β)=βαxα−1e−xβΓ(α)for x≥0 and α,β≥0

**Poisson**

The **poisson distribution** represents the number of times a given event (such as a phone call to a radio show) will occur during a given time interval. Data can range from 0 (no phone calls during the time period) to approaching infinity (the phone never stopped ringing during the time period). Summarized by λλ (“lambda”), the rate that events occur during a given time period. The probability mass function is:

f(k|λ)=λke−λk!f(k|λ)=λke−λk!

## Conditional Distribution

Distributions can also be conditional. Consider an ecommerce site. For all of the customers, we have a distribution of the amount that they have spent on the website. It may look something like this:

Creating a data frame to hold the simulated ecommerce data, and populating it with a

# normally distributed variable with mean 75 and standard deviation 25.

ecommerce = pd.DataFrame()

ecommerce['spending'] = np.random.normal(75, 25, 1000)

# Plot a histogram.

plt.hist(ecommerce['spending'])

plt.show()

<https://xkcd.com/795/>

# DRILL - Descriptive Statistics and Normality

To complete the following drills, you'll need to use your Python skills to create some datasets, then use your new statistical knowledge to summarize them. Choose 6 distributions from the list of random distributions available in NumPy, called “[Distributions”](https://docs.scipy.org/doc/numpy/reference/routines.random.html#distributions)

For each distribution:

1. Generate a random variable with 100 datapoints using the code distributionvar = np.random.distributionname([arguments], 100), replacing distributionvar with an appropriate variable name and distributionname with the name of the distribution you’ve chosen, and filling in the empty space in the parentheses with your chosen values for the appropriate parameters. If you feel uncertain about how to do this, go back to the “Other Distributions” assignment for examples of code to use as a starting point.
2. Graph the variable using a histogram.
3. Compute the mean and standard deviation and plot them as vertical lines on the histogram. (Hint: the “When Does It Break?” assignment you just completed can help you here.)
4. Evaluate whether the descriptive statistics provided useful information about the variable. Can you identify any common characteristics of the distributions that could be usefully described using the mean and/or standard deviation, versus the ones that could not?

Additionally:

1. Generate two normally-distributed variables, one with a mean of 5 and standard deviation of 0.5, and the other with a mean of 10 and standard deviation of 1.
2. Add them together to create a third variable.
3. Graph the third variable using a histogram.
4. Compute the mean and standard deviation and plot them as vertical lines on the histogram.
5. Evaluate the descriptive statistics against the data.

# CLT and Sampling

 Estimated Time: **15 minutes**

We have been discussing the power and convenience of the normal distribution for descriptive statistics. However, while many aspects of nature are normally distributed, many others are not. Fortunately, the [central limit theorem](https://en.wikipedia.org/wiki/Central_limit_theorem) shows that normal distributions can still be used to analyze samples of data from non-normally distributed populations.

When sampling from a population (of any distribution), as the sample size gets larger the sample means tend to follow a normal probability distribution, clustering around the true population mean. The more non-normal the population the larger the samples need to be, but ultimately this means that statistics can be calculated, and population parameters estimated, even when the distribution of a population is unknown.

Given that statistics are generally used to discover information about populations, data scientists need to be able to start doing statistics on a population without knowing much about it. The central limit theorem demonstrates that this is possible, and by doing so, unlocks the door to all the statistics we have discussed so far (mean, median, variance, standard deviation, standard error) and many of the statistics we will be discussing from this point forward.

# [Report a typo or other issue](javascript:void(0);) Central Limit Theorem in Action

 Estimated Time: **1 hour**

## Comparing Groups in a Sample: The Central Limit Theorem in Action

For an example of the central limit theorem in action, we can compare the means of two samples, drawn from two populations. To compare two means, we want to calculate the mean and standard deviation of each sample. Then, we subtract one mean from the other, and look at the size of the difference in the context of the combined variance of the two samples. The larger the difference is relative to the variance, the less likely the difference is to be due to random chance, and the more likely it is to reflect meaningful differences between the two populations.

y¯=x¯2−x¯1y¯=x¯2−x¯1

To combine the variances, the squared standard deviations are divided by the sample size and summed, then we take the square root of the sum. This is the standard error of the difference, representing the variance in the sample differences around the population difference, and you can see the formula written below.

se=s21n1+s22n2⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯√se=s12n1+s22n2

The difference divided by the standard error is a ratio called the **t-value**. T-values put the difference in context by telling us how large the difference is relative to the amount of variance, or noise, in the data. In very noisy data, small differences are more likely to result from the noise rather than from real differences between the population means.

t−value=y¯set−value=y¯se

Given a t-value, we can calculate the probability that a t-value at least this extreme would occur by chance, called a **p-value**. This tells us how likely it is that we would get the sampling data we observe if the two population means were not, in fact, different from one another. The p-value runs between 0 (It is impossible to get a difference of this size or greater in the absence of a real population difference) and 1 (we will always get a difference of this size or greater in the absence of a real population difference). The lower the p-value, the more confidently we can conclude that there is a meaningful difference between the means of the two groups in the population.

Let’s spin up some population data and give this a try. We’ll make two variables to represent two different populations: one a binomially distributed variable with p of 0.2, n=10, and 10000 datapoints (group1), and another binomially distributed variable with p of 0.5, n=10, and 10000 datapoints (group2). The true population difference between the two populations is 0.3.

In [1]:



1

**import** numpy **as** np

2

**import** pandas **as** pd

3

**import** scipy

4

**import** matplotlib.pyplot **as** plt

5

**%**matplotlib inline

In [2]:



1

pop1 = np.random.binomial(10, 0.2, 10000)

2

pop2 = np.random.binomial(10,0.5, 10000)

3

​

4

*# Let’s make histograms for the two groups.*

5

​

6

plt.hist(pop1, alpha=0.5, label='Population 1')

7

plt.hist(pop2, alpha=0.5, label='Population 2')

8

plt.legend(loc='upper right')

9

plt.show()

The populations are not normal. Next, take a sample of 100 from each population and plot them.

In [3]:



1

sample1 = np.random.choice(pop1, 100, replace=**True**)

2

sample2 = np.random.choice(pop2, 100, replace=**True**)

3

​

4

plt.hist(sample1, alpha=0.5, label='sample 1')

5

plt.hist(sample2, alpha=0.5, label='sample 2')

6

plt.legend(loc='upper right')

7

plt.show()

Next, compute the means and standard deviations for each group. Note that the mean represents n \* p: the probability of an event occurring (p) multiplied by the number of repetitions (n). To get p for each sample, divide by n, which we set to 10 when generating the populations.

