

# Exercise 1. Introduction to Python Methods and Data Structures

In GIS modeling or GIS data management, you often need to process a series of steps to get your work done. You might have a workspace full of data to reproject, clip to a study area, or combine in some way to get to an output product. We also often need to process data in different ways depending on conditions – thus we need to make *decisions*, and while high-level decisions require considered thought by users there are many low-level decisions that can be aided by a script programming model.

The main purpose of script programming is to *automate* tedious work in processing our data, and to use *logic* to direct that process. I think those two words are key: *automate* and *logic*. They distinguish this activity from the more common interaction we use computers for most of the time. To communicate by email, to compose a document, or to design a map, we need to *interact*; to process a lot of data, we need to *automate* and use *logic* to guide the automation.

In geoprocessing script logic, we'll make decisions that allow us to, for example, handle rasters differently from vector data, or only set map projections for unprojected data, or process datasets collected only at certain times. For any serious GIS work, scripting and other forms of programming becomes a necessity, not an option.

In this exercise, we'll explore the use of Python to create scripts that allow us to use the vast suite of geoprocessing tools in ArcGIS Pro. All of the tools you can use from the toolbox or in a model can also be used in a Python script. And these scripts can be made into script tools that we can use like any other geoprocessing tools. We'll be doing this in a later section of the exercise.

Here's a guide to going through the exercise. Obviously all of the text is important or I wouldn't have written it. It will guide you through what you're needing to learn. But there will be certain parts where you need to respond, and I've used icons to point these out:

- ➡ This directs you to do something specific, maybe in the operating system or answer something conceptual.
- 💻 Coding you need to do, in the subsequent code cell.
- ❓ Questions to answer in the same markdown cell.
- Similar to a question, but requesting an interpretation you need to provide, in the same markdown cell

## Learning Python in a Jupyter Notebook

Well, you're here in a Jupyter Notebook, which provides us the ability to edit our code as well as Markdown, which you're seeing here. Let's just start by entering in code in a code chunk. The text you're reading now however is in a *Markdown* section, basically for formatted text, but if you run things in the code cell below, you'll get a result. The code cell assigns a scalar object (a type of variable) `x` with the number `5`, then submits that variable to the `print()` function to display what it holds.

- 💻 To run this code cell, press the Run button at the top or type Ctrl-Enter in the code cell.

```
In [ ]: x = 5  
        print(x)
```

5

## Learning more about the Jupyter Notebook or Jupyter Lab environment

Before we go any further, you should learn a bit about the Jupyter Notebook or Lab interface, whichever you're using.

→ Go through Help menu to learn your way around. Jupyter Notebook also has a tour. You'll find keyboard shortcuts, a more extensive reference, and Markdown formatting. And you'll see that there are links to references for a lot of things we'll be using in the class: Python, NumPy, Matplotlib, pandas.

## Writing a code cell

For the first code cell, we provided the code. From here on out you'll want to type in code into the chunk and run it. *Make sure to actually type it -- don't just copy it.* You actually learn by typing. And you should get in the habit of figuring out what you need to type and then typing without looking at the example. Start by studying the instructions, then without looking, type in the code to run it. *You'll learn the most by making mistakes and needing to fix them.*

Now create a character-string scalar named `msg` and assign it "Hello World". It should look like this:

```
msg = "Hello World"  
print(msg)
```

*Reminder: Look for the keyboard icon for when you're supposed to write code in the following code cell:*



```
In [ ]: #
```

Hello World

To get the square of a number, follow it with `**2`.

💻 Apply this to the `x` variable and print the result. `print(x**2)`

```
In [ ]: #
```

25

→ The next code cell should just be run; you don't need to try to figure it out. It provides multiple outputs from a code cell. You can save this with any notebook you create, using it as "boilerplate".

```
In [ ]: from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

## 1.1 Python variables

**What is a Variable?** It is critical that you understand what a variable is. That being said, there are two different meanings of the word variable based upon where you use it. In mathematics, a variable is something like `x` or `y` that holds a numeric value. In computer programming, this is extended to also mean other types of data like text, and the notion of *strings of characters* like "Hello World" are used. They can also be named `x` and `y` etc., but typically variables in computer programs have more descriptive names. The other meaning of the word variable is from the database world, and the GIS world by extension, where a variable might be a column in the database, called a *field* and given a name. In that world something that holds a single value (numerical, text, or other type) is called a *scalar*. But for now, we'll use the word variable for that.

Think of a variable as a box where you can store anything, even other boxes (variables). When programming, we represent variables with one or more letters of our choosing (`x`, `i`, `slope_raster`). We then assign a value to a variable with the equal sign "=" (`name = "Anne"`). This value can be in the format of a number, a string of text, a list, or a complex set of things. We can re-assign a new value to our variable as many times as we want – which is why we call it a variable – its value can vary. Type out the following in console to get a better sense of variables. Text behind the "#" is a comment:

### numeric variables

There are two types of numeric variables in python - **integer** and **floating point**. Integer variables hold integer numbers only. Floating point variables hold decimal point numbers.

- `x = 5` creates integer variable `x` and assigns 5 to it.. the value of `x` is 5
- `x = -17` the value of variable `x` is now -17
- `print(type(x))`
- `x = -23.568` variable `x` is now a floating point type with value -23.568 ]
- Then check its type again.
- `number = x + 10` creates variable `number` , and assigns the value of `x + 10` to it
- `print(number)` prints the value of `number`

Run some similar code below to explore how variables and printing work:

```
In [ ]: x = -17
print(type(x))
x = -23.568
print(type(x))

<class 'int'>
<class 'float'>
```

## string variables (made of text)

Important syntax: A string literal value must be in quotation marks, ei. "Hello"

- `msg = "Hello"` creates string variable msg and assigns "Hello" as its value
- `path = "c:/prog/mydata"` string variable path with value "D:/prog/mydata"

Run some similar code below to explore how string variables work. Make sure to create msg and path variables you'll use below

```
In [ ]: #
```

```
c:/prog/mydata
```

## Boolean variables (true or false)

Another data type is Boolean, meaning `True` or `False`. The constants that can be used to assign these are literally `True` and `False`, but these are typically created by evaluating an expression, like `5 > 6`



```
tf = 5 > 6
print(tf)
print(type(tf))
print(type(True))
```

```
In [ ]: #
```

```
False
<class 'bool'>
<class 'bool'>
```

It's often useful to realize that `True` can also be interpreted or entered as the integer `1`, and `False` is `0` as you can see by including the Boolean variable in a mathematical expression like `False * 2` or `True * 2`.

```
In [ ]: False * 2
```

```
Out[ ]: 0
```

```
In [ ]: True * 2
```

```
Out[ ]: 2
```

```
In [ ]: from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

In fact, any non-zero value will be interpreted as `True`, but `False` is always zero.

Enter some code that will create Boolean variables:

In [ ]:

## Functions, methods and properties

We'll be using a lot of functions, methods and properties, so we should know what they are.

- **Functions** are written as `functionName(input)`. We'll see a lot of these in the math module, but you can see what the built-in functions are at [\(https://docs.python.org/3/library/functions.html\)](https://docs.python.org/3/library/functions.html)

In [ ]:

```
import math
sin = math.sin
radians = math.radians
pi = math.pi
abs(-5)
sin(30/180*pi)
sin(radians(30))
```

Out[ ]:

5

Out[ ]:

0.4999999999999994

Out[ ]:

0.4999999999999994

- **Methods** are written as `obj.method(parameters)` and applies to that object. To see what methods apply to an object, type the dot and press tab. There aren't many for the simple numeric variables. Try it out, but here's one example.

```
y = 0.5
y.
y.as_integer_ratio()
```

In [ ]:

#

Out[ ]:

(1, 2)

In [ ]:

```
mystr = "hello"
mystr.capitalize()
```

Out[ ]:

'Hello'

- **Properties** are similar to methods in being applied to an object, but there are no parameters, thus no () ; they simply are a property of some sort. Once again, to see what properties are available to an object, press tab after the dot. Try it out

```
x = 2
x.
x.denominator
```

In [ ]: #

Out[ ]: 1

## 1.2 Lists, Tuples, and Dictionaries

To make data much more useful than simple scalar variables, programming languages make use of sets of data, traditionally called *arrays*. We'll be looking at a variety of types of these, eventually getting to NumPy arrays and pandas.DataFrame, where we'll work with data like a database. We'll start with looking at the most common set type, the **list**, and then immutable lists called **tuples** and look-up systems called **dictionaries**. These will all serve important purposes in working with data of various types, and especially the *spatial data* that we're most concerned about.

```
In [ ]: from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

### list

A list is simply a set of objects, which might be entered as numeric, Boolean or string constants.

Syntax: To create a list, you must use brackets [] :

- emptyList = [] creates list variable emptyList, which is currently empty
- aList = [5, 7.83, "Fred"] creates a list combining different types of things

Run the above code

In [ ]: #

Out[ ]: [5, 7.83, 'Fred']

You can also populate a list with previously assigned variables or other objects like lists.

Enter some new variables and combine these and some constants into a list:

```
x = 5
msg = "Hello"
path = "c:/py/data"
anotherList = [x, msg, path, 2.7, aList]
anotherList
```

In [ ]: #

Out[ ]: [5, 'Hello', 'c:/py/data', 2.7, [5, 7.83, 'Fred']]

In the next code chunk:

- Create a new variable `a` and assign the value `2`.
- Create a new variable `b` and set its value with the expression: `a + 3`
- Create a new variable `wspath` and set its value to the path to the folder you're working in.

Enter some new variables and combine these and some constants into a list:

In [ ]: #

```
2
5
c:\data\exer
```

### Subsetting lists:

You can pull out a subset of a list (sometimes called *slicing*) by using list indices defining the position or positions of list items that you want to extract. This takes a bit of getting used to, but the key is to know that the index refers to a position between each element in a list, as shown here.

```
lyr = ["geology", "landuse", "publands", "streams", "cities"]
      0           1           2           3           4           5
```

You can use a single index value to refer to a single item, starting at that position, as shown here:

`lyr[1]` starts at beginning of "landuse" , so it refers to "landuse"

In [ ]: `lyr = ["geology", "landuse", "publands", "streams", "cities"]`  
`lyr[1]`

Out[ ]: 'landuse'

In essence, this is shorthand for what could be given as:

```
In [ ]: lyr[1:2]
```

```
Out[ ]: ['landuse']
```

... which means the same thing, starts at 1 and goes to start of 2

*Look for the ? question mark to answer a question, by editing the markdown cell it's in and entering a new line. Once you're done editing, you can "run" it to format it, just like a code cell.*

? However, there's a subtle but important difference. What is it?

: lyr[1:2] returns a list

■ To get the first two items in the list, use [0:2]

```
In [ ]: lyr[0:2] # starts at `0` and goes to the start of `2`
```

```
Out[ ]: ['geology', 'landuse']
```

■ How would you get the first three items?

```
In [ ]: #
```

```
Out[ ]: ['geology', 'landuse', 'publands']
```

■ You can also identify positions relative to the end of the list: lyr[-2:] starts two indices before the end of the list:

```
In [ ]: #
```

```
Out[ ]: ['streams', 'cities']
```

■ How would you get only the last item?

```
In [ ]: #
```

```
Out[ ]: ['cities']
```

■ How would you get only the first item, as a list of that one item? lyr[:1]

```
In [ ]: #
```

```
Out[ ]: ['geology']
```

If you have a list of one item, to get what that one item contains, you can follow it with another list request:

```
In [ ]: print(lyr[1:2])
print(lyr[1:2][0])
print(lyr[1]==lyr[1:2][0])

['landuse']
landuse
True
```

If you can create an empty list (ie. `myList = []`) or you just want to add more items to your list, you can use the `append` method.

■ We haven't used a method before, but it is something you do to a Python object. The syntax for a method is `object.method()`, as in `Addresses.append(a1)` below:

```
a1 = [1212, "First St", "SF", "CA"]
a2 = [2323, "Second St", "Seattle", "WA"]
a3 = [3434, "Third St", "Denver", "CO"]

Addresses = []
Addresses.append(a1)
Addresses.append(a2)
Addresses.append(a3)

print(Addresses[0])
print(Addresses[1][1])
print(Addresses[2][3])
```

```
In [ ]: #
```

```
[1212, 'First St', 'SF', 'CA']
Second St
CO
```

*Similar to a question, provide an interpretation when you see the `..` symbol*

• Interpret what you get from the last statement.

:

⌚ If address1 above is a list, what is Addresses?

: a \_\_ of \_\_.

💻 Then how would you print the house number from the third record of Addresses?

Addresses[2][0]

In [ ]: #

Out[ ]: 3434

💻 Create and print a list built from your name and semester schedule, with the semester schedule built of lists with prefix (e.g. "GEOG") and course number (e.g. "625").

In [ ]:

## append vs extend

When we used `.append` above, each time we append one item, which we saw was often a list. What if we wanted to simply combine two lists into one longer one? That's what `.extend` does, or its equivalent, adding lists together with a `+` operator which is really simpler so we'll use that:

```
Pacific = ['AK', 'CA', 'OR', 'WA']
Desert = ['AZ', 'NV', 'UT']
Mountain = ['ID', 'MT', 'WY', 'CO', 'NM']
WestStates = Pacific + Desert + Mountain
WestStates
```

In [ ]: #

Out[ ]: ['AK', 'CA', 'OR', 'WA', 'AZ', 'NV', 'UT', 'ID', 'MT', 'WY', 'CO', 'NM']

In [ ]: States = Pacific
States.append(Desert)
States

Out[ ]: ['AK', 'CA', 'OR', 'WA', ['AZ', 'NV', 'UT']]

## Sorting lists

`.sort` sorts a list. If we have a random list...

```
In [ ]: from random import random as rnd
mylist = []
for i in range(20):
    mylist.append(int(rnd()*10))
mylist
```

```
Out[ ]: [1, 1, 7, 3, 7, 8, 2, 8, 3, 0, 9, 6, 4, 1, 3, 1, 1, 8, 8, 4]
```

... the `sort()` method sorts it. Note that it changes the list called since it's a method of the list. Then we can display it:

```
mylist.sort()
mylist
```

```
In [ ]: #
```

```
Out[ ]: [0, 1, 1, 1, 1, 1, 2, 3, 3, 3, 4, 4, 6, 7, 7, 8, 8, 8, 8, 9]
```

## Counting lists

`.count(x)` returns a count of the value x specified.

```
for i in range(10):
    print("frequency of {}: {}".format(i, mylist.count(i)))
```

```
In [ ]: from random import random as rnd
mylist = []
for i in range(20):
    mylist.append(int(rnd()*10))
mylist
for i in range(10):
    print("frequency of {}: {}".format(i, mylist.count(i)))
```

```
Out[ ]: [2, 2, 1, 9, 6, 1, 9, 2, 1, 5, 0, 4, 8, 3, 6, 4, 3, 9, 5, 6]
```

```
frequency of 0: 1
frequency of 1: 3
frequency of 2: 3
frequency of 3: 2
frequency of 4: 2
frequency of 5: 2
frequency of 6: 3
frequency of 7: 0
frequency of 8: 1
frequency of 9: 3
```

- We haven't used loops before, but see if you can interpret what the code above did and how it worked.

:

## Tuples

We've just been using lists, which allow you to append new members to the list. Sometimes if you know you're not going to want to change anything in a list you may want to use an *immutable* collection called a *tuple*. It otherwise works the same as as list but to create one you use parentheses instead of brackets. You can also create them with commas, so the following tuples are identical:

```
In [ ]: mytuple1 = 5, 7, "name", 8
mytuple2 = (5, 7, "name", 8)
print(mytuple1)
print(mytuple2)

(5, 7, 'name', 8)
(5, 7, 'name', 8)
```

So we might use tuples in the above example. While we may want to append to the set of addresses, each individual address could be a tuple. So create a comparable example this way:

```
a1 = (1212, "First St", "SF", "CA")
a2 = (2323, "Second St", "Seattle", "WA")
a3 = (3434, "Third St", "Denver", "CO")
```

```
Addresses = []
Addresses.append(a1)
Addresses.append(a2)
Addresses.append(a3)
print(Addresses)
```

```
In [ ]: #
[(1212, 'First St', 'SF', 'CA'), (2323, 'Second St', 'Seattle', 'WA'), (3434,
'Third St', 'Denver', 'CO')]
```

You may want to note that accessing a part of a tuple requires using square brackets (since using parentheses might be interpreted as a method or function call), like this:

```
Addresses[1][1]
```

```
In [ ]: #
Out[ ]: 'Second St'
```

## Dictionaries

Dictionaries are used when you want to store named data as *key:data* pairs, and uses braces to create. We'll find dictionaries useful when we start working with NumPy arrays and pandas.DataFrame, where we'll want to start organizing our data with variable names or individual object names. A dictionary is a set that is *ordered* (as of Python 3.7), mutable, but *do not allow duplicates*.

Two common purposes of dictionaries for GIS data processing is to create rows (records or observations) and to create columns (fields or variables) in our data. We'll start with create a row.

- To create a dictionary, use braces and key:data pairs:

```
CA = {  
    "name": "California",  
    "capital": "Sacramento",  
    "areakm2": 423970,  
    "population": 39538223  
}  
print(len(CA))  
CA
```

In [ ]: #

4

Out[ ]: {'name': 'California',  
 'capital': 'Sacramento',  
 'areakm2': 423970,  
 'population': 39538223}

- Another example is for a row of climate data from a weather station:

```
GROVELAND = {"ELEVATION": 853,  
             "LATITUDE": 37.8444,  
             "LONGITUDE": -120.2258,  
             "PRECIPITATION": 176.02}
```

GROVELAND

In [ ]: #

Out[ ]: {'ELEVATION': 853,  
 'LATITUDE': 37.8444,  
 'LONGITUDE': -120.2258,  
 'PRECIPITATION': 176.02}



# variables

	A	B	C	D	E	F
1	STATION_NA	ELEVATION	LATITUDE	LONGITUDE	PRECIPITATION	TEMPERATURE
2	GROVELAND 2 CA US	853	37.8444	-120.2258	176.02	6.1
3	CANYON DAM CA US	1390	40.17028	-121.08833	164.08	1.4
4	KERN RIVER PH 3 CA US	824	35.78306	-118.43889	67.06	8.9
5	DONNER MEMORIAL ST PARK CA US	1810	39.3239	-120.2331	167.39	-0.9
6	BOWMAN DAM CA US	1641	39.45389	-120.65556	276.61	2.9
7	GRANT GROVE CA US	2012	36.73944	-118.96306	186.18	1.7
8	LEE Vining CA US	2072	37.9567	-119.1194	71.88	0.4
9	OROVILLE MUNICIPAL AIRPORT CA US	58	39.49	121.61832	137.67	10.3
10	LEMON COVE CA US	156	34.27	-120.24	62.74	11.3
11	CALAVERAS BIG TREES CA US	1431	38.27694	-120.31139	254	2.7
12	GRASS VALLEY NUMBER 2 CA US	732	39.2041	-121.068	229.11	6.9
13	PLACERVILLE CA US	564	38.6955	-120.8244	170.69	9.2
14	THREE RIVER EDISON PH 1 CA US	348	36.465	-118.86194	114.05	9.7
15	GLENNVILLE CA US	957	35.7269	-118.7006	95.25	6.5
16	BLUE CANYON AIRPORT CA US	1608	39.2774	-120.7102	268.22	4.1
17	MINERAL CA US	1486	40.3458	-121.6091	217.93	0.9
18	SUSANVILLE 2 SW CA US	1275	40.4167	-120.6631	45.21	2.4
19	BRIDGEPORT CA US	1972	38.2575	-119.2286	41.4	-2.2
20	MANZANITA LAKE CA US	1753	40.54111	-121.57667	132.84	0.2
21	OROVILLE CA US	52	39.51779	-121.55306	122.7	10.7

- Now we'll create a (short) variable PRECIPITATION with keys as indices, and see how the key works this way:

```
PRECIPITATION = {"GROVELAND":176.02,
                 "LEE VINTING":71.88,
                 "PLACERVILLE":170.69}

PRECIPITATION["PLACERVILLE"]
```

In [ ]: #

Out[ ]: 170.69

- Since dictionaries are mutable, we can add to them this way:

```
PRECIPITATION["BRIDGEPORT"] = 41.4
PRECIPITATION
```

In [ ]: #

Out[ ]: {'GROVELAND': 176.02,
 'LEE VINTING': 71.88,
 'PLACERVILLE': 170.69,
 'BRIDGEPORT': 41.4}

There are several dictionary methods, and a very useful one is to access the keys themselves as a list, which then can be used for various purposes, such as looping through the data.

```
print(PRECIPITATION.keys())
for station in PRECIPITATION.keys():
    print(PRECIPITATION[station])
```

```
In [ ]: PRECIPITATION.keys()
         for station in PRECIPITATION.keys():
             print(PRECIPITATION[station])
```

```
Out[ ]: dict_keys(['GROVELAND', 'LEE VINING', 'PLACERVILLE', 'BRIDGEPORT'])
         176.02
         71.88
         170.69
         41.4
```

```
In [ ]: GROVELAND.keys()
```

```
Out[ ]: dict_keys(['ELEVATION', 'LATITUDE', 'LONGITUDE', 'PRECIPITATION'])
```

We'll be looking at dictionaries more when we get to NumPy arrays and pandas, where some database objects expect to be built with dictionaries.

## 1.3 Mathematical Computation Using Operators

Python provides a variety of common mathematical operators, typical of most programming languages, and many more through modules you import – we'll look at these later. See one of the books recommended above, or the help system (under “numeric types”), to learn more about these. Common operators are + - \* / \*\* (raise to a power), % (used for modulo, remainder from division).

### Arithmetic operators

Check the following arithmetic operators as to what they return, integer or floating point:

**Addition or subtraction:**

```
x = 2 + 3
y = 2. + 3
z = 2 - 3
```

Use `type()` to see what type they are:

```
In [ ]: #
```

```
<class 'int'>  
<class 'float'>  
<class 'int'>
```

You don't really need to assign a variable, since you can just check an *expression* like the following.