In [4]:



1

print(sample1.mean())

2

print(sample2.mean())

3

print(sample1.std())

4

print(sample2.std())

5

​

6

*# Compute the difference between the two sample means.*

7

diff=sample2.mean( ) **-**sample1.mean()

8

print(diff)

1.85

5.02

1.32193040664

1.62468458477

3.17

Next, calculate the standard error of the sampling distribution of the difference of the means. First, create an array with the size of each variable and another with the standard deviation of each variable. In this case, the sizes are already known since you provided them earlier, but let’s calculate them anyway.

In [5]:



1

size = np.array([len(sample1), len(sample2)])

2

sd = np.array([sample1.std(), sample2.std()])

3

​

4

*# The squared standard deviations are divided by the sample size and summed, then we take*

5

*# the square root of the sum.*

6

diff\_se = (sum(sd **\*\*** 2 **/** size)) **\*\*** 0.5

7

​

8

*#The difference between the means divided by the standard error: T-value.*

9

print(diff**/**diff\_se)

15.1345842493

Finally, we import the function ttest\_ind from scipy.stats, which calculates the t-value for us (called “statistic”) and also provides the probability calculation (called “pvalue”). The t-value we calculated and the t-value given by the function may differ slightly after the hundredth decimal place. This is due to differences of rounding caused by our multiple-step approach to calculating the t-value.

In [6]:



1

**from** scipy.stats **import** ttest\_ind

2

print(ttest\_ind(sample2, sample1, equal\_var=**False**))

Ttest\_indResult(statistic=15.058721193921539, pvalue=2.9983456846787284e-34)

The t-value scales the difference between the two groups by the amount of variance in the two samples. High variability in samples can lead to groups with means that look very different, but when we look at the histogram we see that most of the values in the sample groups overlap. The groups are so variable that the distribution of values is quite broad. For example, Olympic races are split into men’s races and women’s races because men are, on average, faster than women. However, looking at [the distribution of running speed in the New York marathon](http://www.warandgender.com/wggendif.htm), where women and men run together, it’s clear that running speed has a high degree of variability, and that the distributions for men and women overlap a great deal.

One way to interpret a t-value is as the number of standard errors worth of space separating the group means. A t-value of 2 would indicate that the means are two standard errors apart.

The p-value associated with a t-test indicates the likelihood of getting a difference this large or larger in the samples if the populations were not different. The smaller the p-value, the more likely the difference we see in the samples meaningfully reflects the populations. The p-value in the test you did above is really small, so we can be fairly confident that the difference in means we see is due to a real difference in the population and not due to variability in the samples.

# DRILL - Exploring the Central Limit Theorem

Now that you have some code to create your own populations, sample them, and compare the samples to the populations, it's time to experiment. Using your own Jupyter notebook, or a copy of the notebook from the previous assignment, reproduce the pop1 and pop2 populations and samples, using numpy's binomial function. Specifically, create two binomially distributed populations with n equal to 10 and size equal to 10000. The p-value of pop1 should be 0.2 and the p-value of pop2 should be 0.5. Using a sample size of 100, calculate the means and standard deviations of your samples.

For each of the following tasks, first write what you expect will happen, then code the changes and observe what does happen. Discuss the results with your mentor.

1. Increase the size of your samples from 100 to 1000, then calculate the means and standard deviations for your new samples and create histograms for each. Repeat this again, decreasing the size of your samples to 20. What values change, and what remain the same?
2. Change the probability value (p in the [NumPy documentation](https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.binomial.html)) for pop1 to 0.3, then take new samples and compute the t-statistic and p-value. Then change the probability value p for group 1 to 0.4, and do it again. What changes, and why?
3. Change the distribution of your populations from binomial to a distribution of your choice. Do the sample mean values still accurately represent the population values?

When you've given it a try, you can find a sample solution [here](https://github.com/Thinkful-Ed/data-201-resources/blob/master/solutions/Prep%20course/3.3.7.ipynb).

In this lesson, you were introduced to probability distributions, focusing on the normal distribution, and to the Central Limit Theorem

# Narrative Analytics

Moving forward, you should feel comfortable with the following:

* Differentiating between normal and non-normal distributions
* Explaining the role of the central limit theorem in generalizing from samples to populations
* Calculating and comparing group means using Python

If you're unclear on any of the above, be sure to follow up with your ment

As you're approaching the end of the fundamentals course, we've covered a lot of ground. You've begun to work with Python, learned some of the basics of the data science toolkit, and been introduced to some of the essentials of statistics for data science.

Before we move on, we have to put it all together, which we will do under the umbrella of narrative analytics. This concept of narrative analytics will be a driving force throughout the bootcamp. In this lesson we'll introduce the concept explicitly and provide some key examples to help illustrate how you'll be able to root your work throughout the bootcamp in a narrative form.

# Narrative Analytics

 Estimated Time: **1 hour**

Narrative analytics is the idea of telling a story with data, and it will play a key role in everything you do in this course and throughout your career in data science. As we explore this concept in this lesson, it's important to acknowledge that the job of a data scientist is not always narrative.

Particularly when dealing with technical coworkers, sometimes the job of a data scientist is to just gather and present information with as little influence on the process as possible. This is a more passive form of data science, focused almost solely on the integrity of the data.

However, we will by and large not emphasize that kind of data science, for a few reasons. Firstly, the skills for passive analytics (the math, stats, data collection) are still necessary for narrative analytics. Adding a narrative just puts another layer on top of that work. Secondly, learning the narrative form and how to make a case with data is a difficult skill to master, requiring constant practice and reinforcement. Lastly, adding a narrative is often a key value add of a data scientist. How so? We'll explore that below.

## Everything is a story...

As much as we acknowledge it above as a valid approach to data science, there's really no such thing as truly passive data science. Everything contains some element of narrative to it. Every piece of data says something, and how you present those points inherently affects how such data is consumed. Understanding how that happens is key to being a successful data scientist.

Let's start with a simple example.

Let's say you work at a furniture store and sales for last week were $20,502.

That is a piece of data operating in a relatively minimal amount of context. Perhaps to the furniture store owner, who has experience and context, they can take that piece of information and put it into some kind of understanding of whether those sales were good or bad, up or down, expected or anomalous. However, with just that single piece of data none of that other information is conveyed. The story is short, and it is largely left to the person consuming the data to fill in the narrative around it.

Now let's think of another way we could talk about sales: "Sales last week were $20,502, down 1% over the previous week."

This provides a little more context, but while this may seem dispassionate, there are several choices that were made here that will shape interpretation. Firstly, it only looks back a single week. It doesn't tell us anything about any larger trend that exists or put it into the context of how much volatility we typically see in sales.

If we compared against average sales for the past year, we might see something like sales that week as 10% above the mean. Is that a more descriptive metric? It certainly makes it sound like sales were good. But is the trend improving or leveling off?

## Answers are always a partial picture

Any time you answer a question about data you're inherently summarizing the data. That means something is going to be lost. It's unavoidable. Any interpretation is a version of taking a stance about what the data means.

As a data scientist, your job is not to tell the whole story but to tell the best one you can with the information you have at hand. Sometimes it's to argue for a specific reaction. Sometimes it's to present a certain version of events. If you think sales are weak you're probably not going to compare to last year's average.

Recognize that any way you present your data presents a version of events. It's not that one version is true or one is false. It's that they're different. That's why asking the right question, and not just that but asking a lot of questions, is essential to finding out what is going on with your data.

## Aren't we supposed to be dispassionate?

There's a mistrust around data that it is manipulable. A really talented statistician or data scientist can find ways to present the same dataset to tell contradicting stories. Does that mean any of our work can be dispassionate or done with integrity?

Of course it can.

There are several key things to keep in mind when working with data to ensure that you arrive at the closest thing to truth you can find.

Firstly, try not to decide a story before you look into the data. The analysis you perform should inform the narrative you want to tell, not the other way around.

Always try to ask what else an answer could mean. When you perform analytics, there is often a conclusion that is easiest to draw. Seeing that sales had a down week could easily mean that sales are in trouble. Spending the time to question that conclusion is one of the key features of a good data scientist.

Lastly, be aware that you can always be wrong. Conclusions aren't 100%. There's never just one interpretation. Keep questioning what's going on. Keep asking more questions.

Remain curious.

# What Makes a Good Visual

 Estimated Time: **1 hour**

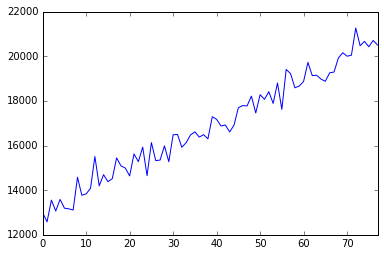
Now, when we talk about the analysis we perform, there are two main things we present. First is numeric summarization and analytics. These are the kinds of things we talked about previously. Statistical tests, measures of central tendency. Those kinds of things.

Then there are visualizations.

Being able to make good visualizations is one of the most valuable skills a data scientist can possess. It is remarkable how often all the statistics and analytics are done and lined up making a compelling statistical case, but people do not buy into it until they see the visualizations.

## Visuals as narrative device

Just like summary statistics, visualizations are a narrative device in your analytic reporting. We're going to continue with the sales example from the previous lesson.



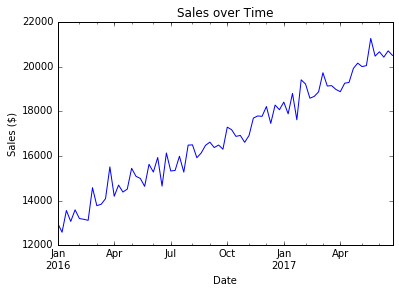
Here we have a plot of sales over time, specifically a line graph, which we covered earlier in the prep course. However, this is not a very good plot. The axes are unlabeled. There's no title. We really know nothing about the plot other than something seems to be going up and to the right.

Why does this matter if you know we're talking about sales? There is a simple fact worth remembering when doing analytic reporting: most people won't read the text. It's a sad truth, because the text is often essential to filling out the context, but most of the time people are busy or distracted or whatever else and are only going to look at the visuals.

As such, it is essential that the visualizations you create can stand on their own, supplying their own context and presenting their own conclusions. A person should be able to look at your graph and understand everything they need to interpret it.

## Elements of a good visual

Let's try a second visualization.



This is a much better visual. We have labels on the axes and a title. You can look at this and quickly understand that sales have been steadily rising for the past year and a half, dating back to January 2016. Because this is a single series of data, a legend isn't really necessary, but if you had multiple lines on a single graph a legend would be valuable.

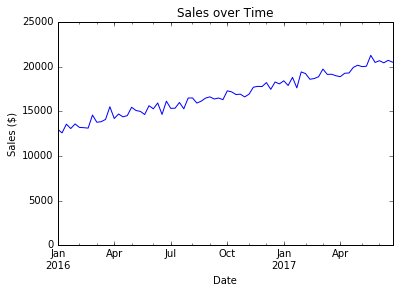
It can be tedious to add these elements to the chart, get the colors right, and place everything in the right place and at the right size so it's all easily visible and comprehensible. That's ok, it really is worth it and it's a critical part of your job. You want your audience to be able to easily consume the information you're sharing, and this is a powerful way to ensure that.

## Visuals imply a stance

Now, this lesson is all about narrative analytics, but we've been talking about visualizations in a relatively absolute way. They should always be easy to read. They should always have proper labeling.

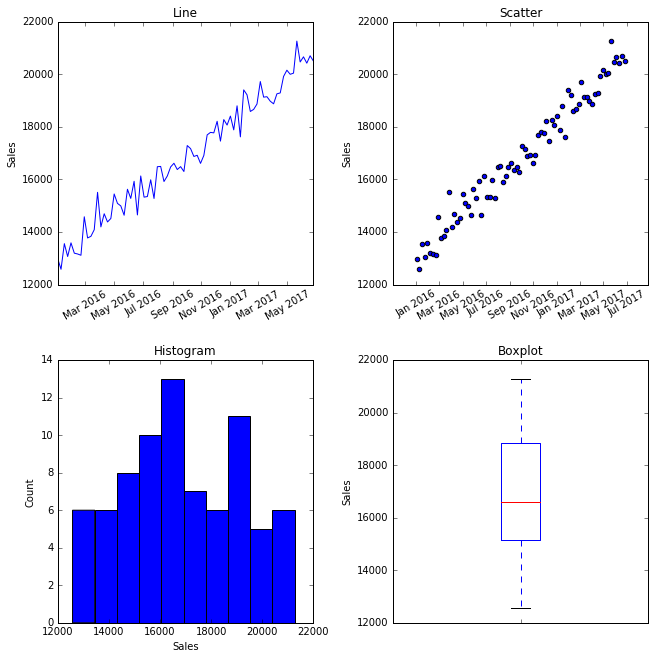
But do they imply a narrative in the same way as we discussed earlier?