```
In [ ]: print(2.+3)
```

```
5.0
```

But by reusing the variable you can avoid testing two different things due to typos, and this is good practice for coding.

```
In [ ]: y = 2. + 3  
print(y)  
type(y)
```

```
5.0
```

```
Out[ ]: float
```

Spaces are typically unnecessary, but test to make sure.

```
z=2-3  
print(z)  
print(type(z))
```

```
In [ ]: #
```

```
-1  
<class 'int'>
```

• Interpretation (Addition and Subtraction):

## Multiply

```
2 * 3 2. * 3
```

For each of these, assign variables, print the result and type, like the following:

```
m = 2 * 3  
print(m)  
print(type(m))
```

```
In [ ]: #
```

```
6.0
<class 'float'>
```

• Interpretation (Multiplication):

## Divide

```
1 / 2
1. / 2
4 / 2
```



```
In [ ]: #
```

```
Out[ ]: 0.5
```

```
In [ ]: #
```

```
Out[ ]: 0.6
```

```
In [ ]: #
```

```
Out[ ]: 2.0
```

• Interpretation (Divide):

## Power

```
5 ** 2
5 ** 2.0
```



```
In [ ]: #
```

```
Out[ ]: 25
```

```
In [ ]: #
```

```
Out[ ]: 25.0
```

• Interpretation (power), including what happened with all integer inputs.

:

## Square root

```
25 ** (1/2)
25 ** (0.5)
```



```
In [ ]: #
```

```
Out[ ]: 5.0
```

## • Interpretation (square root)

:

## Modulo

Occasionally very useful is the remainder from division, called the "modulo". Knowing the remainder from division is great when you have wrapping values, like those of a clock or a compass. You can also use modulo to figure out if one number is divisible by another (modulo will equal 0). In Python, the operator used for modulo -- % -- is unfortunately confusing, but its use is simple, and best seen by example.

```
print("modulus = 10")
for n in range(2, 14, 2):
    print(n, n % 10)    # 10 might be a common repeated value
print("modulus = 360, for compass azimuth (°)")
for n in range(90, 720, 90):
    print(n, n % 360)    # Compass azimuth (°) is a good application of modulus in a
cycle.15
```

```
In [ ]: #
```

```
modulus = 10
2 2
4 4
6 6
8 8
10 0
12 2
modulus = 360, for compass azimuth (°)
90 90
180 180
270 270
360 0
450 90
540 180
630 270
```

## Conversion functions

We'll be looking at many additional functions imported from various modules, but there are some functions that are so commonly needed that they are built in to the core language. Conversion functions of various types are good examples. For example, you often need to convert numbers to other formats, or convert strings to numbers and vice-versa.

`str(5.278)` Converts a number to a string.



In [ ]: #

Out[ ]: '5.278'

`int("4")` Converts a string to an integer number.



In [ ]: #

Out[ ]: 4

`int("4.17")`



?



In [ ]:

`float("4.17")` Converts a string to a floating point number.



In [ ]: #

Out[ ]: 4.17

```
x = 9.53
y = int(x)
print(y)
z = float(y)
```



In [ ]: #

9

⌚ What is different between x and y?

:

⌚ What is different between y and z?

:

⌚ Does int( ) round or truncate (round down)?

:

```
x = "7.25"
int(float(x))
```



In [ ]: #

Out[ ]: 7

⌚ Why did this work ? (In relation to int(x) which raises an error):

:

## 1.4. Character Strings

A string is a sequential set of characters of text, like 'San Francisco' or 'c:/625/pr/hmb/landuse.shp' or '94132'. It helps to think of them as a sequence of characters, a string of characters, with each character having a position starting with zero, *very much like a list*. There are many ways to work with text strings in Python, and many of these come in handy in GIS work, especially when working with datasets, fields, and text field values.

To review: **Functions, Methods, and Properties:** With numerical objects we've been working with **functions** which apply to the numerical object in the form `function(objects)`. A **method** is like a function but it applies to an object that is in a class that includes that method as a capability, and is run as `object.method()`. We'll also see some **properties** which look a bit like methods, with the form `object.property`, and thus have no parameters; they are just properties of the object.

A string object might be a string literal (like "a string" or 'c:/625/pr/hmb/landuse.shp') or a variable that has been assigned a string. Methods are applied by writing them in after a dot following the string object.

The best way to understand strings is to try them in Python, which we'll do next. Key takeaway: *You should clearly understand (1) how a string can be assigned to a variable; (2) how you can use and extract parts of a string variable; and (3) how strings can be manipulated.*

### Some common string methods

Python includes several string manipulation methods. We'll look at:

- `.capitalize()` Capitalize a word (so make the first character a capital)
- `.upper()`
- `.lower()`
- `.isupper()`
- `.islower()`
- `.split()`

Try the following:

```
s = "the science of where"
print(s.capitalize())
print(s.capitalize().isupper())
print(s.upper())
print(s.upper().isupper())
print(s.split(" "))
```



In [ ]:

```
#  
The science of where  
False  
THE SCIENCE OF WHERE  
True  
['the', 'science', 'of', 'where']
```

- Interpret the above. What did you learn from the above?

## String indices

Strings can be dealt with as a list of characters, with the first one having a zero index. A set of string characters can be defined with a format like 1:3, meaning "from the beginning of 1 to the beginning of 3" -- so it doesn't display the character that is at index 3.

```
print(s[0])  
s1 = s[1]  
print(s1)  
print(s[2])  
print(s[-5:])  
print(s[4:11])  
print(s.split(" ")[1])
```



In [ ]:

```
#  
t  
h  
e  
where  
science  
science
```

## String splitting and using len()

As we just saw, strings are indexed like lists, and another similarity is what you get with the `len()` function. The `.split()` method is very useful for breaking apart a string into components based on a separator. Note how `len()` is used here

```
words = s.split(" ")  
print(words[len(words)-1])  
print(len(words[len(words)-1]))
```



In [ ]: #

where  
5

∴ Interpret the above:

## Concatenating strings

In Python, strings can be concatenated using `+`. No spaces are inserted between them however, so you often need to also concatenate spaces:

```
print(words[3] + words[1])
print(words[3] + " " + words[1])
```



In [ ]: #

wherescience
where science

∴ Interpret the above:

## The escape character \

The backslash is an "escape character" which is used for special needs that can't be easily typed. For instance, you can include a single quote alone with `\'`, a new line with `\n` and a tab with `\t`.

```
print('Jerry\'s Kids')
print('\nJerry\'s\nKids')
```



In [ ]: #

∴ Interpret the above:

But what if you want to include a backslash itself, such as in a Windows file path which uses backslashes?

You can create a single backslash by using a double backslash `\\"`, so a path might look like the following.

```
print('d:\\\\work\\\\soil.shp')
```



In [ ]: #

∴ Interpret the above:

## Including quotation marks in a string

It's often useful to be able to include quotation marks in a string, such as when you might specify a variable in a query. Single quotes are paired with other single quotes, and double quotes are paired with other double quotes. Alternatively, you can use the `escape` method `\"` or `\'` to include that quotation mark within a string; or you can use triple quotation marks to do the same thing. The following three examples (and you could come up with more) are equivalent.

```
selstr1 = '"elev" > 1000'  
selstr2 = '\"elev\" > 1000"  
selstr3 = """elev" > 1000"""  
print(selstr1 + "\n" + selstr2 + "\n" + selstr3)
```



In [ ]: #

"elev" > 1000

∴ Interpret the above, and provide other examples:

In [ ]:

## Using positions in strings

The `.find()` method is useful for finding the first occurrence of a substring. Explore Python help on `.find()` to find ways of finding something other than the first occurrence.

```
p = 'd:/work/lu.shp'  
print(p.find('.'))  
print(p[p.find('.'):])  
print(p.split("/"))
```



In [ ]: #

```
10  
.shp  
['d:', 'work', 'lu.shp']
```

∴ Interpret the above:

## Raw strings for dealing with backslashes

You could also convert backslashes to forward slashes, like UNIX uses for filepaths. But if you want to copy and paste a long filepath from Windows, this is a pain.

█ So one solution is to create a *raw string* by prefacing it with an `r` like the following:

```
print(r'd:\work\soil.shp')
```

*Note that a backslash doesn't always produce a special character result; it depends on what follows it. Best practice for filepaths is to always use `r"..."` to create a raw string.*



In [ ]: #

```
Jerry's Kids
```

```
Jerry's  
Kids
```

∴ Interpret the above:

## 1.5 Logical Operators & Boolean Variables

Boolean (true or false) expressions and variables are very useful for testing conditions for conditional processing, a structure we'll get to next. Boolean values are typically created using logical or binary operators. The result of using a logical operator with two values is a Boolean value of true (1) or false (0). In Python 2.4, results display as 'true' or 'false'.

- == equal to (remember that a single = is for assignment)
- < less than
- > greater than
- != not equal to
- <= less than or equal to
- >= greater than or equal to

(there isn't a key on your keyboard to type in something like  $\geq$ ,  $\neq$ , or  $\leq$ ).

Then there are various Boolean operators for combining multiple Boolean values, like

- &, and Boolean AND (both must be true to return true)
- |, or Boolean OR (either can be true to return true)
- not Boolean NOT (reverses true or false, since 1 is True and 0 is False)

```
x = 5
y = 2
print(x > y)
print(not (x > y))
print(x == y)
```



In [ ]: #

```
True
False
False
```

In [ ]: True & False

Out[ ]: False

- Interpret what the above is showing us.

This is a little weird, but what does it show us?

```
print(2*(x>y))
j = (x>y)+(x==y)+(y>x)
print(j)
```



In [ ]: #

```
2
1
```

• Interpret what the above is showing us.

And if you think this is just esoteric, you may be surprised at how useful this is for spatial analysis.

Now for a couple of Boolean operators:

```
print((x>y)|(y>x))
print((x>y)&(y>x))
print((x>y)|(x==y))
print(not (x==y))
print((x>y)|(-1*(x==y))==True)
```



In [ ]: #

```
True
False
True
True
True
```

• Interpret what the above is showing us.

## key

- This directs you to do something specific, maybe in the operating system or answer something conceptual.
- █ Coding you need to do, in the subsequent code cell.
- ⌚ Questions to answer in the same markdown cell.
- Similar to a question, but requesting an interpretation you need to provide, in the same markdown cell

## 2.1 Working with modules

Python has a reasonable set of built-in methods for common programming tasks, but relies substantially on modules for methods. Python comes installed with a large number of modules, which you can find described online. Some of the more useful modules are math, sys, random, array and os.path.

There are also many modules you can download -- for example numeric processing, such as numpy -- but to find the latest, do searches at [www.python.org](http://www.python.org) or google.

To use a given module, it must be imported first. You do a lot of importing modules in Python. Normally you would put a line import at the top of your program. For instance:

```
import sys
```

would occur at the top of your program before you use any sys methods. It doesn't have to be the first line, just before you use it. If you are entering commands through the console, then just do the import before you need to use it.

Multiple modules, separated by commas, can be entered at one time, such as:

```
import sys, string, math, os, arcpy
```

For now, we'll explore some built-in modules.

► First, just run the following boilerplate code to allow multiple outputs from a cell:

```
In [ ]: from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

### math and random modules

Many common mathematical functions are accessed via the math module. For instance, you would need this for trigonometric functions or logarithms. (To use complex numbers, use cmath which has similar functions, but allows for complex number results.)

Try the following statements in the next code cell:

```
import math
print(math.log10(100))
```

Note that all module functions are prefaced with the module name. This allows for re-use of function names in different modules.



In [ ]:

#

2.0

```
print(math.log(100))
```

In [ ]:

#

4.605170185988092

⌚ How is the above different from the previous? :

```
print(math.pi)
```

In [ ]:

#

3.141592653589793

⌚ Other than what it returns how is `math.pi` different from `math.log()`? What do the parentheses indicate? :

```
pi = math.pi print(pi) pi
```

In [ ]:

#

3.141592653589793

Out[ ]: 3.141592653589793

⌚ What does the above tell you about the print statement for variables? :

Note: the following cells relate to trigonometry, and they expect you to remember the types of units used for measuring angles and how basic trigonometric functions work. You may want to review these.

```
sin = math.sin; cos = math.cos; tan = math.tan  
sin(0)  
cos(0)  
sin(pi)  
cos(pi)
```



```
In [ ]: #
```

Out[ ]: 0.0

Out[ ]: 1.0

Out[ ]: 1.2246467991473532e-16

Out[ ]: -1.0

• There are several things to learn from the above code.

- What did the first line do?

:

- Look carefully at the output of `sin(pi)`. It's in *computerese* scientific notation where the `e-16` is giving us the power of 10, so this one should be something like  $1.2246467991473532 \times 10^{-16}$  (and knowing what value to expect from  $\sin(\pi)$ ) what value does this really represent?

:

Let's consider angle unit conversions.

```
asin = math.asin; acos = math.acos; atan2 = math.atan2
sin(math.radians(45))
sin(45/180*pi)
math.degrees(asin(0.5))
asin(0.5)/pi*180
```

■

```
In [ ]: #
```

Out[ ]: 0.7071067811865476

Out[ ]: 0.7071067811865476

Out[ ]: 30.00000000000004

Out[ ]: 30.00000000000004

• Interpret the results of the above code in terms of angle unit conversions:

:

For a module like `math`, the built-in help system may be useful for finding some quick information. Entering `help(math)` may help.

■

In [ ]:

- Use the above to find help on something useful you weren't previously aware of in the math module.

:

Consider the following and envision how it might be useful in working with spatial problems:

```
x0 = 14; y0 = 8
x1 = 17; y1 = 12
dx = x1-x0; dy = y1-y0
h = (dx**2 + dy**2)**0.5
h
math.hypot(dx,dy)
```



In [ ]: #

Out[ ]: 5.0

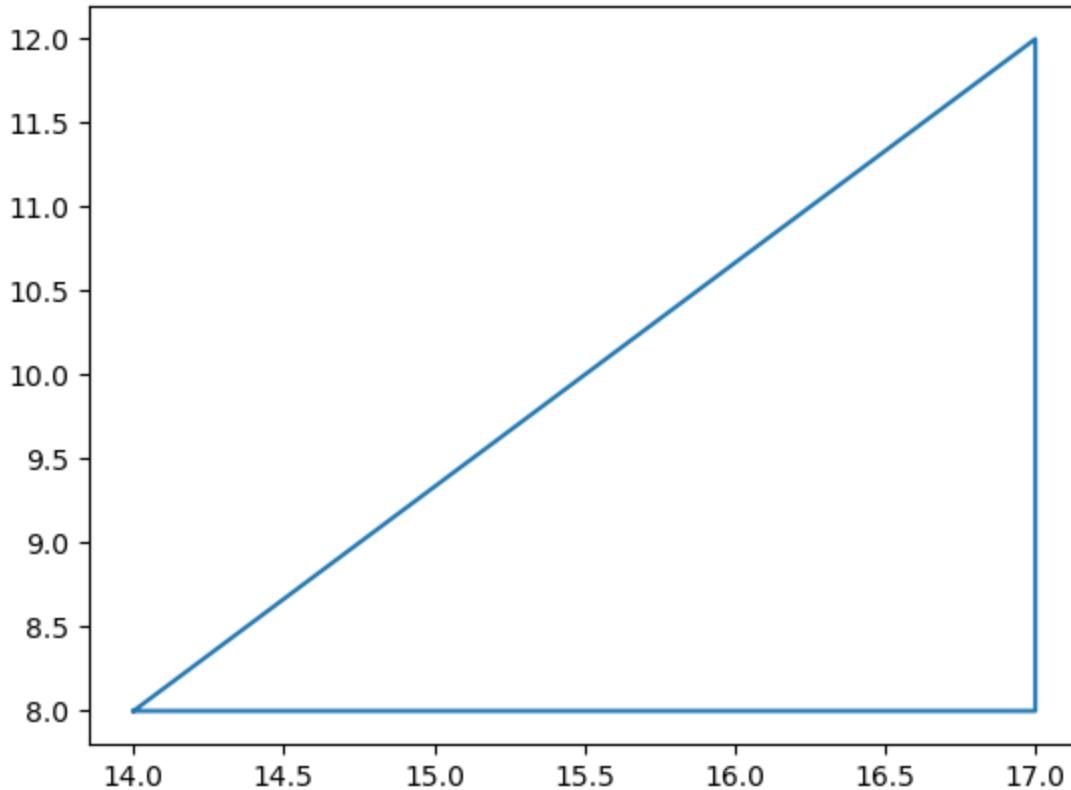
Out[ ]: 5.0

- Interpret what the above is showing us. :

The next graph using these data may help.

```
In [ ]: from matplotlib import pyplot  
pyplot.plot([14,17,17,14],[8,8,12,8])
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1ae33a52640>]
```



```
import random  
print(random.random())  
rnd = random.random  
for i in range(5):  
    print(rnd())
```



```
In [ ]: #
```

```
0.5078208875118928  
0.33065927005499385  
0.33767419682200284  
0.976511861465647  
0.4266331853676444  
0.007580517526020292
```

- Interpret what the above is showing us.

How about integers?

```
rndi = random.randint
for i in range(5):
    print(rndi(0,100))
for i in range(5):
    print(int(rndi()*100))
```



In [ ]: #

```
57
79
19
22
60
11
6
24
44
82
```

∴ Interpret what the above is showing us.

How about a normal distribution?

```
mu = 50
s = 10
for i in range(10):
    print(random.gauss(mu, s))
```



In [ ]: #

```
40.081730002313
58.860089693839356
36.97461795602848
65.74460501389359
42.35850155701855
61.27908385468605
37.605217033380484
38.59688917701296
25.039607376019987
74.712744232311
```

## Sorting and frequency counting some random numbers



```
from random import random as rnd
mylist = []
for i in range(20):
    mylist.append(int(rnd()*10))
mylist
mylist.sort()
mylist
for i in range(10):
    print("frequency of {}: {}".format(i, mylist.count(i)))
```

In [ ]: #

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

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

```
frequency of 0: 1
frequency of 1: 1
frequency of 2: 3
frequency of 3: 2
frequency of 4: 2
frequency of 5: 3
frequency of 6: 4
frequency of 7: 2
frequency of 8: 2
frequency of 9: 0
```

- Interpret what the above is showing us.

:

In [ ]:

### key

- ➡ This directs you to do something specific, maybe in the operating system or answer something conceptual.
- 💻 Coding you need to do, in the subsequent code cell.
- ❓ Questions to answer in the same markdown cell.
- Similar to a question, but requesting an interpretation you need to provide, in the same markdown cell

## 2.2 Flow Control Structures: if, while, for

An important feature of any scripting or programming language is the ability to execute a set of statements as a group, but under controlled conditions. From here on out, we'll want to write scripts and save them into a **py** folder maybe in "pr". For each script use Ctrl-n or File New to create a new script. Maximize your IDE screen so you can see both the script and the console (where outputs go). Then after composing it, save it to your **py** folder, and run the script with the run button. In some cases, I'll suggest names for your scripts; otherwise make one up.

There are three of these control flow operations:

operation	when to operate
if	Only execute the statements if a particular condition is true
while	Keep executing (loop through) the set multiple times while a condition is true
for	Loop through the set for each value in a range of values

These follow Python's clear syntax rules:

- A colon is at the end of the initial statement.
- The block of statements to be processed is indented.

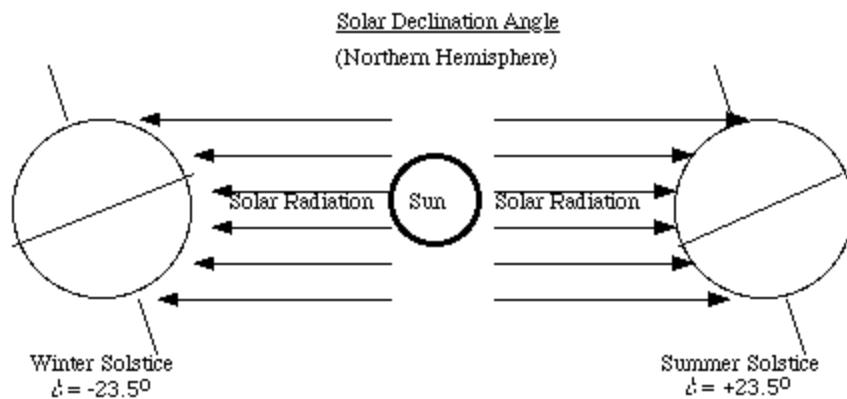
Some important commonalities to these structures:

The if, while, and for statements all end in a colon, followed by an indented block of statements to be used under the conditions defined by the if, while or for. In the code cell, you'll note that after you enter the line with a colon, the next line will be indented. In the line following the code you want to run, backspace to get back to an unindented section.