Of course! There are several different ways to present the same data, and each one invites slightly different conclusions. First, let's use the simple example of axes. Here's the same data as above, presented with different axes:



While there is still an upward trend, it doesn't appear as stark as it did previously. This is because we've now rooted our y-axis at $0, rather than letting it float to $12,000. Neither of these plots is "right" or "wrong", but they do imply two different stories at first glance.

Of course there's also different plots that we could use, and each one tells a different story with the data.



Before you move on, look at these different plots and think about how they convey very different information even though they come from the same data. As you move through the course, remember to make your visualizations as clear and precise as possible. They should have everything you need to understand them, and nothing you don't.

# Narrative Analytics Guided Exam Narrative Analytics Guided Example

Here we're going to dive into a dataset and perform some analytics, working with a dataset from the [Tate](http://www.tate.org.uk/) in the UK. Also note that data and guided example makes a decent example for your capstone report.

# Tate Collection Data Exploration

## Data

We will be working with the Tate's dataset of artists and artworks. This data was released by the Tate, a collection of art museums in the UK, representing 70,000 works either owned by the Tate or jointly with The National Galleries of Scotland as part of their Artist Rooms Series.

For our purposes we are going to assume that it is representative of their larger collection and that it is also therefore representative of the Tate's art collecting in general post 1800 (the earliest acquisition in this dataset was acquired in 1823).

The dataset itself consists of 2 tables, artists and artworks. There are also individual json files for each artist and artwork, but we are not going to pull all of those individual files together.

The artists file contains an id, name, gender, year of birth, year of death, place of birth, place of death, and a url for the Tate's page on the artist. The works file contains accession number, id, artist, artist's role, artist id, title, date, medium, credit line, year, acquisition year, dimensions, inscription, and links to the image and the artwork's Tate webpage.

There are several challenges to this dataset. First, different kinds of artwork will demand different things from their fields. The role field also potentially holds a lot of interesting relationships to art practice.

## Analytic Questions

**#1 Who are the most popular artists in the Tate Collection? Are there any outliers in terms of amount collected?**

Let's start with a simple question about popularity. Which artists and time periods has the Tate prioritized collecting? First we'll approach artists.

In [3]:



1

works.artist.value\_counts().head(20).plot(kind='bar')

2

plt.ylabel('Artworks Count')

Out[3]:

<matplotlib.text.Text at 0x107347c18>

This plot of artworks by artists for the top 20 artists really only shows us one thing (other than providing a list of the 20 most popular artists): The Tate has a lot of works by William Turner.

If anyone knows about this history of British museums this is not a surprise. Turner, an immensely popular artist in his day, left all of his works in his possession to Britain and therefore the Tate as part of what is called The Turner Bequest. It makes up a large portion of the Tate Britain's museum in London.

To look at the relative popularity of other artists let's remove Turner.

In [4]:



1

works.artist.value\_counts().head(50)[1:51].plot(kind='bar', figsize=(10,5))

2

plt.ylabel('Artworks Count')

Out[4]:

<matplotlib.text.Text at 0x10278fef0>

So there seem to be a few other exceptionally popular artists, with the first four or arguably six being collected in meaningfully larger numbers. Note Andy Warhol's presence here as one of the more currently well known artists on this list.

Remember, this is popularity in terms of pieces collected. We have no data about visits or webviews.

**#2 Who are the artists in the Tate collection? How does that vary by geography, age, and living or dead?**

In [5]:



1

len(artists)

Out[5]:

3532

So there are 3,532 artists in the Tate collection. Where are they from? When looking into the placeOfBirth variable, there is a challenge, however. The format is inconsistent. Sometimes the birthplace includes a city, others just a country.

In [6]:



1

*# Process data to create counts by country*

2

​

3

*# Split the place of birth on commas*

4

locations = artists.placeOfBirth.str.split(',', 1).tolist()

5

locations = [x **for** x **in** locations **if** str(x) **!**= 'nan']

6

countries = []

7

​

8

*# Process countries and clean up text*

9

**for** entry **in** locations:

10

c = entry[**-**1]

11

c = c.strip()

12

countries.append(c)

13

countries = pd.DataFrame(countries, columns=['country'])

14

​

15

*# Create numeric counts*

16

cntry\_counts = pd.DataFrame(countries.country.value\_counts())

17

other = int(cntry\_counts[10:].sum())

18

cntry\_counts = cntry\_counts[:10]

19

cntry\_counts.loc[11] = other

20

cntry\_counts = cntry\_counts.rename(index={11: 'Other'})

21

​

22

*# Generate Pie Chart*

23

plt.figure(figsize=(10, 5))

24

plt.pie(cntry\_counts.country)

25

plt.axis('equal')

26

plt.title('Artists Country of Birth')

27

plt.legend(cntry\_counts.index)

Out[6]:

<matplotlib.legend.Legend at 0x1074c5a20>

So, about half of the artists in the collection are from the UK, which again is not hugely surprising as this is a British collection. The two things of note we see here are that the US and Canada are the only two non-European countries in the top 10. Also, the other countries selection is quite large, with a very large number of countries having some representation in the tate collection and making up almost a quarter of their collection.

In [7]:



1

plt.plot(artists.yearOfBirth.value\_counts().sort\_index())

2

plt.title('Artists Born by Year')

Out[7]:

<matplotlib.text.Text at 0x107eb2c88>

You can see that the closer to modern times we get, the more artists we have represented. You see some interesting peaks around the centuries that are perhaps worthy of further investigation. Maybe they're using something other than artist's names to talk about movements?

How does this compare to when artworks were acquired?

In [8]:



1

acquisition\_df = pd.DataFrame(works.acquisitionYear.value\_counts())

2

acquisition\_df = acquisition\_df.sort\_index()

3

plt.plot(acquisition\_df)

4

plt.ylabel('Works Acquired')

Out[8]:

<matplotlib.text.Text at 0x107eeb208>

This shows a collection that seems to have several peaks in its growth. That is consistent with how museum collections tend to grow. While there is some steady acquisition, which is somewhat visible in the more modern years (though this is not a great visual of that), museums tend to see the majority of their growth from large gifts or bequests. The Turner Bequest, again, is the most visible. What does it look like without Turner?

In [9]:



1

acquisition\_df = pd.DataFrame(works[works.artist **!**= 'Turner, Joseph Mallord William'].acquisitionYear.value\_counts())

2

acquisition\_df = acquisition\_df.sort\_index()

3

plt.plot(acquisition\_df)

4

plt.ylabel('Works Acquired')

Out[9]:

<matplotlib.text.Text at 0x107eddeb8>

This shows a few more clear spikes but also a clear narrative that the collection has been growing more rapidly in recent years. This aligns with the skew towards contemporary artists.