**if**

**Scenario:** You would like to create a series of hillshade rasters to represent summer, winter and equinox conditions. The hillshade tool requires inputs of **sun angle** (the sun's maximum altitude in the sky on a given day) and azimuth. The sun angle depends on the solar declination. You could look up values in a table, but why not have the computer derive these from what you know? We'll start with a somewhat informed situation -- we know the values for solar declination for four significant dates during the year:

significant date	solar declination
June solstice (June 21)	23.44
Equinox (Mar 21, Sept 21)	0
December solstice (Dec. 21)	-23.44



- Enter the following code that derives sun angle and azimuth from solar declination (the latitude where the sun's rays are vertical at noon) and latitude. It starts with information to populate two variables, `lat` (latitude for the area of study, negative if south of the equator) and `decl` (solar declination), and from these derives `sunangle` and `azimuth`:

```
lat = 30
decl = 20
sunangle = 90 - lat + decl
azimuth = 180
if sunangle > 90:
    sunangle = 180 - sunangle
    azimuth = 0
print(f"Noon sun angle {sunangle}, at azimuth {azimuth}")
```

(Note that we used an alternative formatted printing method, with `f"..."` ), working much the same as `"...".format()` but perhaps a bit easier to read in code, since the variable names appear where you use them.)

In [ ]: #

Noon sun angle 80, at azimuth 180

For now, we've hard-coded the inputs of `lat` and `decl`. In this case `sunangle` would be assigned the value 80 since  $90 - 30 + 20$  is evaluated as 80. The next line assigns 180 to the variable `azimuth`. There's then a section of statements that assigns new values to `sunangle` and `azimuth` *if* `sunangle` ends up with value greater than 90 after the first two lines are processed. Sun azimuth is either from the south (180) or from the north (0) at solar noon.

Note the formatting of the *if* structure:

- starts with `if` followed by a Boolean expression (`sunangle > 90`) as a *condition* followed by a colon :
- the code that will run *if* the condition is true follows as an indented series of statements
- the next indented code continues the program code: it runs whether or not the if structure runs (as long as there's no error raised)

Here's a simple if structure that just prints out what it's doing:

```
print('\n\nStart of script...')  
x = 5  
if x > 0:  
    print('In the "if" block, since x > 0 ...')  
    print('Still in the indented "if" block ...')  
print("Not indented -- we're at the next step in the script.")
```

In [ ]: #

```
Start of script...  
In the "if" block, since x > 0 ...  
Still in the indented "if" block ...  
Not indented -- we're at the next step in the script.
```

Change the hard-coded `x` assignment to not run the `if` block

In [ ]: #

```
Start of script...  
Not indented -- we're at the next step in the script.
```

## Using `if` with files and folders

In GISci, we often need to work with data, so we'll explore some simple methods of accessing data using relevant file paths, which is one application of flow control structures like `if`.

We just looked at using `if` to demonstrate running blocks of indented code using a condition. In the following, we'll also explore another handy module, `os.path`.

► We'll start by using the operating system to set things up:

- Create a `data` folder in the folder where your jupyter files are being saved.
- Then in the `data` folder create a text file "test.txt". *You might want to make sure your folder is not hiding file extensions (tools/folder options/view tab) so you don't create "test.txt.txt".*

▀ Try the following code. Make sure to indent the print statement.

```
import os.path
if os.path.exists("data"):
    print("data folder exists")
```

In [ ]: #

```
data folder exists
```

⌚ Did the data folder exist? If not, did you remember to create it first?

In our code, we've made use of *relative paths* which is a good practice since it makes your code more portable. However, it's common to need to access *absolute paths*, sometimes on a connected server, external drive, or even on the same computer but in a different location. So we need to know how to work with absolute paths.

► Use your OS to find the path to the `data` folder we just created. If you're not sure how to do this one way in Windows at least is to click in the path area of the file explorer window until it changes to display the path, looking something like mine: `C:\py\ex01\data`.

▀ Alter your code to read as follows, *replacing my path with yours*. Note the use of backslashes in the path:

```
import os.path
if os.path.exists("C:\py\ex01\data"):
    print("data folder exists")
else:
    print("data folder doesn't exist")
```

In [ ]: #

```
data folder doesn't exist
```

Interestingly, the code works fine, but this may be because the interpreter in the IDE is fixing things. You may find that backslashes don't work in some IDEs since a backslash is interpreted as an "escape" character, with "\t" representing a tab, etc., so I'm used to prefacing the path string with an `r`, so the path above would be `r"C:\py\ex01\data"`.

▀ You can also have other conditions to test if the first condition isn't met, using the `elif` statement. Note the use of indentation.

```
import os.path
if os.path.exists(r"C:\py\ex01\data\test.txt"):
    print("Test folder exists.")
    print("Text file exists.")
elif os.path.exists(r"C:\py\ex01\data"):
    print("Test folder exists")
    print("... but text file doesn't.")
else:
    print("Neither exist.")
```

In [ ]: #

Neither exist.

## while

▀ Try the following code, which illustrates a `while` loop:

```
i = 1
while i < 10:
    print(i)
    i = i + 1
```

In [ ]: #

1  
2  
3  
4  
5  
6  
7  
8  
9

The last bit of code employs a variable as a "counter" to keep track of how many times we've gone through the loop. Looking at the last statement we can clearly see that it represents an assignment, not a statement of equality `x` can never be equal to `x + 1`, right? But we can assign a new value of `x` to be one greater than its previous value. This is a very basic but extremely important concept in programming; if you don't feel comfortable with this, ask for help before you continue.

Note: The expression `i < 10` can be evaluated as `True` or `False`, so this allows us to process a set of code after evaluating a condition. As we'll see, there are many situations where we will want to use conditional code -- a type of low-level decision that illustrates why we use computers

Try this code which is similar to the last:

```
i = 1
while i < 10:
    print(i)
    i += 1
```

The line `i += 1` is just shorthand for `i = i + 1`.

Note: an interesting experiment would be to un-indent the last line so it's not in the loop. But if you run it, you'll want to interrupt the kernel using the black square, since you've created an infinite loop.

One advantage of the while loop is it lets us skip the whole section if the condition isn't met to begin with. It's even tempting to use it instead of an if statement, but this is an easy way to get into an endless loop: the while loop will keep repeating until the condition is false. When looping through datasets, a common task in GIS work, the `while` structure is often useful. We'll see some examples when we get to using the geoprocessor.

In [ ]: #

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

## for

Try the following code, which illustrates a for loop:

```
for x in [1, 2, 3, 4]:
    msg = "Hello World"
    print(str(x) + " " + msg)
```

In [ ]: #

```
1 Hello World  
2 Hello World  
3 Hello World  
4 Hello World
```

Replace the list [1, 2, 3, 4] with range(4) .

This will also run it four times, but how does it differ?

In [ ]: #

```
0 Hello World  
1 Hello World  
2 Hello World  
3 Hello World
```

Note the use of lists and the len function in the following.

```
featclasses = ["geology", "landuse", "publands"]  
flds = ["TYPE-ID", "LU-CODE", "PUBCODE"]  
for f in featclasses: print(f)  
for j in range(len(featclasses)):  
    print(j, featclasses[j], flds[j])
```

What values do you get for j and why?

:

In [ ]: #

```
geology  
landuse  
publands  
0 geology TYPE-ID  
1 landuse LU-CODE  
2 publands PUBCODE
```

A very useful task for programming is to do something to every file (of a given type) in a folder. We might want to reproject or clip every shapefile in a folder, for instance. We can use the `os` package to look at file names and identify the files with its extension, like `.shp` for instance.

► Go to your `data` folder and create some more text files, all ending with `.txt`. It doesn't matter what's in them; we're just going to use them to provide a list.

▀ Then use this code to list the names of the text files. You can imagine that we could then do something with each file, but for now we're going to list them.

```
import os
print(os.getcwd())
ws = "data"
ilist=os.listdir(ws)
txtfiles = [] # Start with an empty list
for i in ilist:
    if i.endswith(".txt"):
        txtfiles.append(i)
txtfiles
for f in txtfiles:
    print(f)
```

In [ ]:

```
#
```

```
c:\Users\900008452\Box\course\625\exer\Ex02_LogicFlowIO
```

Out[ ]:

```
['untitled.txt']
```

```
untitled.txt
```

*Optional:* If you have some shapefiles (including the various files that go with them), copy them into the `data` folder and modify the code to list them. There's just one simple change to make in the code.

## Combining a for and an if

- ▀ Use the sunangle code to complete the following, producing a list of monthly sun angles and azimuths from the declinations provided (each at about the 21st of the month, starting with Dec 21), starting with empty `sunangles` and `azimuths` lists and appending the values you derive.

```
lat = 37
decls = [-23.44, -20, -12, 0, 12, 20, 23.44, 20, 12, 0, -12, -20]
```

.... code to complete, ending with the following list outputs ...

```
decls
sunangles
azimuths
```

```
In [ ]: #
Out[ ]: [-23.44, -20, -12, 0, 12, 20, 23.44, 20, 12, 0, -12, -20]
Out[ ]: [29.56, 33, 41, 53, 65, 73, 76.44, 73, 65, 53, 41, 33]
Out[ ]: [180, 180, 180, 180, 180, 180, 180, 180, 180, 180, 180]
```

Debug trick: toggling comments... A useful method to try in the code editor is toggling on and off commenting on a section of code: just select anywhere in a single or multiple lines of code and press `Ctrl- /`. That toggles it on or off. This is very handy for checking variables before running something that uses them. For the above code, try this out by commenting out the two `.append` lines and inserting lines that just print the sunangle and azimuth. You can then toggle on and off those print statements as well. So far, our code isn't very complicated, but if you practice this, you'll find it really helpful as things get more challenging...

## key

- This directs you to do something specific, maybe in the operating system or answer something conceptual.
- ▀ Coding you need to do, in the subsequent code cell.
- \_QMARK Questions to answer in the same markdown cell.
- Similar to a question, but requesting an interpretation you need to provide, in the same markdown cell

## 2.3 Creating functions with def

Somewhat similar to a module, but much simpler, is a function you define in your code to be used simply by using the function name later in your code, and providing any parameters you want to use. The function can thus be used as a variable in your program, by *calling* the function and *returning* the value specified by the `return` command line in the function definition. This is easier to understand with an example, which creates a `distance()` function.

```
In [ ]: # boilerplate
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

The following code defines a `distance()` function that uses the Pythagorean theorem to calculate the straight line distance between any two UTM coordinates, which we'll provide as points `a` and `b`.

```
def distance(pt0, pt1):
    import math
    dx = pt1[0] - pt0[0]
    dy = pt1[1] - pt0[1]
    return math.sqrt(dx**2 + dy**2)
```

The function name is `distance` takes in two parameters, each of which is assumed to be a list or tuple representing a coordinate pair of `[x,y]` or `(x,y)`.

Note the importance of the `return` command: In the R language, the value returned is simply the expression in the defined function, so the last line would simply read `dist`, but Python requires you explicitly identify it with the `return` command line.

Note that the `def` structure is similar to the flow control structures we just looked at: with a colon on the `def` line and the lines of code indented.

```
In [ ]: #
```

Now create the inputs as two points `a` and `b` entered as hard-coded tuples, and process them with the new function:

```
a = (520382, 4152373); b = (520782, 4152673)
distance(a, b)
```

```
In [ ]: #
```

```
Out[ ]: 500.0
```

Then try the same thing, creating `c` and `d`, but create them as lists instead of tuples. Or with one a tuple, the other a list; shouldn't matter.

In [ ]: #

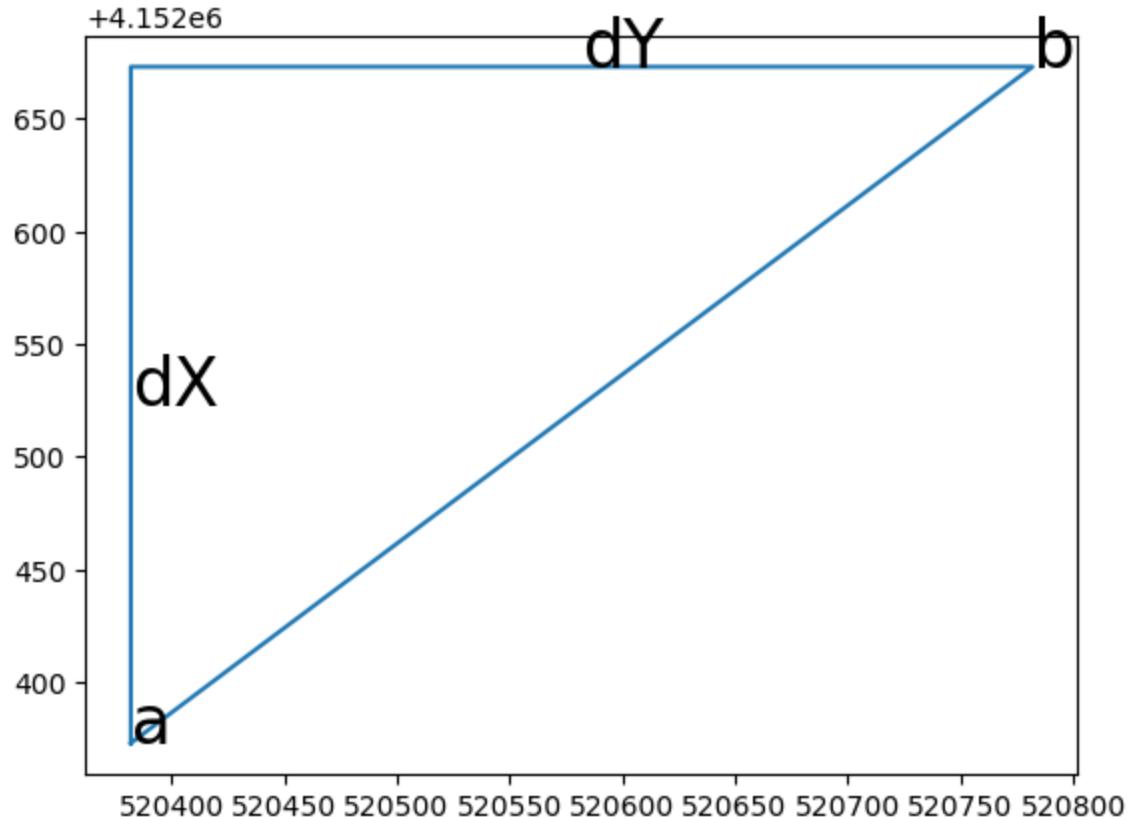
Out[ ]: 500.0

A graph may help, and we'll jump ahead and use a little numpy and matplotlib. Feel free to copy and paste this into the next code cell, since we haven't learned about numpy and matplotlib yet, so this isn't the time to figure it out yet (unless you're really anxious...)

```
import matplotlib
import numpy as np
from matplotlib import pyplot as plt
x = np.array([a[0],a[0],b[0],a[0]])
y = np.array([a[1],b[1],b[1],a[1]])
fig, ax = plt.subplots()
plt.plot(x,y)
ax.axis('equal')
plt.text(*a,"a",size=24)
plt.text(*b,"b",size=24)
dXlab = (a[0],(b[1]-a[1])/2+a[1])
dYlab = ((b[0]-a[0])/2+a[0],b[1])
plt.text(*dXlab,"dX",size=24)
plt.text(*dYlab,"dY",size=24)
```

```
In [ ]: #
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1ae33ac6e80>]  
Out[ ]: (520362.0, 520802.0, 4152358.0, 4152688.0)  
Out[ ]: Text(520382, 4152373, 'a')  
Out[ ]: Text(520782, 4152673, 'b')  
Out[ ]: Text(520382, 4152523.0, 'dX')  
Out[ ]: Text(520582.0, 4152673, 'dY')
```



```
distance(pt1=(520782, 4152673), pt0=(520382, 4152373))
```

```
In [ ]: #
```

```
Out[ ]: 500.0
```

Modify the script to also return the angle in degrees between the two points. Note that we are using azimuth type angles, with the  $0^\circ$  to the north and going around clockwise. rules for trigonometry-type Rename the function to be `distangle` and return as a *tuple* with return dist, angle. (Hint: use `math.degrees` and `math.atan2(dx,dy)`)

Note that we are using azimuth-type angles, used for mapping, with the  $0^\circ$  to the north and going around clockwise, in contrast to the way you would have learned it in math classes with  $0^\circ$  to the right and going around counter-clockwise. That's why we're specifying `(dx,dy)` instead of `(dy,dx)` as we would in standard use in math classes. The azimuth in the triangle above is the angle at  $a$ .

In [ ]: #

Test it out a couple of ways:

```
distangle(a,b)
distangle(pt1=(520782, 4152673), pt0=(520382, 4152373))
```

In [ ]: #

Out[ ]: (500.0, 53.13010235415598)

In [ ]: #

Out[ ]: (500.0, 53.13010235415598)

Change the parameters for the `distangle()` call to look instead like:

```
distangle(pt0=a, pt1=b)
distangle(pt1=b, pt0=a)
distangle(pt1=a, pt0=b)
```

In [ ]: #

Out[ ]: (500.0, 53.13010235415598)

Out[ ]: (500.0, 53.13010235415598)

Out[ ]: (500.0, -126.86989764584402)

Note the difference in order! Function parameters have a particular order, but if we name them we can provide them in a different order.

## Unpacking a list or tuple

Have a look at the script above where we used matplotlib to make a graph. (We'll be learning about matplotlib in the next section.) You'll notice that the text-plotting commands do something unfamiliar with the point tuples `a` and `b` by specifying them as `*a` and `*b` (and similarly for `dXlab` and `dYlab` )

```
plt.text(*a,"a",size=24)
plt.text(*b,"b",size=24)
```

We're seeing *unpacking* which is something we need to use with the pyplot `.text` method which we can see by requesting help on the method something like `help(plt.text)` , which works because we've imported `matplotlib.pyplot` as `plt` . The required first three parameters of the `.text` method are `x`, `y`, `s` for the coordinates followed by the text string to plot. To provide `a` and `b` as `x` , `y` we need to unpack the tuple and that's what `*` does.

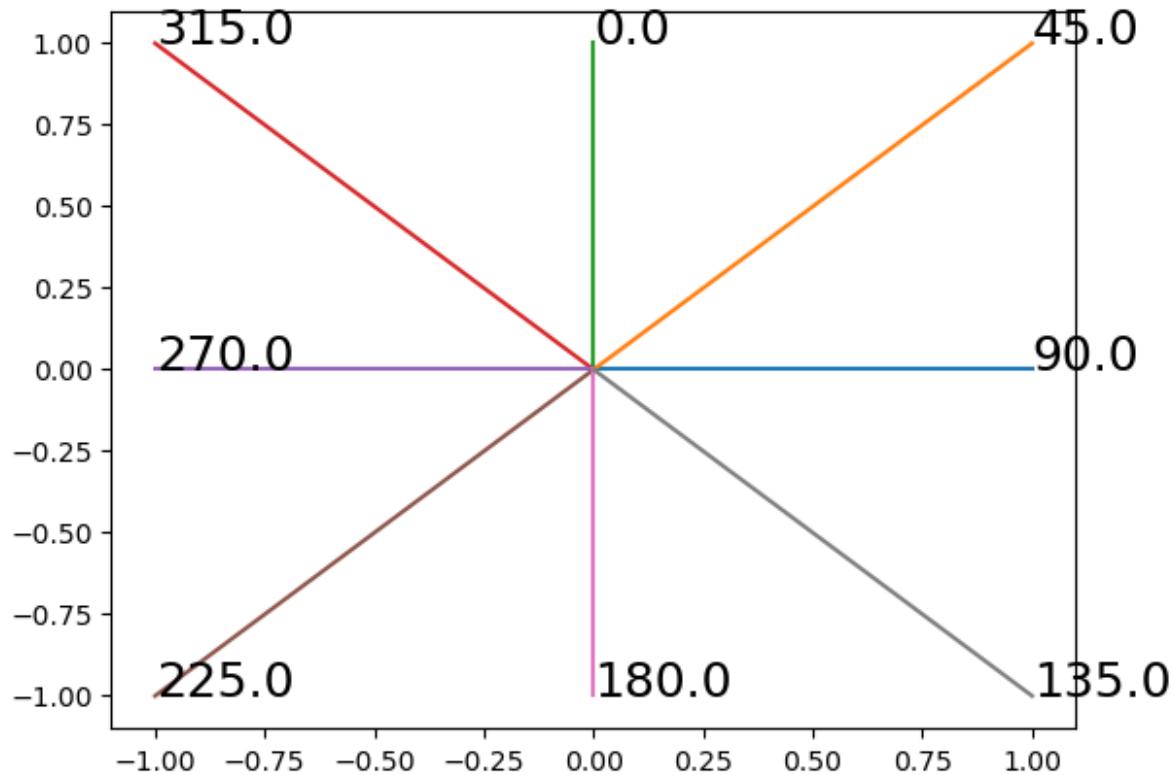
Here's another example of the same thing where we can apply our `distangle()` function to add text showing angles in degrees to a plot, and unpack the `pt` tuple.

Again, feel free to just copy and paste this into the cell; we'll learn this stuff next week.