Let's move on to one last subject: determining the portion of the artists who are living. For this, we'll use year of death as an indicator.

In [10]:



1

living = pd.DataFrame(artists.yearOfDeath.isnull())

2

living = pd.DataFrame(living.yearOfDeath.value\_counts())

3

living.plot(kind='bar', legend=**False**)

4

plt.title('Artists Who are No Longer Living')

Out[10]:

<matplotlib.text.Text at 0x1090af4e0>

This shows a surprisingly large portion of the artists collected by the Tate are still living. However, when put into context that the Tate is one of the largest supporters of contemporary art in Britain, hosting the largest prize for contemporary art with the Turner Prize, it does fit their profile.

**#3 What are the most popular mediums and how does medium affect size?**

So it would be tempting to start with medium just as the data provides it. However, this reveals a bit of a problem.

In [11]:



1

works.medium.value\_counts().head(10)

Out[11]:

Graphite on paper 26167

Oil paint on canvas 3383

Screenprint on paper 2984

Lithograph on paper 2721

Watercolour on paper 1890

Etching on paper 1793

Graphite and watercolour on paper 1680

Ink on paper 880

Intaglio print on paper 820

Photograph, gelatin silver print on paper 750

Name: medium, dtype: int64

There are way too many kinds of medium, and with a level of subtlety that we don't really want. We'll group some together.

We're also dropping Turner here because he has 25,000 works on paper that skew all counts towards that.

In [12]:



1

*# Remove Turner*

2

turnerless\_artworks = works[works['artist'] **!**= 'Turner, Joseph Mallord William']

3

*# Coerce to Numeric*

4

turnerless\_artworks.height = pd.to\_numeric(turnerless\_artworks.height, errors = 'coerce')

5

turnerless\_artworks.width = pd.to\_numeric(turnerless\_artworks.width, errors = 'coerce')

6

turnerless\_artworks.depth = pd.to\_numeric(turnerless\_artworks.depth, errors = 'coerce')

7

turnerless\_artworks = turnerless\_artworks[turnerless\_artworks['units']=='mm']

8

turnerless\_artworks = turnerless\_artworks[turnerless\_artworks.height.notnull()]

9

​

10

*## The error is just because of how we did the conditional select and we don't need to be worried about it...*

/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/pandas/core/generic.py:2773: SettingWithCopyWarning:

A value is trying to be set on a copy of a slice from a DataFrame.

Try using .loc[row\_indexer,col\_indexer] = value instead

See the caveats in the documentation: <http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy>

self[name] = value

In [13]:



1

*# Aggregate to new medium\_agg column*

2

turnerless\_artworks['medium\_agg'] = 'other'

3

turnerless\_artworks.loc[turnerless\_artworks['medium'].str.contains("paper", na=**False**),'medium\_agg'] = 'paper'

4

turnerless\_artworks.loc[turnerless\_artworks['medium'].str.contains("canvas", na=**False**),'medium\_agg'] = 'canvas'

5

turnerless\_artworks.loc[turnerless\_artworks['medium'].str.contains("wood", na=**False**),'medium\_agg'] = 'wood'

6

turnerless\_artworks.loc[turnerless\_artworks['medium'].str.contains("paint on", na=**False**),'medium\_agg'] = 'other painted panel'

7

turnerless\_artworks.loc[turnerless\_artworks['medium'].str.contains("Bronze", na=**False**),'medium\_agg'] = 'sculpture'

8

turnerless\_artworks.loc[turnerless\_artworks['medium'].str.contains("Plaster", na=**False**),'medium\_agg'] = 'sculpture'

9

turnerless\_artworks.loc[turnerless\_artworks['medium'].str.contains("Marble", na=**False**),'medium\_agg'] = 'sculpture'

10

turnerless\_artworks.loc[turnerless\_artworks['medium'].str.contains("Stone", na=**False**),'medium\_agg'] = 'sculpture'

11

turnerless\_artworks.loc[turnerless\_artworks['medium'].str.contains("plate", na=**False**),'medium\_agg'] = 'plate'

12

turnerless\_artworks.loc[turnerless\_artworks['medium'].str.contains("photograph", na=**False**),'medium\_agg'] = 'photo'

13

​

14

turnerless\_artworks['surface'] = turnerless\_artworks.height **\*** turnerless\_artworks.width

In [14]:



1

turnerless\_artworks[['medium\_agg','height','width','depth','surface']].groupby('medium\_agg').describe()

Out[14]:

|  |  | **depth** | **height** | **surface** | **width** |
| --- | --- | --- | --- | --- | --- |
| **medium\_agg** |  |  |  |  |  |
| **canvas** | **count** | 83.000000 | 260.000000 | 2.590000e+02 | 259.000000 |
| **mean** | 134.957831 | 1776.396154 | 3.424765e+06 | 1495.119691 |
| **std** | 355.563519 | 1543.269130 | 3.831638e+06 | 890.336830 |
| **min** | 5.500000 | 105.000000 | 1.260000e+04 | 120.000000 |
| **25%** | 27.500000 | 729.250000 | 5.518635e+05 | 727.500000 |
| **50%** | 35.000000 | 1512.500000 | 2.283380e+06 | 1410.000000 |
| **75%** | 53.000000 | 2290.750000 | 4.707936e+06 | 2135.000000 |
| **max** | 2185.000000 | 16000.000000 | 2.173195e+07 | 4860.000000 |
| **other** | **count** | 953.000000 | 1301.000000 | 1.298000e+03 | 1298.000000 |
| **mean** | 761.288772 | 1189.179554 | 1.880836e+06 | 960.942604 |
| **std** | 1448.336740 | 1866.683697 | 4.996881e+06 | 1019.110727 |
| **min** | 3.000000 | 6.000000 | 1.681000e+03 | 5.000000 |
| **25%** | 125.000000 | 273.000000 | 8.754350e+04 | 290.000000 |
| **50%** | 300.000000 | 620.000000 | 3.955995e+05 | 603.000000 |
| **75%** | 750.000000 | 1460.000000 | 1.597850e+06 | 1314.750000 |
| **max** | 18290.000000 | 37500.000000 | 9.125394e+07 | 10900.000000 |
| **other painted panel** | **count** | 442.000000 | 4408.000000 | 4.408000e+03 | 4408.000000 |
| **mean** | 76.997738 | 1032.772913 | 1.392090e+06 | 960.962568 |
| **std** | 290.694723 | 820.503273 | 2.247036e+06 | 639.367519 |
| **min** | 2.000000 | 45.000000 | 3.306000e+03 | 50.000000 |
| **25%** | 20.000000 | 514.000000 | 2.917325e+05 | 508.000000 |
| **50%** | 30.000000 | 787.000000 | 6.247550e+05 | 762.000000 |
| **75%** | 55.000000 | 1270.000000 | 1.558060e+06 | 1244.250000 |
| **max** | 5486.000000 | 11900.000000 | 3.623310e+07 | 4580.000000 |
| **paper** | **count** | 160.000000 | 19800.000000 | 1.978000e+04 | 19780.000000 |
| **mean** | 277.478125 | 417.212288 | 2.608455e+05 | 407.575925 |
| **std** | 695.113346 | 379.088368 | 7.540381e+05 | 326.287752 |
| **min** | 1.000000 | 15.000000 | 2.370000e+02 | 3.000000 |
| **25%** | 28.250000 | 200.000000 | 3.904200e+04 | 190.000000 |
| **50%** | 45.000000 | 320.000000 | 1.029940e+05 | 320.000000 |
| **...** | **...** | ... | ... | ... | ... |
| **photo** | **std** | 467.421236 | 2106.887208 | 6.188098e+06 | 962.547810 |
| **min** | 18.000000 | 90.000000 | 1.332000e+04 | 100.000000 |
| **25%** | 25.000000 | 303.500000 | 9.896750e+04 | 380.000000 |
| **50%** | 39.500000 | 690.000000 | 3.884175e+05 | 608.000000 |
| **75%** | 146.500000 | 1395.250000 | 1.696120e+06 | 1202.750000 |
| **max** | 2015.000000 | 19890.000000 | 5.431500e+07 | 4892.000000 |
| **plate** | **count** | 8.000000 | 344.000000 | 3.440000e+02 | 344.000000 |
| **mean** | 1997.625000 | 347.148256 | 1.258063e+05 | 264.523256 |
| **std** | 4215.212448 | 478.247577 | 4.437804e+05 | 210.308546 |
| **min** | 25.000000 | 76.000000 | 7.752000e+03 | 102.000000 |
| **25%** | 75.000000 | 305.000000 | 6.984500e+04 | 229.000000 |
| **50%** | 585.500000 | 305.000000 | 6.984500e+04 | 229.000000 |
| **75%** | 1107.750000 | 305.000000 | 6.984500e+04 | 229.000000 |
| **max** | 12360.000000 | 8850.000000 | 6.549000e+06 | 2235.000000 |
| **sculpture** | **count** | 620.000000 | 639.000000 | 6.390000e+02 | 639.000000 |
| **mean** | 406.875806 | 619.608764 | 6.975306e+05 | 765.388106 |
| **std** | 504.036311 | 791.822922 | 1.405889e+06 | 657.149556 |
| **min** | 8.000000 | 18.000000 | 1.296000e+03 | 19.000000 |
| **25%** | 152.000000 | 222.000000 | 7.777050e+04 | 304.000000 |
| **50%** | 270.000000 | 406.000000 | 2.032000e+05 | 533.000000 |
| **75%** | 445.000000 | 714.500000 | 6.622965e+05 | 1029.000000 |
| **max** | 5800.000000 | 11250.000000 | 1.796402e+07 | 3750.000000 |
| **wood** | **count** | 208.000000 | 349.000000 | 3.490000e+02 | 349.000000 |
| **mean** | 548.336538 | 1067.189112 | 2.160677e+06 | 1065.908309 |
| **std** | 923.748764 | 1597.806055 | 8.422669e+06 | 1145.166249 |
| **min** | 6.000000 | 30.000000 | 6.300000e+03 | 35.000000 |
| **25%** | 91.500000 | 340.000000 | 1.416000e+05 | 400.000000 |
| **50%** | 260.000000 | 611.000000 | 4.966000e+05 | 756.000000 |
| **75%** | 510.000000 | 1073.000000 | 1.247400e+06 | 1310.000000 |
| **max** | 6300.000000 | 14850.000000 | 1.324620e+08 | 11960.000000 |

64 rows × 4 columns

In [15]:



1

​

2

turnerless\_artworks[['medium\_agg', 'height']].boxplot(by='medium\_agg', figsize=(10,20))

3

​

Out[15]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x109093e48>

# ple

You should now be more familiar with the concept of narrative analytics. This is an idea that will come up again and again throughout the course, and indeed some of the topics that we covered here had been introduced in less explicit ways previously.

Going forward, be sure to think about how you use statistics and visualizations to tell a story. There are countless choices made whenever you work with data, and each one of those shapes the story you are telling. It's not that there is one right way to do it, but be aware that there are many right ways and you have to pick the one that does the best job conveying what you are intending.

4

[Report a typo or other issue](javascript:void(0);)

# Unit4 Career planning and capstone report

You've completed your introduction to programming in Python, fundamental data science tools, and summary statistics. You're ready to apply what you've learned to a real-world project and technical evaluation. But first we'll take an interlude to discuss the reason you're most likely here: making your career as a data scientist.

In this lesson we'll take a break from the tech and math topics we've been focusing on and tackle something much squishier: the job market and your future career. The following assignments are designed to help you explore the variety of data science work being done, understand the skills companies are looking for, find your future professional community, and create a preliminary vision for your career.

As a starting point, make a copy of this [career planning document](https://docs.google.com/document/d/1lMU3tzcSd9TRRjJxUgqK_xYqegG_xQaEA6fsS-AkHbE/edit) for yourself. You'll use it to organize your work in the coming lesson and then review it with your mentor and your program manager.

# Explore the landscape

 Estimated Time: **2-3 hours**

The world of data science is broad and loosely defined, and attracts people from a variety of professional backgrounds. Many people have tried to define what, exactly, data science is and what skills and competencies are required to succeed in this profession, but there is no single generally accepted definition or taxonomy of skills. The upside is that the variety allows you to specialize your career according to your strengths and interests. The downside is that communicating what you do and how you create value to employers, or even to yourself, can be tricky.

In the twenty minute presentation from DataGotham below Harlan Harris talks through his data-driven research project on this topic. Give it a watch, and as you do see if his conclusions are consistent with your own experience.

Harlan presents five skill groupings:

* programming,
* statistics
* math / operations research
* business, and
* machine learning / big data

as well as four "self-id" groupings:

* "data businessperson"
* "data creative"
* "data researcher", and
* "data engineer".