```
import matplotlib
import numpy as np
from matplotlib import pyplot as plt
origin = (0,0)
for pt in [(1,0),(1,1),(0,1),(-1,1),(-1,0),(-1,-1),(0,-1),(1,-1)]:
    dist, angle = distangle(origin,pt)
    x = np.array([origin[0],pt[0]])
    y = np.array([origin[1],pt[1]])
    plt.plot(x,y)
    plt.text(*pt, str(angle % 360), size=18)
```

```
In [ ]: #
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1ae33b59c70>]
Out[ ]: Text(1, 0, '90.0')
Out[ ]: [<matplotlib.lines.Line2D at 0x1ae33b6b100>]
Out[ ]: Text(1, 1, '45.0')
Out[ ]: [<matplotlib.lines.Line2D at 0x1ae33ab4f70>]
Out[ ]: Text(0, 1, '0.0')
Out[ ]: [<matplotlib.lines.Line2D at 0x1ae33b6b670>]
Out[ ]: Text(-1, 1, '315.0')
Out[ ]: [<matplotlib.lines.Line2D at 0x1ae33b6bb80>]
Out[ ]: Text(-1, 0, '270.0')
Out[ ]: [<matplotlib.lines.Line2D at 0x1ae33b7c070>]
Out[ ]: Text(-1, -1, '225.0')
Out[ ]: [<matplotlib.lines.Line2D at 0x1ae33b7c4f0>]
Out[ ]: Text(0, -1, '180.0')
Out[ ]: [<matplotlib.lines.Line2D at 0x1ae33e14af0>]
Out[ ]: Text(1, -1, '135.0')
```



- In the last line of code above, there are two parameters provided after `*pt`. Describe what's going on with the next parameter and how that relates to what you see on the graph. Maybe experiment with changing it.
- :

- Let's change our functions to use four inputs to demonstrate the need for unpacking.

```
def dist(x1,y1,x2,y2):
    import math
    dx = x2 - x1
    dy = y2 - y1
    return math.sqrt(dx**2 + dy**2)
```

In [ ]: #

- So to send our `a` and `b` points to `dist()`, we'll unpack them:

```
dist(*a, *b)
```

In [ ]: #

Out[ ]: 500.0

## Unpacking a dictionary

Unpacking a dictionary is a little bit more complicated but has an interesting application for functions. It does require that the dictionary use the variable names expected by the function. One advantage of this is in providing inputs in a different order:

Since functions allow inputs to be provided in a different order by including their variable names, such as `dist(x1=0, x2=1, y1=5, y2=5)` working with the defined function where they are expected in a different order:

```
def dist(x1,y1,x2,y2):
```

- The following code illustrates this and shows the input provided by unpacking and the equivalent standard input:

```
Xs = {"x1": 0, "x2": 1}
Ys = {"y1": 5, "y2": 5}
print("dictionary input: {}".format(dist(**Xs, **Ys)))
print("equivalent standard input: {}".format(dist(x1=0, x2=1, y1=5, y2=5)))
```

In [ ]: #

```
dictionary input: 1.0
equivalent standard input: 1.0
```

## key

- This directs you to do something specific, maybe in the operating system or answer something conceptual.
- ▀ Coding you need to do, in the subsequent code cell.
- ⌚ Questions to answer in the same markdown cell.
- Similar to a question, but requesting an interpretation you need to provide, in the same markdown cell

## 2.4 Input/Output

In this section we'll look at input and output methods, starting with user input and output displays, then input and output of files.

### Output display

We'll start with output, since we've already been using a variety of output methods, including expressions in code cells, the print statement and formatted output using ".format" and "f" methods. We'll look at these in more detail. For now, *don't run the boilerplate we've been using, and if you have, then restart the kernel.*

- ▀ Let's start with the snippet of the sunangle code, and then make it more useful. We'll start with our example where we "hard code" data directly into it by assigning variables. We'll print the results using the expression method.

```
lat = 40
decl = 23.44
sunangle = 90 - lat + decl
azimuth = 180
if sunangle > 90:
    sunangle = 180 - sunangle
    azimuth = 0
sunangle
azimuth
```

In [ ]: #

Out[ ]: 73.44

Out[ ]: 180

Note that without the InteractiveShell setting in our boilerplate, *you only get one output from a code cell*, and that will be the last expression in the code cell, so the `azimuth`.

- One solution is to change the multiple expressions to one, but converted to a tuple with: `sunangle`, `azimuth`

```
In [ ]: #
```

```
Out[ ]: (73.44, 180)
```

- Now change it back to two separate expressions, run the boilerplate, and then run the above code again to see the difference.

```
In [ ]: from IPython.core.interactiveshell import InteractiveShell  
InteractiveShell.ast_node_interactivity = "all"
```

```
In [ ]: #
```

```
Out[ ]: 73.44
```

```
Out[ ]: 180
```

This is still pretty minimal in the way of an output, since we have to figure out what expression produced each output, and sometimes an expression output is missing some useful formatting.

- For instance try this code that concatenates two strings and inserts a new line with `\n`, as an expression:

```
"To be or not to be?" + "\nThat is the question."
```

```
In [ ]: #
```

```
Out[ ]: 'To be or not to be?\nThat is the question.'
```

- Now put that expression in a `print()` statement to see the line, we'll see that line formatting:

```
In [ ]: #
```

```
To be or not to be?  
That is the question.
```

## Formatted output

- Going back to our sunangle code, change the expressions to print statements.

In [ ]: #

```
73.44  
180
```

Well, in this case that's really no different, since we're just printing a number, so you might think we could do without that `InteractiveShell` boilerplate and just insert multiple print statements, but we'll find when we get to **pandas** that printing a dataframe isn't as nice looking as just putting the dataframe name as an expression, allowing Jupyter to format it in a different way. But that's later; we'll continue now with formatted output using either the `".format` or `f"` method which we've used a bit before, but need to explore further.

We'll use a numerical format for floating point numbers:

- For noon sun angle we'll specify a number occupying 5 spaces, and 2 decimal places with `:5.2f`
- For azimuth we'll specify 3 spaces with 0 decimal places with `:3.0f`

▀ Try both of these methods that should produce the same result:

```
print("Noon sun angle: {0:5.2f}°, at azimuth {1:3.0f}°".format(sunangle, azimuth))
```

or

```
print(f"Noon sun angle: {sunangle:5.2f}°, at azimuth {azimuth:3.0f}°")
```

In [ ]: #

```
Noon sun angle: 73.44°, at azimuth 180°  
Noon sun angle: 73.44°, at azimuth 180°
```

## Input

Now let's get away from the hard-coded input. We probably don't want to have to alter our program every time we want to run it. There are a variety of ways of providing input. One way is to use the `input` method.

▀ Modify the code for getting the two inputs this way:

```
lat = float(input("Latitude: "))  
decl = float(input("Solar Declination: "))
```

Look for a prompt when you run this for entering Latitude and Solar Declination.

In [ ]: #

```
Noon sun angle: 72.0°, at azimuth 180.0°
```

Try this with different values (Don't type values into the code, but respond to the prompts). Valid values of latitude are -90 to 90, while valid solar declination values are -23.44 to 23.44. Try -70 for latitude and 23.44 for declination to see what the sun angle would be at 70°S on the June solstice. Interpret what the above is showing us.

## Reading and Writing Text Files

A common need is to work with text files, especially CSV (comma separated variable) files that most programming environments (and Excel) work with regularly.

### Writing a text file

In this section, we'll start by hard-coding some data and writing it out to a text file, then we'll read in that text file.

*Note: the first coding problem assumes you have previously created a data folder. It doesn't matter what's in it, but we'll need it to be there to create an output.*

▀ We'll start by simply displaying a set of three points each stored as *tuples* of values (id, name, x, y), in a container list, then just print these out:

```
ptData = [(1,"Trail Jct Cave",483986,4600852),
           (2,"Upper Meadow",483473,4601523),
           (3,"Sky High Camp",485339,4600001)]
for pt in ptData:
    print("{}{},{}{},{}{}".format(pt[0],pt[1],pt[2],pt[3]))
```

In [ ]: #

```
1,Trail Jct Cave,483986,4600852
2,Upper Meadow,483473,4601523
3,Sky High Camp,485339,4600001
```

Writing out a CSV text file is pretty easy using the `csv` module and its methods `.writer` that creates the object that is opened by `with open()`, and `.writerow` for writing the header row of field names, then `.writerows` for writing an iterable list of row tuples. The `newline=''` option is needed to avoid creating extra line ends as carriage returns.

- ▀ The complete script will need the creation of the `ptData` above followed by:

```
import csv
with open("data/marblePts.csv", 'w', newline='') as out:
    csv_out=csv.writer(out)
    csv_out.writerow(['ID','Name','Easting','Northing'])
    csv_out.writerows(ptData)
```

In [ ]: #

Out[ ]: 13

- Check what you get by opening it in Excel. Once you've confirmed you got what you expected, close Excel.

You may have already discovered this, but a common problem with reading and writing data happens when two programs are trying to access the same data, for instance if you have the CSV open in Excel and then try to write it again from Python -- you'll get a message that it's locked, and it can be difficult to fix the problem, often requiring closing out of both Jupyter and Excel, if not something more extreme.

Note the use of the `with ...:` structure. It's handy because it sets up an environment that applies *only within the indented code of the structure*. We'll see it again when working in arcpy to apply environment settings that we only want to use within the structure.

## Reading a text file

We'll just read in the text file we just wrote, a couple of different ways.

- First we'll use the built-in methods.

```
infile = "data/marblePts.csv"
f = open(infile, "r")
firstline = True
for line in f:
    if firstline: firstline=False
    else:
        values = line.split(",")
        id = int(values[0])
        name = values[1]
        x = float(values[2])
        y = float(values[3])
        print("{}{},{}{},{}{}".format(id, name, x, y))
f.close()
```

In [ ]: #

Out[ ]: 'ID,Name,x,y\n'

```
1,Trail Jct Cave,483986.0,4600852.0
2,Upper Meadow,483473.0,4601523.0
3,Sky High Camp,485339.0,4600001.0
```

## Using csv

- The csv module also has a reader method. A simple display of the data could work this way:

```
import csv
with open("data/marblePts.csv", newline="") as csvfile:
    dta = csv.reader(csvfile, delimiter=",")
    for row in dta:
        print(row)
```

In [ ]: #

```
['ID', 'Name', 'x', 'y']
['1', 'Trail Jct Cave', '483986', '4600852']
['2', 'Upper Meadow', '483473', '4601523']
['3', 'Sky High Camp', '485339', '4600001']
```

■ What did the `csv.reader` return: what does each row represent?

:

■ Modify the last line of code to instead display each line as a single string separated by commas with

```
", ".join(row)
```

In [ ]: #

```
Out[ ]: 'ID, Name, x, y'  
Out[ ]: '1, Trail Jct Cave, 483986, 4600852'  
Out[ ]: '2, Upper Meadow, 483473, 4601523'  
Out[ ]: '3, Sky High Camp, 485339, 4600001'
```

■ We might take this a bit further and populate variables:

```
import csv  
with open("data/marblePts.csv", newline="") as csvfile:  
    dta = csv.reader(csvfile, delimiter=",")  
    firstline = True  
    for row in dta:  
        if firstline:  
            row  
            firstline=False  
        else:  
            id = int(row[0])  
            name = row[1]  
            x = float(row[2])  
            y = float(row[3])  
            print("{} , {} , {} , {}" .format(id, name, x, y))
```

In [ ]: #

```
Out[ ]: ['ID', 'Name', 'x', 'y']  
1, Trail Jct Cave, 483986.0, 4600852.0  
2, Upper Meadow, 483473.0, 4601523.0  
3, Sky High Camp, 485339.0, 4600001.0
```

When we get to **pandas** we'll be working with dataframes which are a better way of working with tabular data like this, and we'll learn about methods of reading and writing converting CSVs into dataframes and vice versa. So we don't need to look at these methods much longer. But before we leave, you'll note that the code above already knew the variable names, and how many they are. We might want to detect what those variables are, if they're stored in the first row of the data.

▀ This is a good use for a dictionary, which makes it easy to get variable names from strings we read from somewhere, like the first line of the `data/marblePts.csv` file. There's probably an easier way of doing this, but this works:

```
import csv
with open("data/marblePts.csv", newline="") as csvfile:
    dta = csv.reader(csvfile, delimiter=",")
    firstline = True
    for row in dta:
        if firstline:
            fields = row
            print(f"fields: {fields}")
            firstline=False
            marblePts=[]
            valueTuples=[]
        else:
            for i in range(len(fields)):
                valueTuples.append((fields[i],row[i]))
            marblePts.append(dict(valueTuples))
marblePts
```

In [ ]: #

```
fields: ['ID', 'Name', 'x', 'y']
```

Out[ ]: [{`'ID': '1', 'Name': 'Trail Jct Cave', 'x': '483986', 'y': '4600852'`},  
 {`'ID': '2', 'Name': 'Upper Meadow', 'x': '483473', 'y': '4601523'`},  
 {`'ID': '3', 'Name': 'Sky High Camp', 'x': '485339', 'y': '4600001'`}]

- Interpret some key features in the code above to make it work.

:

- ?
- What's the purpose of the statement `firstline=False` ?

:

- ?
- Why did we `for i in range(len(fields)):` instead of something like `for fld in fields:` which would have also looped through the fields?

:

- █ Provide some code that uses the `marblePts` data created. For instance, print out the `Names`, one per row.

In [ ]: #

```
Trail Jct Cave
Upper Meadow
Sky High Camp
```

## key

- This directs you to do something specific, maybe in the operating system or answer something conceptual.

- █ Coding you need to do, in the subsequent code cell.

- ?
- Questions to answer in the same markdown cell.

- Similar to a question, but requesting an interpretation you need to provide, in the same markdown cell

## 3 Intro to NumPy and Matplotlib

We'll explore these together since Matplotlib gives us a nice way of visualizing our NumPy data, but we'll start by using Matplotlib with list data.

```
In [ ]: # boilerplate
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

### Learning matplotlib

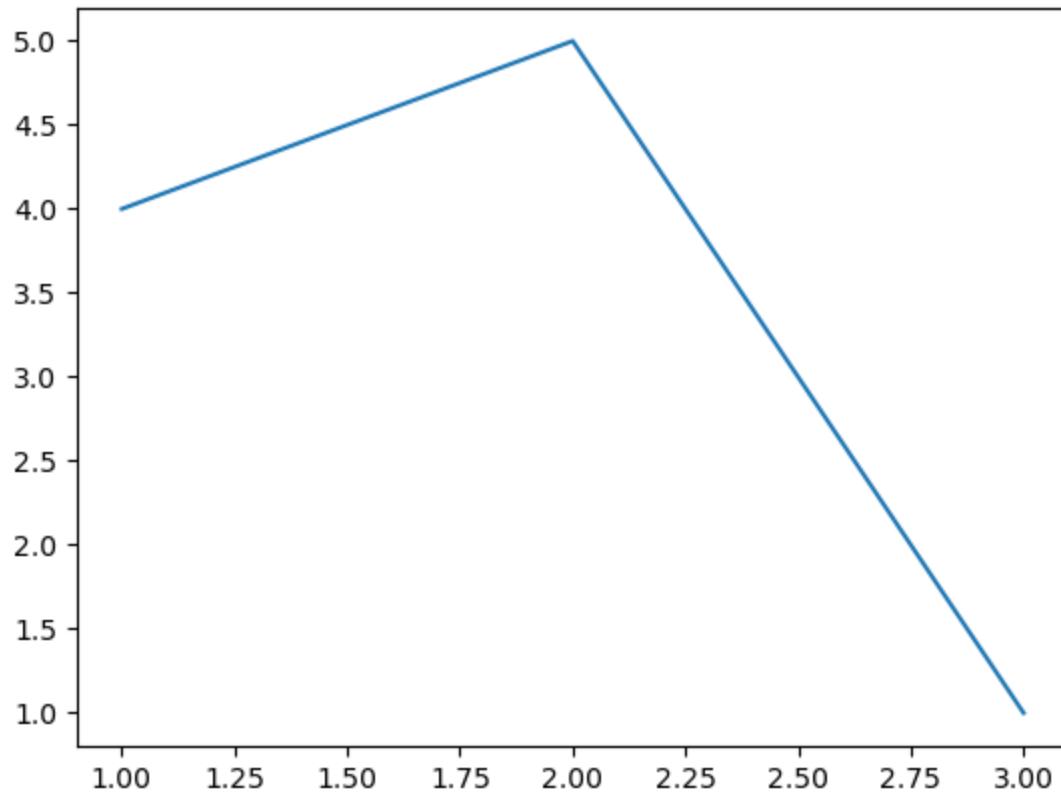
The matplotlib package does a *lot*. You will find that it's pretty much the only graphics system in Python, yet there is an enormous amount of graphical work done with it. Different applications will use customized backends developed in matplotlib, and each of these include specialized routines and ways of working, but all within matplotlib. We will be focusing on just what we need to productively use the package, but you should refer to <http://matplotlib.org> (<http://matplotlib.org>) for a lot more information. Much of the documentation is fairly cryptic, but one quick way of getting a sense of what you can do is to explore examples at <https://matplotlib.org/stable/gallery/index.html> (<https://matplotlib.org/stable/gallery/index.html>) where you can also see the code that generates them.

For our first Matplotlib plot, we'll create a simple line plot from three pairs of coordinates, with x coming from one list and y from the other. We'll import the `matplotlib` module and from it import the `pyplot` interface (similar to MATLAB), which includes the `plot` function that plots a line plot by default.

```
import matplotlib
from matplotlib import pyplot
pyplot.plot([1,2,3],[4,5,1])
```

```
In [ ]: #
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1ecdbb224c0>]
```



A slightly more efficient way to start the plot is to import pyplot as plt

```
from matplotlib import pyplot as plt  
plt.plot([1,2,3],[4,5,1])
```

■ Change your code to read as above, and then add a second line feature to the plt object with:

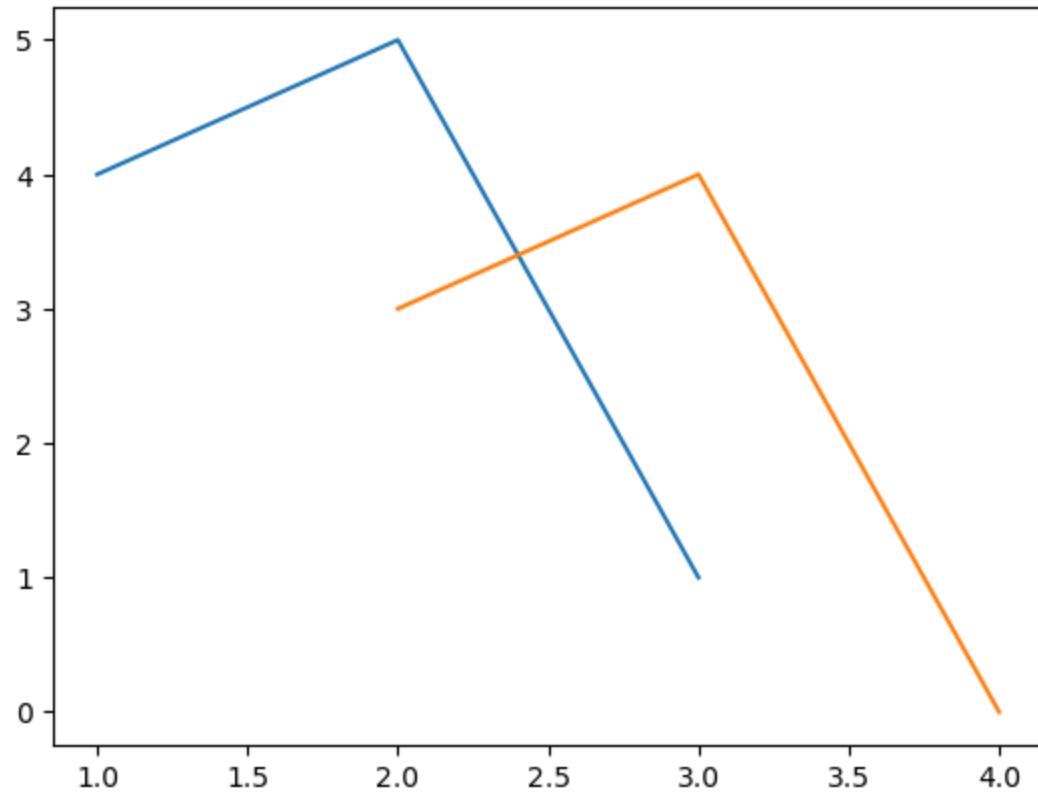
```
plt.plot([2,3,4],[3,4,0])
```

Note that the axes expanded a bit to include the new feature.

```
In [ ]: #
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1ecdbf9a190>]
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1ecdbf9ae50>]
```

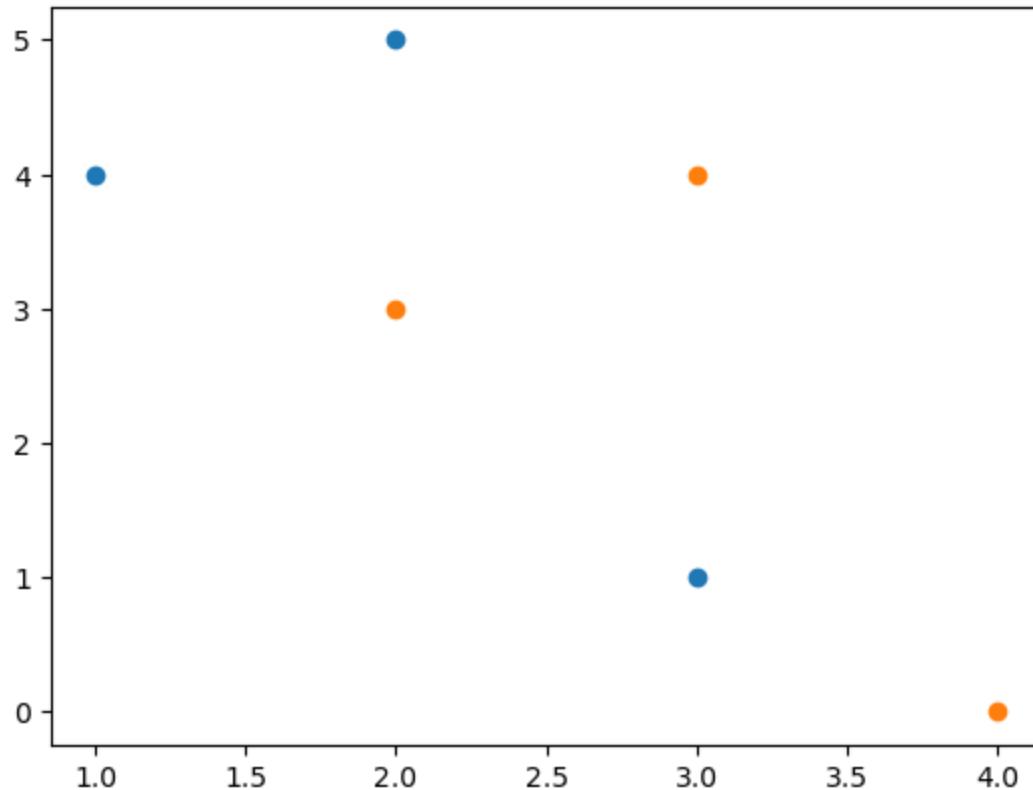


☰ Change .plot to .scatter to create points instead of lines.

```
In [ ]: #
```

```
Out[ ]: <matplotlib.collections.PathCollection at 0x1ecdbbd9130>
```

```
Out[ ]: <matplotlib.collections.PathCollection at 0x1ecdbbd9610>
```

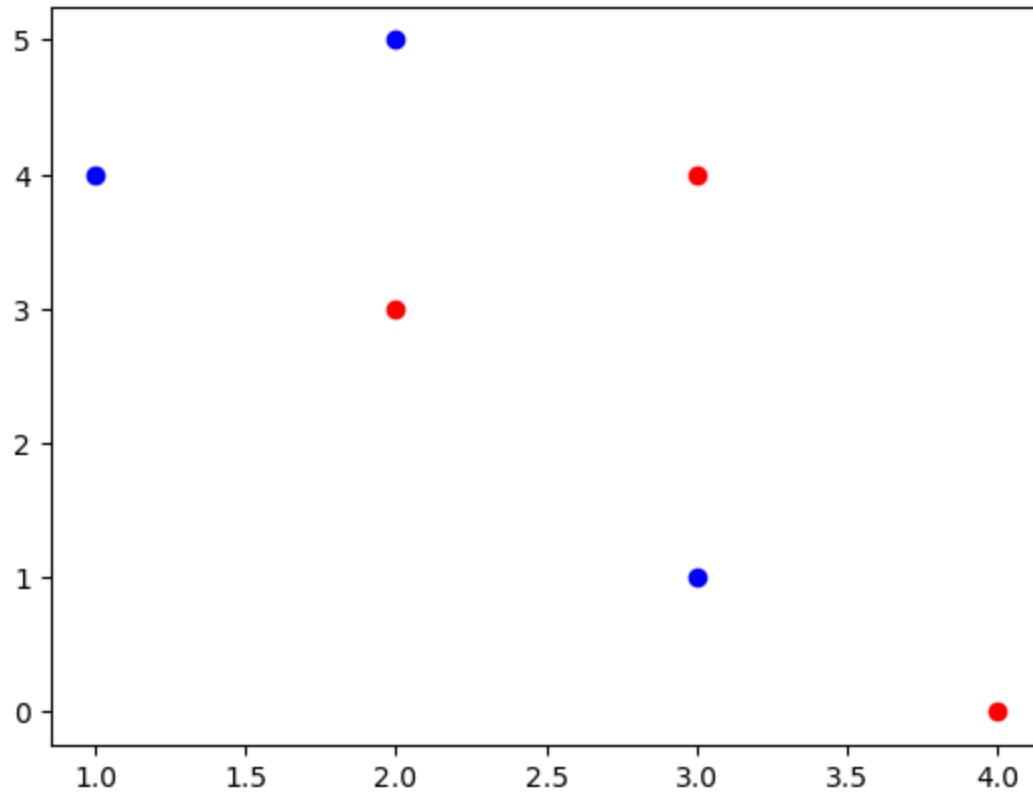


- Now change both methods back to `.plot`, but after the y coordinates list add a third parameter 'bo' to the first to make blue solid circles and to the second 'ro' to make red solid circles.

```
In [ ]: #
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1ecdbf1d070>]
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1ecdbb66040>]
```

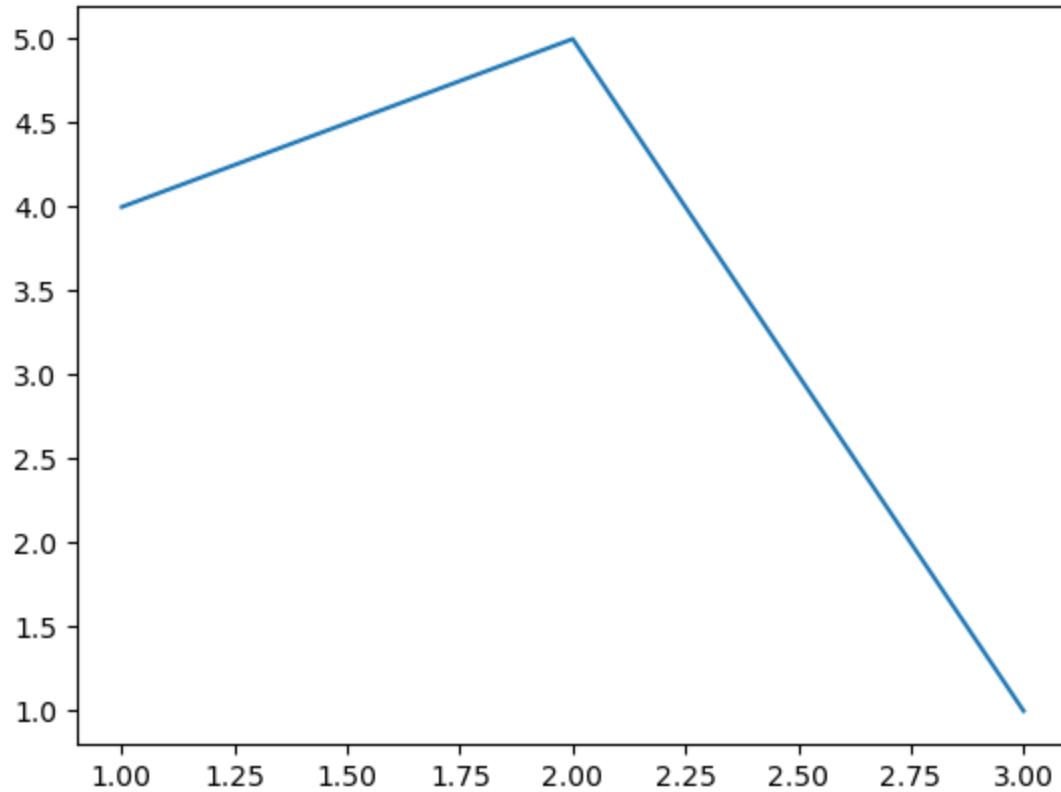


Matplotlib expects numpy arrays as input *or objects that can be converted to such with `numpy.asarray()`* so in the first plot, this was happening behind the scene -- the simple lists were converted to numpy arrays. This is what it looks like more explicitly:

```
import numpy
from matplotlib import pyplot as plt
plt.plot(numpy.asarray([1,2,3]),numpy.asarray([4,5,1]))
```

```
In [ ]: #
```

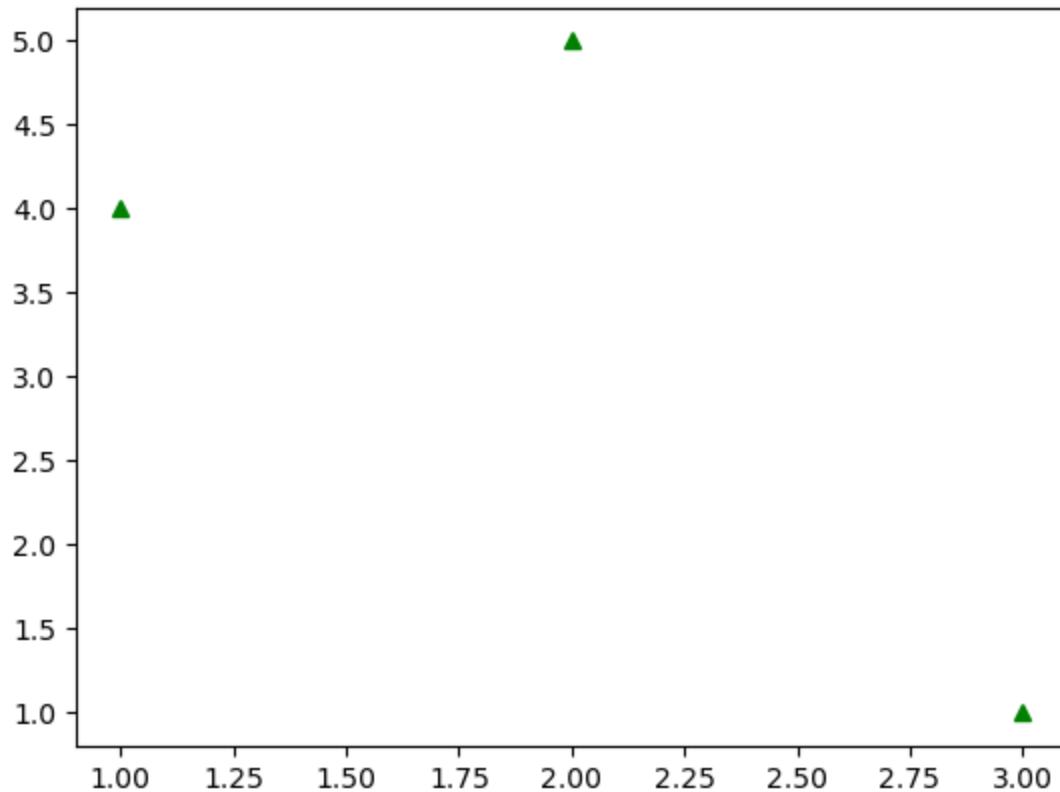
```
Out[ ]: [<matplotlib.lines.Line2D at 0x1eecd74b9a0>]
```



💻 Change the above to create green triangles with 'g^'

```
In [ ]: #
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1eece81c760>]
```



There's a lot to Matplotlib and the pyplot methods.

→ Refer to the cheatsheets and guides at <https://matplotlib.org/cheatsheets/> (<https://matplotlib.org/cheatsheets/>)

## NumPy

Ok, we jumped ahead a bit there, so we should formally introduce NumPy. There's a fair amount you can learn about NumPy, "*the fundamental package for scientific computing in Python*" (<https://numpy.org/doc/stable/user/whatisnumpy.html> (<https://numpy.org/doc/stable/user/whatisnumpy.html>)), but we're only going to need to explore a relatively small part of it.

## Introduction to ndarray objects

These n-dimensional arrays are at the core of NumPy. Here are some of their characteristics:

- They are immutable, having a fixed size when you create them. If you change their size, a new one will actually be created.
- The elements must all be of the same type. You might imagine these to be numerical, but the elements can actually be objects of a complex structure, just each object has the same structure of every other one in the array.
- They facilitate numerical operations, allowing them to execute efficiently, so have been adopted by many applications that use Python (such as ArcGIS) to crunch large data sets.
- Mathematical operations use vectorization methods (similar to R), with element-by-element operations coded simply. Say you have two ndarrays `a` and `b` (or one could be a constant or scalar variable) -- to multiply them simply requires `c = a * b`
- They can have more than one dimension, and the dimensions of ndarrays are called **axes**

The analogous data structure in R is a *vector*, which can also have multiple dimensions, where they're called *matrices* or *arrays*, and similarly all elements must be the same type.

So let's create some:

- ▀ While not required, it's good practice for code readability to standardize on the main numpy object name and call it `np`, so starting our code with `import numpy as np` is what we'll do. We'll start by creating a mixed-type list, then try to build an array out of it.

```
import numpy as np  
mixedList = [1, "x", 5]  
myArray = np.asarray(mixedList)
```

In [ ]: #

- ▀ That worked, but what did we get?

```
myArray
```

In [ ]: #

Out[ ]: array(['1', '3', '5'], dtype='<U11')

Note that the dtype (data type) is <U11, for which "U" refers to the *unicode* character data type, with 11 part just referring to how many characters are in the string (and oddly 11 is the minimum). When I first saw this, I thought the 11 referred to the number of bits, but if you put a string longer than 11 characters in there, you'll find it uses a larger number. A while back, the main character coding system used was ASCII, which stands for "American Standard Code for Information Interchange" and it was a 7-bit code, capable of handling all of the Latin characters ('a':'z','A':'Z') used in the U.S. (thus *American* standard code), Arabic numerals ('0':'9'), and other things typically on your keyboard, and some special needs like line ends and tabs. That was fine for normal computing where computer languages used English anyway, but hampered the use of quite a few other languages with different characters. In the first extension of this to 8 bits, Greek was added first, though mostly for mathematical symbols, and also accented letters like é (which I typed using Alt 130 on the numeric keypad). Unicode expands this to character sets in all kinds of other languages.

But back to *numeric* arrays...

## Creating an ndarray with np.arange()

We'll then create something similar to what we used on base Python as `range(12)` which returns a list, instead using `np.arange(12)` which returns an ndarray with one axis.

```
import numpy as np  
a = np.arange(12)  
a
```

In [ ]: #

Out[ ]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

## Setting the dimensions with .reshape()

▀ Use the `.reshape()` method to change the dimensions of the ndarray, and check the `shape`, `ndim`, `size`, and `dtype` properties:

```
a = a.reshape(3,4)  
a  
a.shape  
a.ndim  
a.size  
a.dtype
```

```
In [ ]: #
```

```
Out[ ]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

```
Out[ ]: (3, 4)
```

```
Out[ ]: 2
```

```
Out[ ]: 12
```

```
Out[ ]: dtype('int32')
```

As we just saw, not only is the array numerical, but it's of a given type, `int32`, so an integer occupying 32 bits of memory each. Remember that each element in an ndarray has the same size.

▀ Let's use a little vectorization to see what happens if we convert to a different data type:

```
b = a * 1.5
b
b.dtype
```

```
In [ ]: #
```

```
Out[ ]: array([[ 0. ,  1.5,  3. ,  4.5],
               [ 6. ,  7.5,  9. , 10.5],
               [12. , 13.5, 15. , 16.5]])
```

```
Out[ ]: dtype('float64')
```

## Creating an ndarray from a collection: `np.array()`

▀ To create an ndarray from a collection of values, use the `array` method. The array method takes a single input (though it also accepts parameters such as `dtype`), which is a list or tuple of objects. Note that the `dtype` is determined based upon what values are provided, with the most efficient type chosen if all elements work. A typical example is integer favored over float, but only if all objects provided as inputs are integers.

```
b = np.array([1,2,3])
b
b.dtype
```

```
In [ ]: #
```

```
Out[ ]: array([1, 2, 3])
```

```
Out[ ]: dtype('int32')
```

### ▀▀ One float creates a float...

```
c = np.array([1,2.0,3])  
c  
c.dtype
```

```
In [ ]: #
```

```
Out[ ]: array([1., 2., 3.])  
Out[ ]: dtype('float64')
```

### ▀▀ Tuple:

```
t = np.array((1,2.0,3))  
t  
t.dtype
```

```
In [ ]: #
```

```
Out[ ]: array([1., 2., 3.])  
Out[ ]: dtype('float64')
```

So there's no difference between a numpy array created from a list than created from a tuple.  
Dictionaries are a little different, and are created as *objects*.

### ▀▀ Dictionary:

```
d = np.array({"first":1.0, "second":4.2, "third": 0.5})  
d  
d.dtype
```

```
In [ ]: #
```

```
Out[ ]: array({'first': 1.0, 'second': 4.2, 'third': 0.5}, dtype=object)  
Out[ ]: dtype('O')
```

## Simple element-wise mathematical operations

- See what you get by multiplying each of the above ndarrays by 2 (e.g. `2 * b`) and adding them together, etc.

```
In [ ]: #
Out[ ]: array([2., 4., 6.])
Out[ ]: array([0., 0., 0.])
Out[ ]: array([1., 1., 1.])
```

The `array` method transforms collections of collections into 2D arrays, collections of collections of collections for 3D, etc.

## Special ndarrays: zeros and ones

- Zeros and ones are often useful, either as placeholders to be replaced with other values, or a simple way to produce `False` and `True` values. Note that these functions let you *dimension* the arrays when you create them, *using either a list or tuple* -- the result is no different.

```
np.zeros([3,4])
np.ones((3,4))
```

```
In [ ]: #
Out[ ]: array([[0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.]])
Out[ ]: array([[1., 1., 1., 1.],
               [1., 1., 1., 1.],
               [1., 1., 1., 1.]])
```

## Math functions in numpy

NumPy functions are needed to provide mathematical functions of ndarrays. The `math` module only works with scalars, so we'll need to use NumPy functions instead.

- So let's use the NumPy variety of a trig function.

```
np.sin(a)
```

- However the built-in Python arithmetic operators work with ndarrays.

```
a**2
```

```
In [ ]: #
```

```
Out[ ]: array([[ 0.           ,  0.84147098,  0.90929743,  0.14112001],
               [-0.7568025 , -0.95892427, -0.2794155 ,  0.6569866 ],
               [ 0.98935825,  0.41211849, -0.54402111, -0.99999021]])
```

```
Out[ ]: array([[ 0,   1,   4,   9],
               [ 16,  25,  36,  49],
               [ 64,  81, 100, 121]], dtype=int32)
```

## Statistical summaries of ndarrays

- There are some useful statistical summaries that apply to the entire collection in the ndarray.

```
a.sum()  
a.min()  
a.max()  
a.mean()  
a.var()**0.5 # sd
```

```
In [ ]: #
```

```
Out[ ]: 66  
Out[ ]: 0  
Out[ ]: 11  
Out[ ]: 5.5  
Out[ ]: 3.452052529534663
```

## Random numbers

There's a lot to the world of probability and statistics, and students are referred to those courses and that literature for learning more. Generating random numbers (or rather *pseudo-random* numbers) is an important part of that. We'll just use a couple of common methods -- creating uniform and normally distributed random numbers -- but you should refer to <https://numpy.org/doc/stable/reference/random/index.html> (<https://numpy.org/doc/stable/reference/random/index.html>) for more NumPy methods for this.

- >Create an 2x3 ndarray of uniform random numbers with

```
np.random.rand(2,3)
```

```
In [ ]: #
```

```
Out[ ]: array([[0.42818175, 0.38804206, 0.9714968 ],
   [0.63946514, 0.76844697, 0.63632965]])
```

## Using a random number seed

If you want to create the same sequence of random numbers for repeatability, you can set a random number seed. The advantage of a seed is reproducibility; you'll always get the same sequence of random numbers for a given seed. The source <https://numpy.org/doc/stable/reference/random/index.html> (<https://numpy.org/doc/stable/reference/random/index.html>) recommends using the `default_rng` (random number generator) which has one parameter: the random number generator "seed". We'll use this for the following random number problems and use `42`, the answer to everything according to *The Hitchhiker's Guide to the Galaxy* (Douglas Adams (1978)).

To initiate the random number generator as `r` with a seed, we use the `.default_rng` method of `np.random`:



```
r = np.random.default_rng(42)
```

```
In [ ]: #
```

From there any call to `r` to access random numbers will follow the sequence initiated. (To start an identical sequence again, just run the `r` assignment above again.)

```
r = np.random.default_rng(42)
r.random(5)
r = np.random.default_rng(42) # to restart the same sequence
print("This should be the same as above:")
r.random(5)
print("But this continues the generation:")
r.random(5)
```

```
In [ ]: #
```

A set of random numbers using that seed of 42:

```
Out[ ]: array([0.77395605, 0.43887844, 0.85859792, 0.69736803, 0.09417735])
```

This should be the same as above, since we restarted the rng:

```
Out[ ]: array([0.77395605, 0.43887844, 0.85859792, 0.69736803, 0.09417735])
```

But this continues the generation:

```
Out[ ]: array([0.97562235, 0.7611397 , 0.78606431, 0.12811363, 0.45038594])
```

From here on, we'll either use `r` object to create random numbers from this seed or use `np.random` if we don't care about the seed, so for example we might do one or the other of the following to generate an ndarray of 5 uniformly distributed random numbers:

```
r.random(5)  
np.random.random(5)
```

```
In [ ]: #
```

```
Out[ ]: array([0.37079802, 0.92676499, 0.64386512, 0.82276161, 0.4434142 ])
```

```
Out[ ]: array([0.12805638, 0.07960468, 0.5564425 , 0.93766976, 0.61591846])
```

Let's use reshape to create a random set of XY values from a random 1D:

```
xy_data = r.random(12).reshape(6,2)  
xy_data
```

Then extract the X and Y values as arrays.