These distinctions are useful to make, but they're far from the only way to slice and dice the problem. Other people try using [venn diagrams](http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram?rq=data%20science%20venn%20diagram) to make sense of the world of data science and still others divide it into profiles like [communicators, visualizers, data wranglers, modelers, programmers, and technologists](https://www.mango-solutions.com/blog/launching-the-data-science-radar).

Take a few minutes to read the articles linked above, then complete [the survey behind Harlan's talk](http://survey.datacommunitydc.org/) above and this data science ["radar" challenge](https://www.mango-solutions.com/radar/), recognizing that you may not be familiar with everything those surveys ask and that's fine. Answer as you would once you've completed this bootcamp. Reflect on your results, the content above, and what data science means to you for a paragraph or three in the first section of your career planning document.

# Survey the job market

Knowing what aspect(s) of data science you'd like to work on is all well and good. But most of us like to eat, and to eat you'll probably need to create business value (for yourself or for a company) and turn some of that value into income.

## Job postings

Survey the job postings in the city where you plan to work at the end of this bootcamp and pull out five job advertisements you're most excited about. Add them to your career planning document, along with a sentence about why you chose that particular job. Here are some suggestions of places to find data science job postings.

* [Angel list](https://angel.co/jobs)
* [LinkedIn](https://www.linkedin.com/jobs/search?keywords=data+scientist&location=Greater+New+York+City+Area&trk=jobshomev2_2boxsearch&orig=JSHP&locationId=us%3A70)
* [Indeed](https://www.indeed.com/q-Data-Scientist-jobs.html)
* [Kaggle](https://www.kaggle.com/jobs)
* [Glassdoor](https://www.glassdoor.com/Job/data-scientist-jobs-SRCH_KO0,14.htm)
* [Data Science Central](http://www.datasciencecentral.com/page/job-board)
* [Data Elixir](https://jobs.dataelixir.com/)
* Individual company job pages, like Facebook's ["Data" jobs in Seattle](https://www.facebook.com/careers/search/?q=data&location=seattle).

If appropriate, expand your search to include related job titles with "Data" in them like Data Architect, Data Engineer, Data Manager or Data Analyst, and titles with related keywords like "Machine Learning" or "Statistician".

## Company-first research

Next, turn your attention away from job postings and towards companies, whether or not they're hiring. Your goal is to put together a short hit list of your top five dream companies. Add them, along with a sentence about what they do and why you chose each one, to your planning document. If any of these companies overlap with the job postings you found above, add additional companies to your hit list until you have five total novel companies listed here.

# Find your people

 Estimated Time: **1 hour**

You're moving into an industry with a strong and active culture. It's important for you to become familiar with that culture and to get to know people working now in the job you want.

## Events

Many cities have an active data science event scene. The best place to learn about local events is meetup.com. Search meetup (here are [relevant events near Seattle](https://www.meetup.com/find/?allMeetups=false&keywords=data&radius=10&userFreeform=seattle&gcResults=Seattle%2C+WA%2C+USA%3AUS%3AWashington%3AKing+County%3ASeattle%3Anull%3Anull%3A47.6062095%3A-122.3320708&change=yes&sort=default&eventFilter=mysugg), for example) and list each group you might like to check out on your planning document.

## People

During the bootcamp you'll learn how to network effectively in order to gather useful intelligence about hiring companies and teams, find unlisted job openings, skip past initial applicant filters, and earn referrals from your peers.

For now, simply find LinkedIn profiles for two people: (1) a person you don't know who is currently working at a company on your hit list and has a job title you want, and (2) a first or second degree connection in the industry.

When [searching LinkedIn](https://www.linkedin.com/search/results/people/) take advantage of their filters. If you don't have first or second degree data science connections in your LinkedIn network right now, add [Grae](https://www.linkedin.com/in/graed) or [Darrell](https://www.linkedin.com/in/darrellsilver) and do another pass. We get a lot of LinkedIn connection requests so if you send one be sure to note that you're connected through Thinkful.

# Write your own story

 Estimated Time: **2-3 hours**

Think about your ideal career two or three years from now. What particular skills have you built? What techniques and tools are your specialty? What accomplishments might you have made? Is your title "Data Scientist", or something different? Are you working in a particular industry, or solving a particular class of problems?

For this final assignment you have writing prompts about the ideal job you envision above. Write responses to each and add them to your planning document.

1. Describe your ideal job in one or two paragraphs as though you're talking with an industry professional. They're familiar with the business and with the culture, they know the jargon and the buzzwords. Focus on the skills, tools, and / or industry you hope to deeply specialize in.
2. Describe your ideal job in one or two paragraphs as though you're talking with a non-technical family member or friend. Avoid jargon and keep things simple while remaining specific. If you went deep in your specialization above, emphasize here the breadth of skills and work.
3. Synthesize the two exercises above into a short aspirational "professional summary" that might be appropriate on your LinkedIn profile, resume, or portfolio in the future.

When you've completed this assignment submit a link to your career planning document below and [schedule a session with Matt](https://calendly.com/mattshull/career-call) to discuss the job search.

# Capstone Data Analysis Report

 Estimated Time: **15-17 hours**

Now that we’ve reached the end of the course it’s time for you to demonstrate what you’ve learned. First, you'll complete a technical evaluation that covers the programming and statistics fundamentals you've worked on in this course. Next, as a capstone to this course, you'll complete an Analytic Report and Research Proposal on a data set of your choosing. Finally, you'll have your first mock interview, where you'll present and defend your capstone during a call with a member of the Thinkful data science team

# Technical evaluation

 Estimated Time: **1-2 hours**

Over the past few weeks, you’ve worked hard to build a foundation in programming and statistics. You’ve learned to solve problems using Python, implement the data science toolkit, generate different types of visualizations to represent datasets, and apply summary statistics to tell a story about a dataset. You should be proud of the work you’ve done to get this far!

To ensure you're prepared to enter the full Data Science Bootcamp, you'll complete a technical evaluation that covers concepts you've learned over the past few weeks.

## Evaluation structure

The technical evaluation consists of 15 multiple choice questions and should take you between 30 and 60 minutes to complete. The first 10 questions are each worth 1 point, and the last 5 questions are each worth 2 points, for a total of 20 possible points. You need to score 16 points to pass the evaluation.

Some of the questions test your knowledge of statistical concepts, while other ones require you to use the data science toolkit to analyze data we provide.

Before starting, download [this dataset](https://tf-curricula-prod.s3.amazonaws.com/assets/plane_crashes_data.csv), which contains data about plane crashes between 1950 and 2009. Fire up a Jupyter notebook and create a dataframe with this data. Furthermore, you'll have to run a Jupyter notebook and use the data science toolkit in order to answer the questions that ask for computations on the dataset.