```
xdata = xy_data[:,0]  
ydata = xy_data[:,1]  
xdata, ydata
```

```
In [ ]: #
```

```
Out[ ]: array([[0.37079802, 0.92676499],  
               [0.64386512, 0.82276161],  
               [0.4434142 , 0.22723872],  
               [0.55458479, 0.06381726],  
               [0.82763117, 0.6316644 ],  
               [0.75808774, 0.35452597]])
```

```
Out[ ]: (array([0.37079802, 0.64386512, 0.4434142 , 0.55458479, 0.82763117,  
                0.75808774]),  
         array([0.92676499, 0.82276161, 0.22723872, 0.06381726, 0.6316644 ,  
                0.35452597]))
```

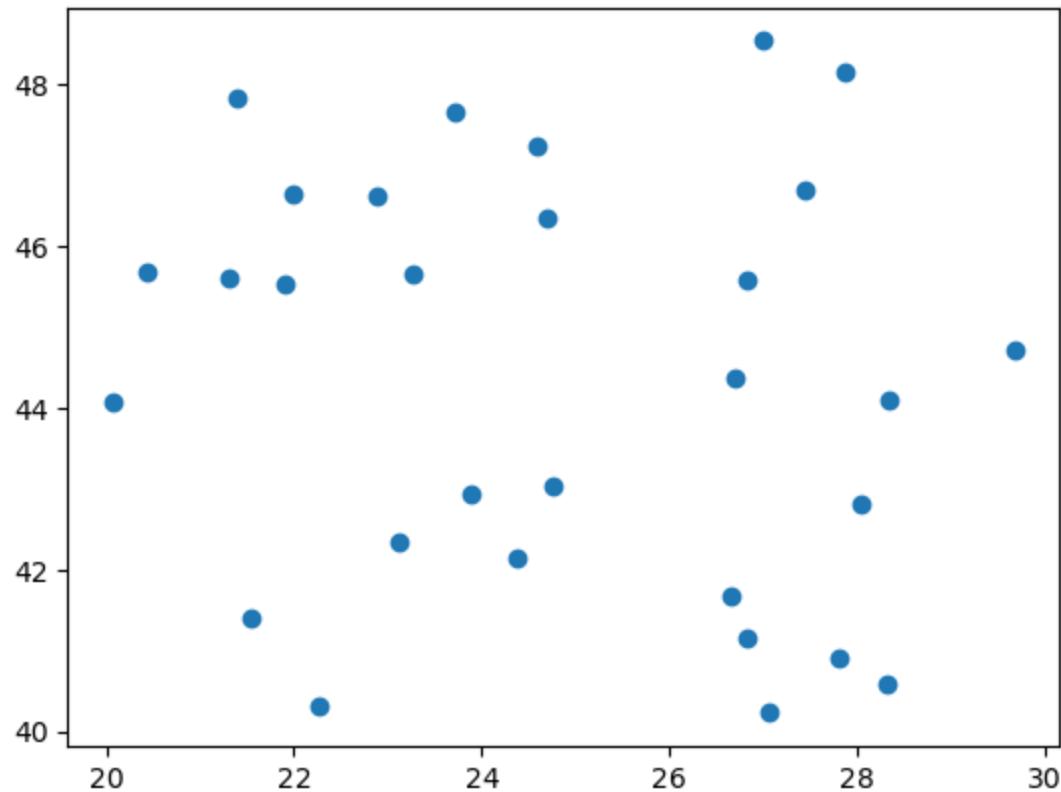
• After you've experiment with the above code, interpret the use of accessors in `xy_data[:,0]` :

:

■ Create a scatterplot of 30 random points (just use a 1D ndarray), with x values ranging from 20 to 30 and y values ranging from 40 to 50. The basic algorithm for this is `r * (max-min) + min` if `r` is a random number (or an array of random numbers) between 0 and 1, the minimum value is `min`, the maximum value is `max`.

```
In [ ]: #
```

```
Out[ ]: <matplotlib.collections.PathCollection at 0x1eece877e20>
```



## Normally distributed random numbers

▀▀ We can also create normally distributed random numbers. Here are two ways of creating one using the standard method of z scores (with a mean of 0 and a standard deviation of 1) within a 5x5 ndarray:

- the first using the `normal` method which uses the parameters `loc` for mean, `scale` for standard deviation, and `size` for sample size (including structure as a tuple if desired) with the usage `normal(loc=0.0, scale=1.0, size=None)` , so if we want to default to the mean and standard deviation, we need to use a specific size reference:

```
r.normal(size=(5,5))
```

- the second using `standard_normal` which uses those same defaults and just needs the size, so parameter specificity isn't required:

```
r.standard_normal((5,5))
```

In [ ]: #

```
Out[ ]: array([[-1.44711247, -1.32269961, -0.99724683,  0.39977423, -0.90547906],
 [-0.37816255,  1.2992283 , -0.35626397,  0.73751557, -0.93361768],
 [-0.20543756, -0.95002205, -0.33903308,  0.84030814, -1.72732042],
 [ 0.43442364,  0.2377356 , -0.59414996, -1.44605785,  0.07212951],
 [-0.52949271,  0.23267621,  0.02185215,  1.60177889, -0.23935563]])
```

In [ ]: #

```
Out[ ]: array([[ 0.43442364,  0.2377356 , -0.59414996, -1.44605785,  0.07212951],
 [-0.52949271,  0.23267621,  0.02185215,  1.60177889, -0.23935563],
 [-1.02349749,  0.17927563,  0.21999668,  1.35918758,  0.83511125],
 [ 0.35687106,  1.46330289, -1.18876305, -0.63975153, -0.92657594],
 [-0.3898098 , -1.37668615,  0.63515095, -0.2222227 , -1.47080629]])
```

▀▀ Specifying a different mean and standard deviation is pretty easy with `np.random.normal` :

```
mu, sigma = 100, 10 # mean and standard deviation
print(f"The mean (\u03bc 'mu') is {mu} and the standard deviation (\u03c3 'sigma') is {sigma}.")
r.normal(mu,sigma, (2,12))
```

```
In [ ]: #
```

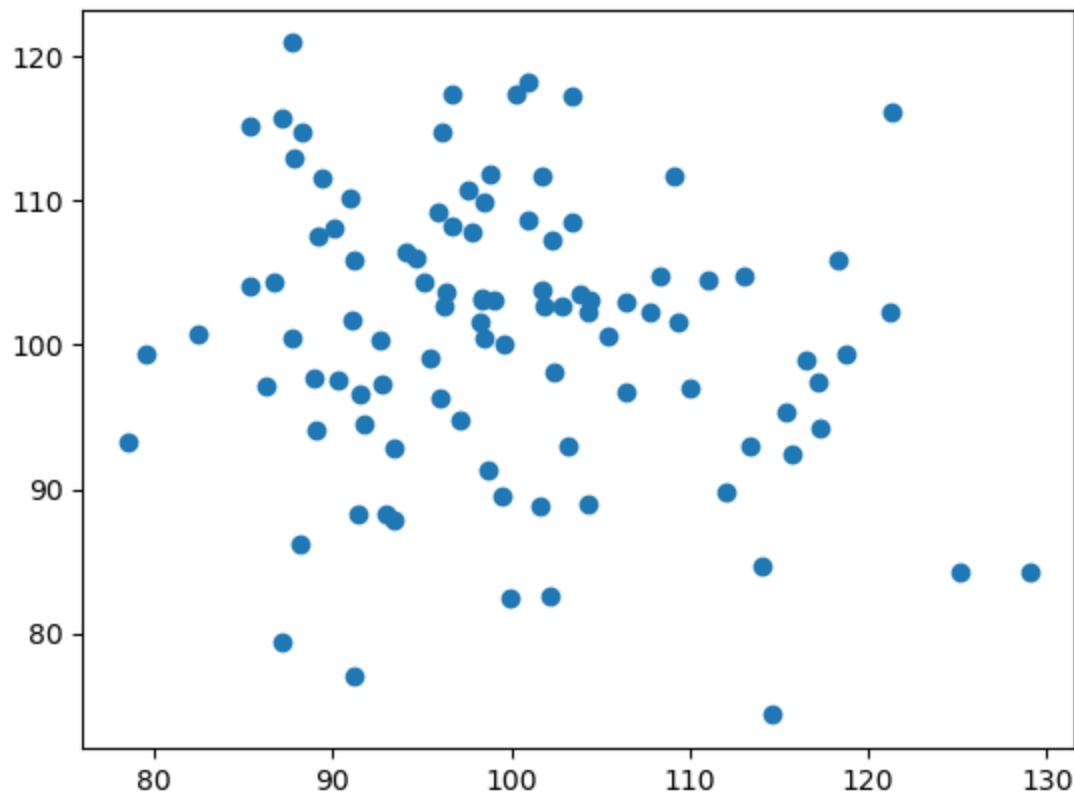
The mean ( $\mu$  'mu') is 100 and the standard deviation ( $\sigma$  'sigma') is 10.

```
Out[ ]: array([[ 91.23139222,  99.05737446,  82.42271609,  85.32954755,
   121.29247112,  87.12577419,  89.03214422, 118.36913528,
  129.05067169,  88.28433371,  96.31751043, 103.41555551],
 [117.28697644,  90.13142922,  97.54722154, 107.77337576,
 104.34766074,  96.23843929,  98.66177035,  86.25104192,
  97.61826256,  97.3361251 , 102.3216989 ,  94.44672781]])
```

- >Create a scatter plot of 100 normally distributed random x and y values (again just using 1D arrays), using the above mu and sigma values for both spatial dimensions.

```
In [ ]: #
```

```
Out[ ]: <matplotlib.collections.PathCollection at 0x1eece8f23d0>
```



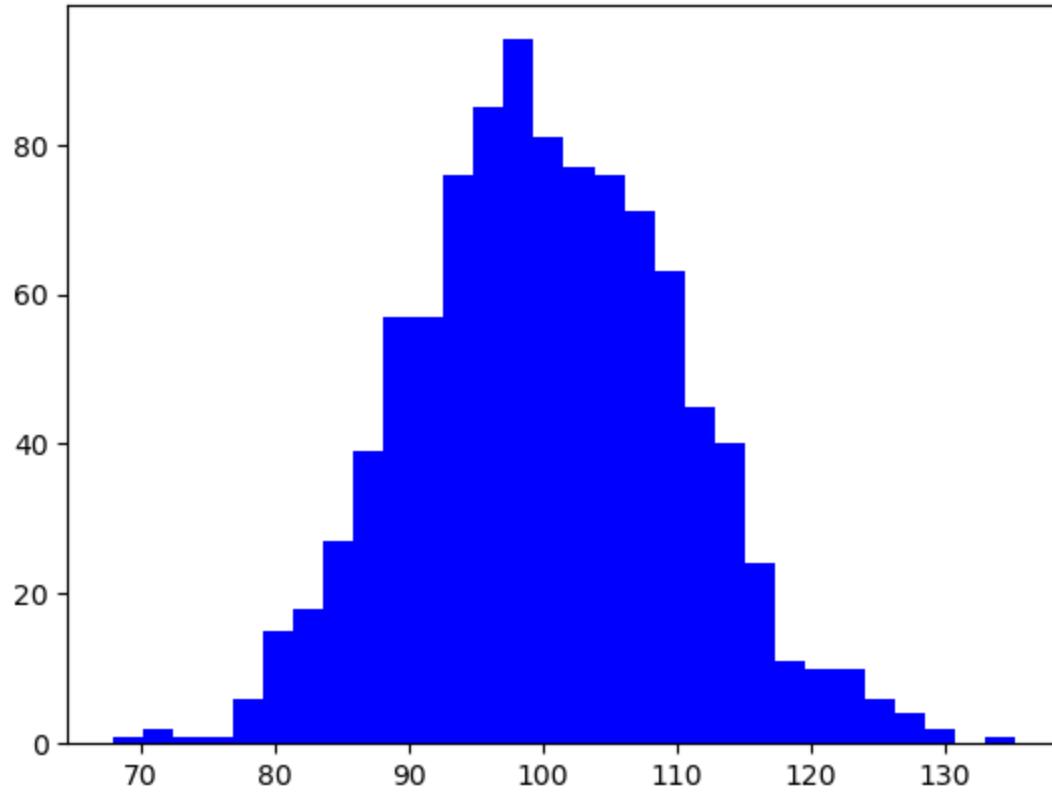
## Histogram.

Here's some code that creates a histogram. It's straightforward.

```
import numpy as np
mu, sigma = 100, 10 #
s = np.random.normal(mu, sigma, 1000)
plt.hist(s, 30, color="blue")
plt.show()
```

In [ ]: #

Out[ ]: (array([ 1., 2., 1., 1., 6., 15., 18., 27., 39., 57., 57., 76., 85.,
 94., 81., 77., 76., 71., 63., 45., 40., 24., 11., 10., 10., 6.,
 4., 2., 0., 1.]),  
array([ 67.97687354, 70.21905859, 72.46124363, 74.70342868,
 76.94561373, 79.18779878, 81.42998383, 83.67216888,
 85.91435393, 88.15653898, 90.39872402, 92.64090907,
 94.88309412, 97.12527917, 99.36746422, 101.60964927,
 103.85183432, 106.09401936, 108.33620441, 110.57838946,
 112.82057451, 115.06275956, 117.30494461, 119.54712966,
 121.78931471, 124.03149975, 126.2736848 , 128.51586985,
 130.7580549 , 133.00023995, 135.242425 ]),  
<BarContainer object of 30 artists>)



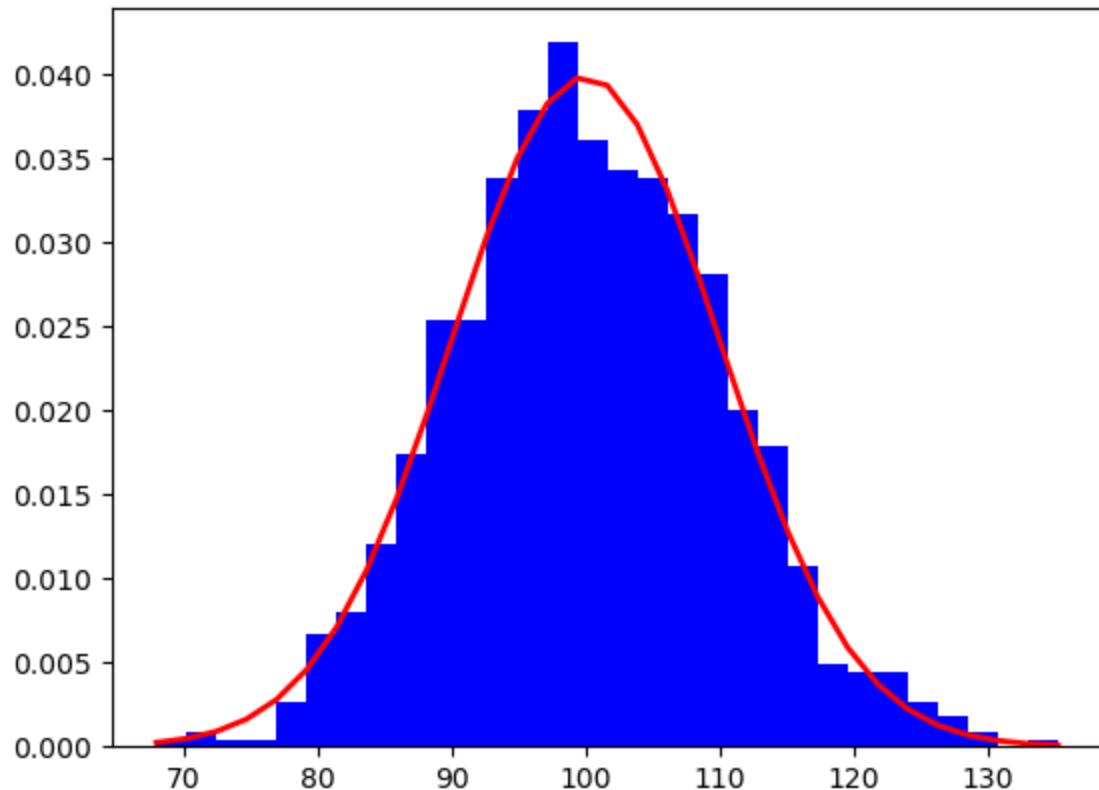
## Density plot

Creating a density plot is more complicated, so here's that completed code that you can puzzle at.

```
count, bins, ignored = plt.hist(s, 30, density=True, color="blue")
plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
         np.exp( - (bins - mu)**2 / (2 * sigma**2) ), linewidth=2, color='r')
plt.show()
```

In [ ]: #

Out[ ]: [<matplotlib.lines.Line2D at 0x1eecea510d0>]



## Creating plots from 2D ndarrays

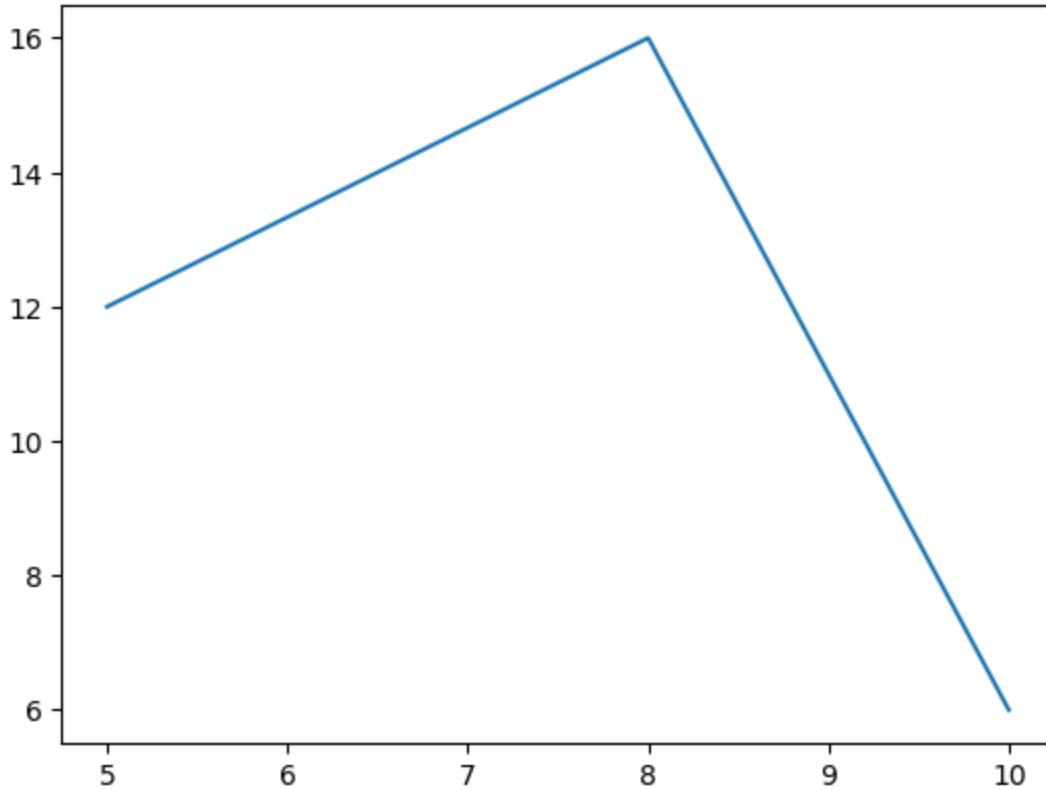
We'll create a 2D array and then use its two dimensions (0 and 1) as inputs to create a plot:

```
xy = np.array([(5,8,10),(12,16,6)])
print(xy)
plt.plot(xy[0],xy[1])
```

```
In [ ]: #
```

```
Out[ ]: array([[ 5.,  8., 10.],
   [12., 16.,  6.]])
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1ecdbfc4eb0>]
```



- Let's convert the axes of our 2D array to create 1D arrays, then add a bit to our Matplotlib skills by creating a title and label axes names:

```
x=xy[0]
type(x)
x.ndim
y=xy[1]
plt.plot(x,y)
plt.title('My Data')
plt.ylabel('Y axis')
plt.xlabel('X axis')
plt.show()
```

```
In [ ]: #
```

```
Out[ ]: numpy.ndarray
```

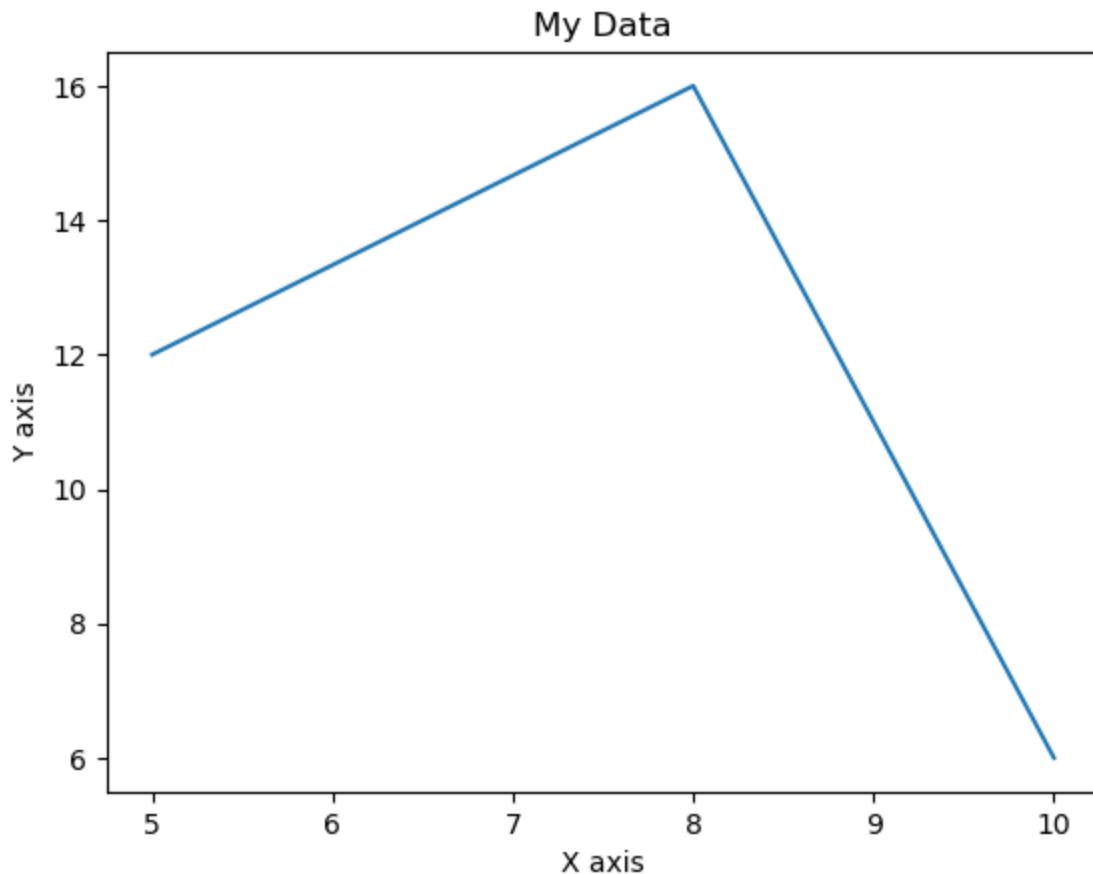
```
Out[ ]: 1
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1ecdc039850>]
```

```
Out[ ]: Text(0.5, 1.0, 'My Data')
```

```
Out[ ]: Text(0, 0.5, 'Y axis')
```

```
Out[ ]: Text(0.5, 0, 'X axis')
```



## Creating a raster of XY locations with `np.meshgrid()`

This might seem like an odd thing to do, but it's useful: Create a grid (like a raster) that stores the X coordinate as its (Z) value, and at the same time create one that stores the Y coordinate as its (Z) value.  Best seen by trying it:

```
[X,Y] = np.meshgrid(np.arange(5),np.arange(5))  
X  
Y
```

In [ ]: #

```
Out[ ]: array([[0, 1, 2, 3, 4],
   [0, 1, 2, 3, 4],
   [0, 1, 2, 3, 4],
   [0, 1, 2, 3, 4],
   [0, 1, 2, 3, 4]])
```

```
Out[ ]: array([[0, 0, 0, 0, 0],
   [1, 1, 1, 1, 1],
   [2, 2, 2, 2, 2],
   [3, 3, 3, 3, 3],
   [4, 4, 4, 4, 4]])
```

⌚ If we're looking at these 2D ndarrays as a raster, where's the origin? [*What's the meaning of origin as x and y values in any 2-dimensional graph?*]

:

One application of this is to create simulated rasters or to support mathematical transformation of real DEM rasters, as we used in a waveform landform study applying Fourier transforms:

Davis & Chojnacki (2017). Two-dimensional discrete Fourier transform analysis of karst and coral reef morphologies. *Transactions in GIS* 21(3). DOI 10.1111/tgis.12277 .

[https://www.researchgate.net/publication/316702784\\_Two-dimensional\\_discrete\\_Fourier\\_transform\\_analysis\\_of\\_karst\\_and\\_coral\\_reef\\_morphologies\\_DAVIS\\_and\\_CHOJNACKI](https://www.researchgate.net/publication/316702784_Two-dimensional_discrete_Fourier_transform_analysis_of_karst_and_coral_reef_morphologies_DAVIS_and_CHOJNACKI)  
[https://www.researchgate.net/publication/316702784\\_Two-dimensional\\_discrete\\_Fourier\\_transform\\_analysis\\_of\\_karst\\_and\\_coral\\_reef\\_morphologies\\_DAVIS\\_and\\_CHOJNACKI](https://www.researchgate.net/publication/316702784_Two-dimensional_discrete_Fourier_transform_analysis_of_karst_and_coral_reef_morphologies_DAVIS_and_CHOJNACKI)

## Meshgrids for mathematical functions

💻 We'll use a plot function for making a 3D plot...

```
def plot3d(array2d, label):
    fig = plt.figure()
    # old code : ax = fig.gca(projection='3d')
    ax = plt.axes(projection = '3d')
    surf = ax.plot_surface(X, Y, array2d)
    ax.set_zlim(-1.01, 1.01)
    plt.title(label)
    plt.show()
```

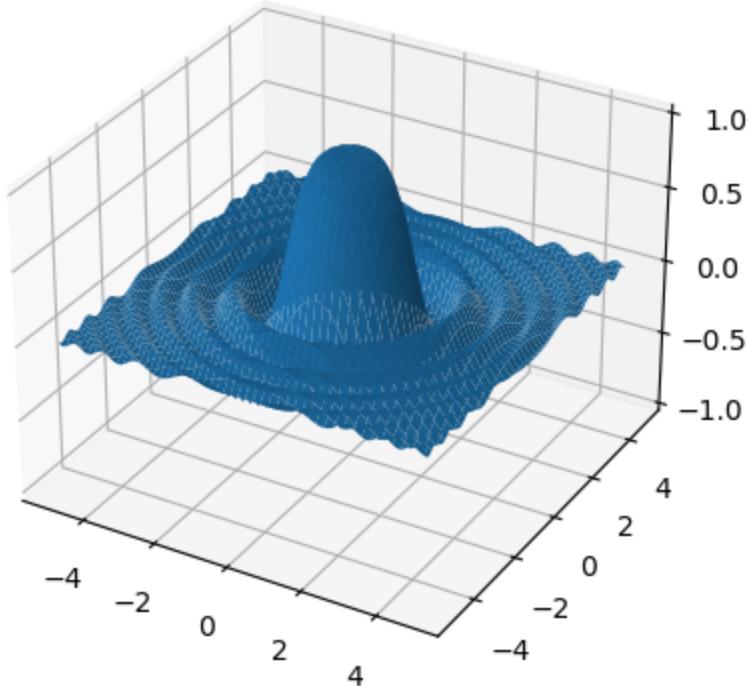
In [ ]: #

... then use this to plot a function.

```
[X,Y] = np.meshgrid(np.arange(-5,5,0.1), np.arange(-5,5,0.1))
f = np.sin(X**2 + Y**2)/(X**2 + Y**2)
plot3d(f,label=f"sin(X\u00b2 + Y\u00b2) / (X\u00b2 + Y\u00b2)")
```

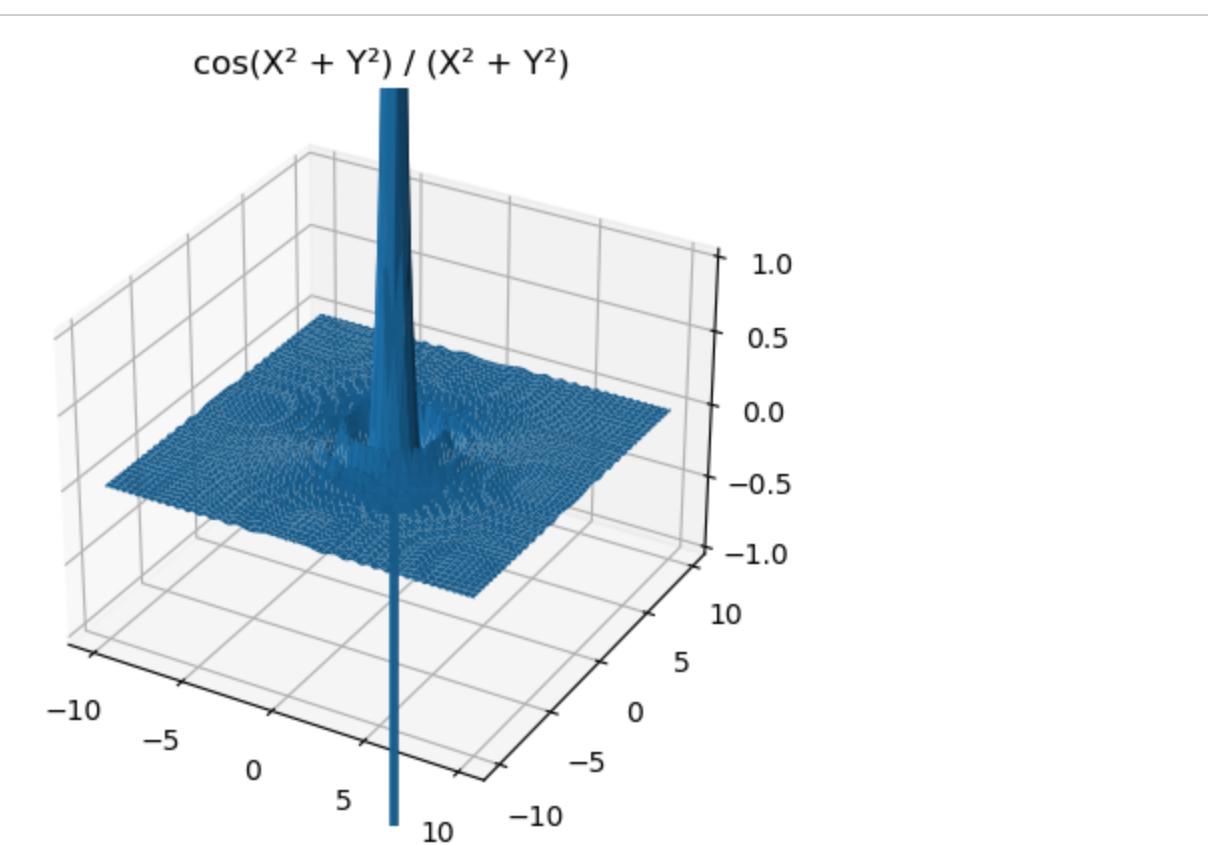
In [ ]: #

$$\sin(X^2 + Y^2) / (X^2 + Y^2)$$



Change the code to display the cosine instead, and increase the range width from -10 to 10, same steps.

In [ ]:



In [ ]:

## Reading in coordinates and plotting data

We'll see better methods when we get to pandas and Geopandas, but it is possible to read in coordinates and data into a 2D ndarray from a spreadsheet, in our case "npdata/exported\_ptdata.csv". We're going to be interested in 3 variables: `X`, `Y`, and `CATOT`, so we'll need to know where they are in the csv file.

▀ We'll start with a fairly simple way to read the first line to get the field names as a list. The following code opens the data, removes any spaces (hopefully we don't have any in our field names), splits the line of text at comma delimiters to create a list, then displays and closes the file.

```
f = open("npdata/exported_ptdata.csv", "r")
fields = f.readline().replace(" ","").split(",")
fields
f.close()
```

```
In [ ]: #
```

```
Out[ ]: ['X',  
         'Y',  
         'FID',  
         'AREA',  
         'PERIMETER',  
         'SAMPLES_',  
         'SAMPLES_ID',  
         'CATOT',  
         'MGTOT',  
         'ALK',  
         'SIO2',  
         'PH',  
         'TEMP',  
         'PCO2',  
         'IONSTR',  
         'TDS',  
         'CMOL',  
         'IONERR',  
         'CATXS',  
         'SATCALC',  
         'SATDOLO',  
         'SATQU',  
         'SATCX_5',  
         'SATCBLAC',  
         'NEGERR',  
         'CO2PERC',  
         'NEGPCO2\n']
```

→ But for a better view of the data, open the `exported_ptdata.csv` from the `npData` folder into Excel.

⌚ Knowing how Python counts things, what column numbers are `X`, `Y`, and `CATOT` in?

:

→ Close the file in Excel so you don't create a schema lock error (where two programs are trying to access the same data at the same time.)

💻 We'll skip the header line with variable names, and read into an ndarray using the comma delimiter.

```
from numpy import genfromtxt  
my_data = genfromtxt('npData/exported_ptdata.csv', delimiter=',', skip_header=1)  
my_data.ndim  
my_data
```

In [ ]: #

Out[ ]: 2

```
Out[ ]: array([[ 4.85701969e+05,  4.60295000e+06,  0.00000000e+00, ...,
   9.17000000e+00,  1.95000000e-01, -2.71000000e+00],
   [ 4.85956281e+05,  4.60294450e+06,  1.00000000e+00, ...,
   6.01000000e+00,  1.38000000e-01, -2.86000000e+00],
   [ 4.85589781e+05,  4.60293400e+06,  2.00000000e+00, ...,
  -1.19400000e+01,  2.51000000e-01, -2.60000000e+00],
   ...,
   [ 4.84966031e+05,  4.60018550e+06,  5.90000000e+01, ...,
  -6.51900000e+01,  6.00000000e-03, -4.21000000e+00],
   [ 4.85371469e+05,  4.59999550e+06,  6.00000000e+01, ...,
  8.07600000e+01,  4.30000000e-02, -3.37000000e+00],
   [ 4.84911094e+05,  4.59972650e+06,  6.10000000e+01, ...,
  7.74600000e+01,  2.60000000e-02, -3.59000000e+00]])
```

From this, we can see that the data is an ndarray with 2 axes.

To populate X, Y, and Ca (we'll use the element symbol instead of CATOT), we need to replace the underline in the following with the position of the field in the list of fields. I'll give you the first one.

```
X = my_data[:,0]
```

```
Y = my_data[:, _ ]
```

```
Ca = my_data[:, _ ]
```

In [ ]: #

Now we'll build the plot, starting with importing the module and selecting a style.

```
from matplotlib import pyplot as plt
from matplotlib import style
style.use('ggplot')
```

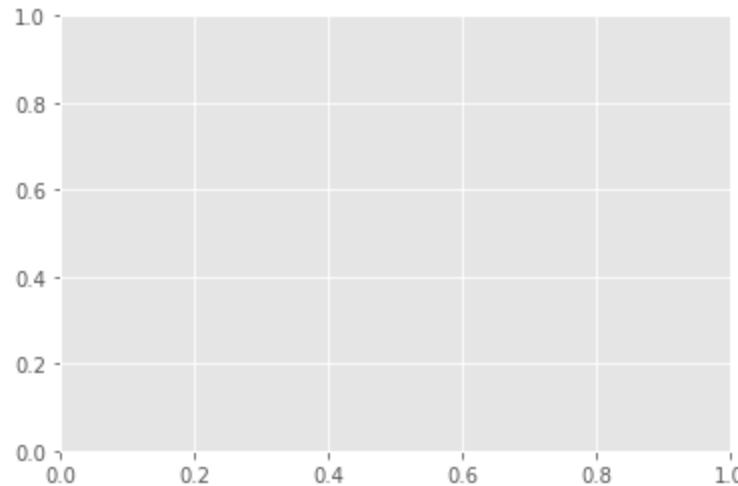
In [ ]: #

Some of the settings are specific to the plot itself and some are for the figure as a whole. The plot itself uses "axes" and typically a subplot is defined called `ax`, and if two separate y axes are used, might be created as `ax1` and `ax2`. To see what is created by default we can see what is returned by:

```
plt.subplots()
```

```
In [ ]: #
```

```
Out[ ]: (<Figure size 432x288 with 1 Axes>,
<matplotlib.axes._subplots.AxesSubplot at 0x1f8f9f57388>)
```

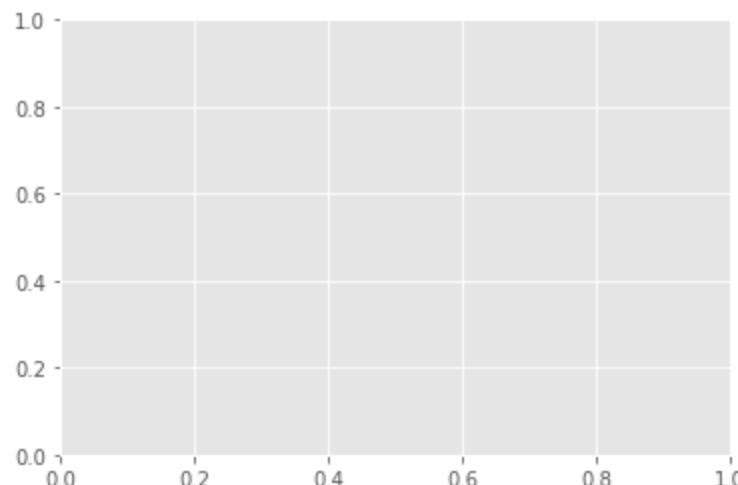


... which should show an empty default plot, but as you can see returns a tuple: (<Figure size 432x288 with 1 Axes>, <AxesSubplot:>)

... so we can assign the figure and axis to `fig, ax` using the method we've used before:

```
fig, ax = plt.subplots()
```

```
In [ ]: #
```



Then we can create a scatter plot and legend by referencing the `ax` subplot. We'll set the color (`c=`) to the variable `Ca` and color map (`cmap=`) to a yellow-orange-brown sequence "`YlOrBr`".

```
scatter = ax.scatter(X,Y,c=Ca,cmap="YlOrBr")
legend1 = ax.legend(*scatter.legend_elements(),
                   loc="lower right", title="Ca mg/L")
```

In [ ]: #

To make the scatterplot look like a map, we can just set the UTM coordinates (in metres) to have equally scaled axes, and then add a title to the overall figure (`fig.suptitle`) and the plot (`ax.set_title`).

```
ax.axis('equal')
ax.set_title("Calcium concentrations")
fig.suptitle("Marble Mountains, CA")
```

In [ ]: #

Out[ ]: (482578.3984375, 486117.1328125, 4599565.325, 4603111.175)

Out[ ]: Text(0.5, 1.0, 'Calcium concentrations')

Out[ ]: Text(0.5, 0.98, 'Marble Mountains, CA')

## Matplotlib elements: *put it all together*

Take all of the code starting from:

```
from numpy import genfromtxt
```

down to here and put it all in one code cell, below. To get this all to display in Jupyter, we need to have the `fig, ax` setting in the same code cell (I don't completely understand why), so we'll include all of the code that reads the data and creates the plot. End it by writing out the figure as a (default) .png with:

```
plt.savefig("MarbleCA")
```

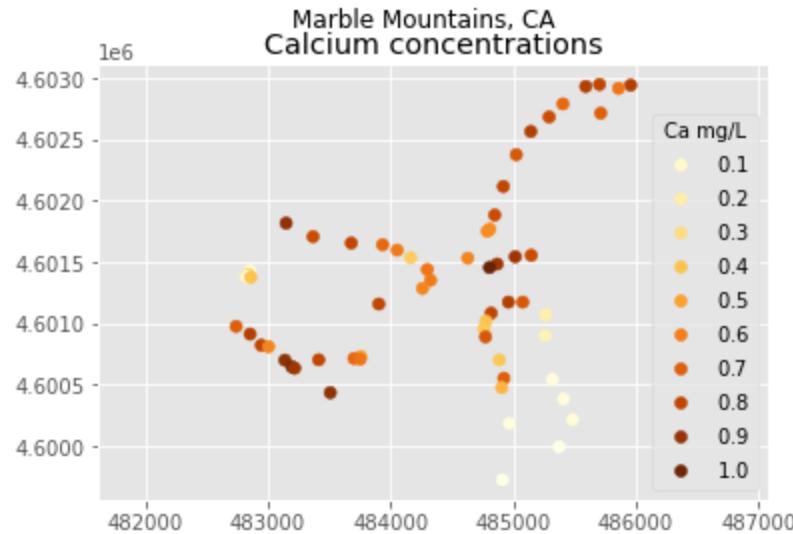
... which will save it in the same folder where this `.ipynb` file resides.

```
In [ ]: #
```

```
Out[ ]: (482578.3984375, 486117.1328125, 4599565.325, 4603111.175)
```

```
Out[ ]: Text(0.5, 1.0, 'Calcium concentrations')
```

```
Out[ ]: Text(0.5, 0.98, 'Marble Mountains, CA')
```



- ➡ Again, there's a lot more to Matplotlib (as well as NumPy), so if you have time I'd recommend going through the Basic, Pyplot and Image tutorials at <https://matplotlib.org/stable/tutorials> (<https://matplotlib.org/stable/tutorials>)

## key

- ➡ This directs you to do something specific, maybe in the operating system or answer something conceptual.
- 💻 Coding you need to do, in the subsequent code cell.
- ❓ Questions to answer in the same markdown cell.
- Similar to a question, but requesting an interpretation you need to provide, in the same markdown cell

# Introduction to Pandas

In this section we'll be taking a look at the powerful data analysis python library called pandas. We'll start by creating some dataframes. Then we'll explore various methods and properties available to us to access data within our dataframes.

- Enter the code below in your first cell, and since we'll also be using numpy, include that as well as pandas:

```
import pandas as pd  
import numpy as np
```

In [ ]: #

- Run the boilerplate that provides multiple outputs per cell:

```
In [ ]: from IPython.core.interactiveshell import InteractiveShell  
InteractiveShell.ast_node_interactivity = "all"
```

## Pandas data structures

### Series

- We'll start by building three series of data in code for selected weather stations in the Sierra, with temperature representing February normal temperatures. This example illustrates one way of building a dataframe (and happens to be similar to how you'd do it in R, but instead of calling them "vectors" we'll call them "series"). We'll then display one to see what a series looks like. The data we're entering here are all in the same order based on the weather station they come from, but we'll use either lists or tuples. We'll see their names shortly.