This is an open book test, and you're welcome to consult the curriculum to answer questions. You should not, however, seek help from your mentor or other students.

You may only submit your evaluation **once**; any subsequent submissions will be omitted.

When you're ready, you can take your [technical evaluation here](https://docs.google.com/forms/d/e/1FAIpQLSfi3ItSorSkxNxNErepZy4OABxfKdfT6qpxkbWmIfQJElDFOA/viewform).

## Next steps

Once you've submitted your technical evaluation, you'll have the opportunity to review the questions you missed and the correct answers. Your performance on this evaluation and your capstone, as well as feedback from your mentor, will all be considered when determining your admission to the full Data Science Bootcamp. The next lesson covers the requirements for your capstone.

# Capstone Analytic Report and Research Proposal

 Estimated Time: **13-15 hours**

As a capstone to this fundamentals course, prepare an Analytic Report and Research Proposal on a dataset of your choosing. Your Report should accomplish these three goals:

1. **Describe your dataset**. Describe and explore your dataset in the initial section of your Report. What does your data contain and what is its background? Where does it come from? Why is it interesting or significant? Conduct summary statistics and produce visualizations for the particular variables from the dataset that you will use.
2. **Ask and answer analytic questions**. Ask three analytic questions and answer each one with a combination of statistics and visualizations. These analytic questions can focus on individuals behaviors or comparisons of the population.
3. **Propose further research**. Lastly, make a proposal for a realistic future research project on this dataset that would use some data science techniques you'd like to learn in the bootcamp. Just like your earlier questions, your research proposal should present one or more clear questions. Then you should describe the techniques you would apply in order to arrive at an answer.

See this [recent analysis on 2016 celebrity deaths](https://medium.com/@jasoncrease/was-2016-especially-dangerous-for-celebrities-79d79b9fae02#.zd8hv5jge) for an excellent example of data-driven story telling that presents a problem, explores data, and produces an answer. The analytics are more robust techniques than we've covered so far, but the general idea and tone are spot on.

## Report guidelines

Keep these guidelines in mind as you draft your Report:

* **Length**. Your Report should be three to five pages long with visualizations. Short and clear is better than long and opaque[.](https://en.wikipedia.org/wiki/Obfuscation#Eschew_obfuscation)
* **Structure**. Pay attention to the narrative structure of your Report. Each section should flow into the next and make a logical, readable text. Don't simply create a list of bullet points or present code without explanation.
* **Format**. The best format for your Report is an interactive Jupyter notebook ipynb file. However, you are welcome to use any format you like, so long as you're able to include visualizations and include (or link to) the code you use to generate your visualizations and summary statistics. If a Jupyter notebook would be too much overhead or unduly distract you from creating good content, markdown files (hosted perhaps on GitHub or as a gist), blog posts, or even Word or Google documents are acceptable.

## Getting started

Your first step is choosing an interesting dataset to work with. You're welcome to use any dataset you like. If you aren't sure which one to use or are looking for inspiration, check out this [collection of open data sources](https://github.com/Thinkful-Ed/data-201-resources/blob/master/data-sources.md). Before deciding on a particular dataset think about the kinds of questions you might be able to answer. Consider the format of the data. Do you know how to (or will you quickly be able to learn to) access and load it? Once you've chosen a dataset, write out some of those preliminary questions. Having them early will help guide your initial data exploration.

In order to conduct summary statistics and prepare visualizations you'll need to collect the data and load it into Python / pandas. Some of the data in the sources above will be in a format we didn't teach you to load in this fundamentals course. If necessary, refer back to the lesson on [working with files](https://courses.thinkful.com/data-201-prepv1/assignment/2.1.4) or the [pandas I/O documentation](http://pandas.pydata.org/pandas-docs/stable/io.html).

Once you've loaded your data, dig around with pandas and matplotlib to explore it. What variables does your data contain and what distributions do you think they have? Does the data bear on the preliminary questions you wrote down? What new questions might you answer? How does the data look when you plot it out?

At this point you should be ready to start writing your Report. Decide what format to use, which three analytic questions you'll ask and answer, which research questions you'd like to ask and which data science techniques might be appropriate to answering them. If necessary, do independent research now about the field of data science, or discuss the topic with your mentor, to decide which potential techniques you could use.

## Evaluation

You are encouraged to make use of every resource at your disposal in putting together your Report. This extends to getting preliminary feedback on your work from your mentor or from other friends and family. However, you should be ready to explain every decision, conclusion, and visualization you make and all of the code you write.

When you're ready, you can submit your final Report at the bottom of this page. Once submitted, be sure to [schedule a time to meet with someone](https://dashboard.thinkful.com/) for a capstone review. Also make sure that you've already [submitted your career story assignment](https://courses.thinkful.com/data-201-prepv1/project/4.1.4) and [scheduled a time to review the assignment in a group Career Q&A Session host by Matt](https://www.thinkful.com/open-sessions/qa-sessions/data%20science/). To help you prepare for your capstone review, here are a few examples of the types of questions you'll be asked:

* Did you have any challenges with this data?
* Why did you choose this dataset?
* How did your dataset inform the questions you chose to explore?
* What issues did you run into while analyzing your data?
* Imagine someone sees this visualization out in the wild, separated from your report. What conclusions would you expect them to draw? Is that the conclusion that you want them to draw?
* How could you make your conclusions more rigorous?

You should also take some time to review [the rubric that you'll be scored on](https://docs.google.com/spreadsheets/d/18Z0aaE6mWIhUomdzfVZqYeAzw51-iEz_lGHIPwM-3ls/edit#gid=0).

Here's a few last pieces of advice:

* Grammar matters. A lot. This should be a professional and easy to read document.
* State the questions you aim to answer clearly and answer them specifically. Make sure to use markdown to properly format your questions.
* **Including your code is required** but we should also be able to read your report and understand your visualizations without having to look at that code. Whether you include your code in the report with an iPython notebook or in a separate file is up to you.
* Your goal should be to give us an understanding of your dataset and the behaviors present in it. As such use analytics and statistics to tell a story about the data, don't just give us statistics without context.
* Try to translate real questions into statistical questions rather than simply ask statistical questions.
* Use at least 2-3 different types of charts from the fundamentals course to display the data.
* Be clear about any assumptions you make about the data and validate those assumptions if possible.
* Ensure that your dataset actually has the information to answer the questions you're asking. Does the dataset have a bias? Is it incomplete? Problems with your dataset can easily lead to problems in your analysis if you don't address them.

[Report a typo or other issue](javascript:void(0);)