```
import pandas as pd  
elevation = pd.Series([52, 394, 510, 564, 725, 848, 1042, 1225, 1486, 1775, 1899, 2  
551])  
latitude = pd.Series([39.52, 38.91, 37.97, 38.70, 39.09, 39.25, 39.94, 37.75, 40.3  
5, 39.33, 39.17, 38.21])  
temperature = pd.Series((10.7, 9.7, 7.7, 9.2, 7.3, 6.7, 4.0, 5.0, 0.9, -1.1, -0.8,  
-4.4)) # tuple also works, effect is same
```

```
In [ ]: #
```

```
Out[ ]: 0    10.7
        1     9.7
        2     7.7
        3     9.2
        4     7.3
        5     6.7
        6     4.0
        7     5.0
        8     0.9
        9    -1.1
       10    -0.8
       11    -4.4
      dtype: float64
```

Note that when we instantiate the series, the numeric indices are automatically created.

▀ We can even create a series with a range of values or random numbers.

```
pd.Series(np.arange(5))
```

```
In [ ]: #
```

```
Out[ ]: 0    0
        1    1
        2    2
        3    3
        4    4
      dtype: int32
```

▀ ... or a constant ...

```
twos = pd.Series(2, index=range(5))
twos
```

```
In [ ]: #
```

```
Out[ ]: 0    2
        1    2
        2    2
        3    2
        4    2
      dtype: int64
```

⌚ What would you have gotten without the index setting, just `pd.Series(2)`? (First think of a likely answer, then try it.)

:

▀ ... or some random numbers. The following creates `r` as a random number generator (rng) with a seed of 42.

```
r = np.random.default_rng(seed=42)
pd.Series(r.random(10))
```

In [ ]: #

Out[ ]: 0 0.773956  
1 0.438878  
2 0.858598  
3 0.697368  
4 0.094177  
dtype: float64

Out[ ]: 0 0.773956  
1 0.438878  
2 0.858598  
3 0.697368  
4 0.094177  
dtype: float64

## Series from a dictionary

Dictionaries are used a lot in Pandas, so let's instead instantiate a series from a dictionary.

▀ We'll build an elev series this way, using the station name as the key for the index. Note that we'll create a new series of elevation, but name it `elev` to retain both versions since we'll use each type below when we're building dataframes. We'll do the same for latitude (`Lat`) and temperature (`temp`).

```
elevDict = {"Oroville":52,
            "Auburn":394,
            "Sonora":510,
            "Placerville":564,
            "Colfax":725,
            "Nevada City":848,
            "Quincy":1042,
            "Yosemite":1225,
            "Sierraville":1516,
            "Truckee":1775,
            "Tahoe City":1899,
            "Bodie":2551}
elev = pd.Series(elevDict)
elev
```

```
In [ ]: #
```

```
Out[ ]: Oroville      52
         Auburn       394
         Sonora       510
         Placerville   564
         Colfax        725
         Nevada City  848
         Quincy        1042
         Yosemite      1225
         Sierraville   1516
         Truckee       1775
         Tahoe City    1899
         Bodie         2551
         dtype: int64
```

■ A series is like a NumPy ndarray , so we can do similar operations, such as: elev[0]

```
In [ ]: #
```

```
Out[ ]: 52
```

■ Using a named index, find the elevation of "Placerville":

```
In [ ]: #
```

```
Out[ ]: 564
```

But if you want an actual ndarray , you can use .to\_numpy .

■ Using the .to\_numpy method on elev , derive elevarray :

```
In [ ]: #
```

```
Out[ ]: array([ 52, 394, 510, 564, 725, 848, 1042, 1225, 1516, 1775, 1899,
                2551], dtype=int64)
```

```
In [ ]:
```

## Vectorization of a series

Just as we saw with NumPy arrays, we can *vectorize* series, so for instance apply a mathematical operation or function.

■ Given this, how would we derive elevft from elev (which is in metres), assuming we knew there are 0.3048 m in 1 foot.

```
In [ ]: #
```

```
Out[ ]: Oroville      170.603675
         Auburn       1292.650919
         Sonora        1673.228346
         Placerville   1850.393701
         Colfax        2378.608924
         Nevada City  2782.152231
         Quincy        3418.635171
         Yosemite      4019.028871
         Sierraville   4973.753281
         Truckee       5823.490814
         Tahoe City    6230.314961
         Bodie          8369.422572
         dtype: float64
```

We're going to build a DataFrame from the three series and have the named indices of station names consistent for each.

- Make a copy of the latitude and temperature series we created earlier...

```
lat = latitude.copy()
temp = temperature.copy()
```

```
In [ ]: #
```

Note that if we had instead tried to copy them with `lat = latitude` etc., that would have created a reference to the same series so anything we modified to the one affects the other; using the `copy` method creates a separate object.

- ... then assign the indices from `elev` to the other two (since we know they're in the same order):

```
lat.index = elev.index
temp.index = elev.index
```

... and then display each to confirm that they have these named indices.

```
In [ ]: #
```

```
Out[ ]: Oroville      39.52
         Auburn       38.91
         Sonora       37.97
         Placerville   38.70
         Colfax        39.09
         Nevada City  39.25
         Quincy        39.94
         Yosemite      37.75
         Sierraville   40.35
         Truckee       39.33
         Tahoe City    39.17
         Bodie         38.21
         dtype: float64
```

```
Out[ ]: Oroville      10.7
         Auburn        9.7
         Sonora        7.7
         Placerville    9.2
         Colfax         7.3
         Nevada City   6.7
         Quincy         4.0
         Yosemite       5.0
         Sierraville    0.9
         Truckee        -1.1
         Tahoe City     -0.8
         Bodie          -4.4
         dtype: float64
```

## DataFrame

- Now we'll create a dataframe providing each series in a dictionary key:value pair, where the key is the variable or field name, and the value is the corresponding series. We made the field name the same as the series name. *We'll start with the numerically indexed series.*

```
sierra_i = pd.DataFrame({"temperature": temperature, "elevation": elevation, "latitude": latitude})
sierra_i
```

*Note that we've made sure that they're all in the right order using the process we followed.*

In [ ]: #

Out[ ]:

	temperature	elevation	latitude
0	10.7	52	39.52
1	9.7	394	38.91
2	7.7	510	37.97
3	9.2	564	38.70
4	7.3	725	39.09
5	6.7	848	39.25
6	4.0	1042	39.94
7	5.0	1225	37.75
8	0.9	1486	40.35
9	-1.1	1775	39.33
10	-0.8	1899	39.17
11	-4.4	2551	38.21

- Now we'll create a dataframe using the series with named indices. We had used abbreviated series names, but we'll make the field names longer.

```
sierra = pd.DataFrame({"temperature": temp, "elevation": elev, "latitude": lat})
sierra
```

In [ ]: #

Out[ ]:

	temperature	elevation	latitude
Oroville	10.7	52	39.52
Auburn	9.7	394	38.91
Sonora	7.7	510	37.97
Placerville	9.2	564	38.70
Colfax	7.3	725	39.09
Nevada City	6.7	848	39.25
Quincy	4.0	1042	39.94
Yosemite	5.0	1225	37.75
Sierraville	0.9	1516	40.35
Truckee	-1.1	1775	39.33
Tahoe City	-0.8	1899	39.17
Bodye	-4.4	2551	38.21

- We can also access individual series, by using object.property format such as ...

```
sierra.elevation
```

```
In [ ]: #
```

```
Out[ ]: Oroville      52
         Auburn       394
         Sonora       510
         Placerville   564
         Colfax        725
         Nevada City  848
         Quincy        1042
         Yosemite      1225
         Sierraville   1516
         Truckee       1775
         Tahoe City    1899
         Bodie         2551
         Name: elevation, dtype: int64
```

- ... ■ or individual values:

```
sierra.elevation["Truckee"]
```

```
In [ ]: #
```

```
Out[ ]: 1775
```

- What do you get when you apply the method .index to sierra ?

```
In [ ]: #
```

```
Out[ ]: Index(['Oroville', 'Auburn', 'Sonora', 'Placerville', 'Colfax', 'Nevada City',
               'Quincy', 'Yosemite', 'Sierraville', 'Truckee', 'Tahoe City', 'Bodie'],
               dtype='object')
```

- What do you get when you apply the method .columns to sierra ?

```
In [ ]: #
```

```
Out[ ]: Index(['temperature', 'elevation', 'latitude'], dtype='object')
```

- Interpret any similarities or differences for the .index and .columns results:

▀ You may remember the `len()` function from base Python, which returns the length of strings and lists. With numpy arrays it returns the total size, and is the same as what you get with `.size` on that array. Contrast what you get with `len()` and `.size` on the `sierra` dataframe, and interpret it below.

```
In [ ]: sierra.size  
len(sierra)
```

```
Out[ ]: 36
```

```
Out[ ]: 12
```

..

## Reading files into dataframes

We'll take a look at loading data from external files into a dataframe. If you recall from the lecture there are various file types we can load into a dataframe with the Pandas library. Below we'll examine reading in a csv file.

- You can read up on the various file types here ([https://pandas.pydata.org/pandas-docs/stable/user\\_guide/io.html](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html))

▀ We'll start with a bit more of the Sierra climate data, and read in a bit longer data frame from a CSV file

```
sierraFeb = pd.read_csv('pdData/sierraFeb.csv')  
sierraFeb
```

In [ ]: #

Out[ ]:

	STATION_NAME	COUNTY	ELEVATION	LATITUDE	LONGITUDE	PRECIPITATION	TEMPERATI
0	GROVELAND 2, CA US	Tuolumne	853.4	37.8444	-120.2258	176.02	
1	CANYON DAM, CA US	Plumas	1389.9	40.1705	-121.0886	164.08	
2	KERN RIVER PH 3, CA US	Kern	823.9	35.7831	-118.4389	67.06	
3	DONNER MEMORIAL ST PARK, CA US	Nevada	1809.6	39.3239	-120.2331	167.39	
4	BOWMAN DAM, CA US	Nevada	1641.3	39.4539	-120.6556	276.61	
...	...	...	...	...	...	...	...
77	PACIFIC HOUSE, CA US	El Dorado	1051.6	38.7583	-120.5030	220.22	
78	MAMMOTH LAKES RANGER STATION, CA US	Mono	2378.7	37.6478	-118.9617	NaN	
79	COLFAX, CA US	Placer	725.4	39.0911	-120.9480	207.26	
80	COLGATE POWERHOUSE, CA US	Yuba	181.4	39.3308	-121.1922	168.91	
81	BODIE CALIFORNIA STATE HISTORIC PARK, CA US	Mono	2551.2	38.2119	-119.0142	39.62	

82 rows × 7 columns



▀ It's usually a good idea to set a useful named index:

```
sierraFeb.set_index("STATION_NAME")
```

In [ ]: #

Out[ ]:

	STATION_NAME	COUNTY	ELEVATION	LATITUDE	LONGITUDE	PRECIPITATION	TEMPERATURE
	GROVELAND 2, CA US	Tuolumne	853.4	37.8444	-120.2258	176.02	6.1
	CANYON DAM, CA US	Plumas	1389.9	40.1705	-121.0886	164.08	1.4
	KERN RIVER PH 3, CA US	Kern	823.9	35.7831	-118.4389	67.06	8.9
	DONNER MEMORIAL ST PARK, CA US	Nevada	1809.6	39.3239	-120.2331	167.39	-0.9
	BOWMAN DAM, CA US	Nevada	1641.3	39.4539	-120.6556	276.61	2.9
	...	...	...	...	...	...	...
	PACIFIC HOUSE, CA US	El Dorado	1051.6	38.7583	-120.5030	220.22	NaN
	MAMMOTH LAKES RANGER STATION, CA US	Mono	2378.7	37.6478	-118.9617	NaN	-2.3
	COLFAX, CA US	Placer	725.4	39.0911	-120.9480	207.26	7.3
	COLGATE POWERHOUSE, CA US	Yuba	181.4	39.3308	-121.1922	168.91	NaN
	BODIE CALIFORNIA STATE HISTORIC PARK, CA US	Mono	2551.2	38.2119	-119.0142	39.62	-4.4

82 rows × 6 columns



... and from that we can reference a single column as a series `sierraFeb.PRECIPITATION`

```
In [ ]: #
```

```
Out[ ]: 0    176.02
        1    164.08
        2    67.06
        3    167.39
        4    276.61
        ...
       77   220.22
       78      NaN
       79   207.26
       80   168.91
       81   39.62
Name: PRECIPITATION, Length: 82, dtype: float64
```

## Vectorizing series from DataFrames

Just as with ndarrays, series can be vectorized.

- We can either use it from the DataFrame, or pull the series out as an individual series:

```
elevm = sierraFeb.ELEVATION
elevft = elevm / 0.3048
elevft
```

```
In [ ]: #
```

```
Out[ ]: 0    2799.868766
        1    4560.039370
        2    2703.083990
        3    5937.007874
        4    5384.842520
        ...
       77   3450.131234
       78   7804.133858
       79   2379.921260
       80   595.144357
       81   8370.078740
Name: ELEVATION, Length: 82, dtype: float64
```

- Or add a series to an existing dataframe

```
sierraFeb["ELEVATION_FT"] = sierraFeb.ELEVATION / 0.3048
sierraFeb
```

In [ ]: #

Out[ ]:

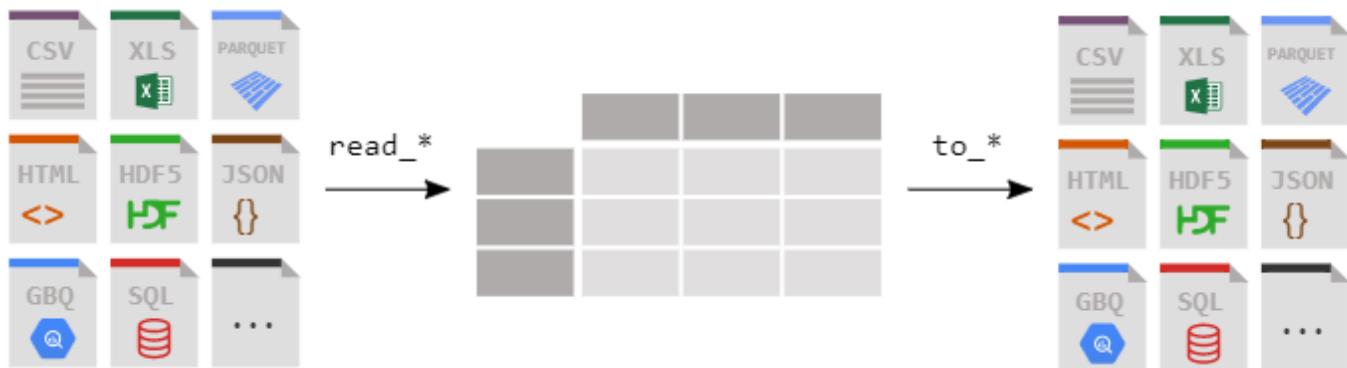
	STATION_NAME	COUNTY	ELEVATION	LATITUDE	LONGITUDE	PRECIPITATION	TEMPERATI
0	GROVELAND 2, CA US	Tuolumne	853.4	37.8444	-120.2258	176.02	
1	CANYON DAM, CA US	Plumas	1389.9	40.1705	-121.0886	164.08	
2	KERN RIVER PH 3, CA US	Kern	823.9	35.7831	-118.4389	67.06	
3	DONNER MEMORIAL ST PARK, CA US	Nevada	1809.6	39.3239	-120.2331	167.39	
4	BOWMAN DAM, CA US	Nevada	1641.3	39.4539	-120.6556	276.61	
...	...	...	...	...	...	...	...
77	PACIFIC HOUSE, CA US	El Dorado	1051.6	38.7583	-120.5030	220.22	
78	MAMMOTH LAKES RANGER STATION, CA US	Mono	2378.7	37.6478	-118.9617	NaN	
79	COLFAX, CA US	Placer	725.4	39.0911	-120.9480	207.26	
80	COLGATE POWERHOUSE, CA US	Yuba	181.4	39.3308	-121.1922	168.91	
81	BODIE CALIFORNIA STATE HISTORIC PARK, CA US	Mono	2551.2	38.2119	-119.0142	39.62	

82 rows × 8 columns



## JSON (and other tabular) files

JSON files are dictionary-like files that are often used for input and output with various apps. To write and read JSON files is similar to writing and reading CSVs and quite a lot of other formats, as you can see at [\(https://pandas.pydata.org/docs/getting\\_started/intro\\_tutorials/02\\_read\\_write.html\)](https://pandas.pydata.org/docs/getting_started/intro_tutorials/02_read_write.html), using either `read_*` or `to_*`.



- We'll look at this for JSON files by first creating a JSON file (just so we have something to read, but also to show how we do this)...

```
sierraFeb.to_json("pdData/sierraFeb.json")
```

In [ ]: #

- ... and then reading it in.

```
sierraFebNew = pd.read_json("pdData/sierraFeb.json")
sierraFebNew
```

In [ ]:

#

Out[ ]:

	STATION_NAME	COUNTY	ELEVATION	LATITUDE	LONGITUDE	PRECIPITATION	TEMPERATI
0	GROVELAND 2, CA US	Tuolumne	853.4	37.8444	-120.2258	176.02	
1	CANYON DAM, CA US	Plumas	1389.9	40.1705	-121.0886	164.08	
2	KERN RIVER PH 3, CA US	Kern	823.9	35.7831	-118.4389	67.06	
3	DONNER MEMORIAL ST PARK, CA US	Nevada	1809.6	39.3239	-120.2331	167.39	
4	BOWMAN DAM, CA US	Nevada	1641.3	39.4539	-120.6556	276.61	
...	...	...	...	...	...	...	...
77	PACIFIC HOUSE, CA US	El Dorado	1051.6	38.7583	-120.5030	220.22	
78	MAMMOTH LAKES RANGER STATION, CA US	Mono	2378.7	37.6478	-118.9617	NaN	
79	COLFAX, CA US	Placer	725.4	39.0911	-120.9480	207.26	
80	COLGATE POWERHOUSE, CA US	Yuba	181.4	39.3308	-121.1922	168.91	
81	BODIE CALIFORNIA STATE HISTORIC PARK, CA US	Mono	2551.2	38.2119	-119.0142	39.62	

82 rows × 8 columns



Just to see what the JSON file looks like, we could open it in a text editor (like Notepad) separately, or we can just use the standard Python read method:

```
infile = "pdData/sierraFeb.json"
f = open(infile, "r")
f.readline()
f.close()
```

In [ ]: #

Out[ ]: '{
 "STATION\_NAME": {
 "0": "GROVELAND 2, CA US",
 "1": "CANYON DAM, CA US",
 "2": "KERN RIVER PH 3, CA US",
 "3": "DONNER MEMORIAL ST PARK, CA US",
 "4": "BOWMAN DAM, CA US",
 "5": "BRUSH CREEK RANGER STATION, CA US",
 "6": "GRANT GROVE, CA US",
 "7": "LEE VINING, CA US",
 "8": "OROVILLE MUNICIPAL AIRPORT, CA US",
 "9": "LEMON COVE, CA US",
 "10": "CALAVERAS BIG TREES, CA US",
 "11": "BUCKS CREEK, CA US",
 "12": "GRASS VALLEY NUMBER 2, CA US",
 "13": "PLACERVILLE, CA US",
 "14": "THREE RIVERS EDISON PH 1, CA US",
 "15": "GLENNVILLE, CA US",
 "16": "MATHER, CA US",
 "17": "BLUE CANYON AIRPORT, CA US",
 "18": "GEM LAKE, CA US",
 "19": "MINERAL, CA US",
 "20": "SUSANVILLE 2 SW, CA US",
 "21": "BRIDGEPORT, CA US",
 "22": "GOLD RUN 2 SW, CA US",
 "23": "FORESTILL RANGER STATION, CA US",
 "24": "MANZANITA LAKE, CA US",
 "25": "OROVILLE, CA US",
 "26": "QUINCY, CA US",
 "27": "VISALIA, CA US",
 "28": "PORTOLA, CA US",
 "29": "BALCH POWER HOUSE, CA US",
 "30": "DOWNIEVILLE, CA US",
 "31": "HAIWEE, CA US",
 "32": "VOLTA POWER HOUSE, CA US",
 "33": "CAMP PARDEE, CA US",
 "34": "ELLERY LAKE, CA US",
 "35": "TRUCKEE RANGER STATION, CA US",
 "36": "BISHOP AIRPORT, CA US",
 "37": "DESABLA, CA US",
 "38": "LAKE SABRINA, CA US",
 "39": "ASH MOUNTAIN, CA US",
 "40": "SOUTH LAKE TAHOE AIRPORT, CA US",
 "41": "LINDSAY, CA US",
 "42": "CHERRY VALLEY DAM, CA US",
 "43": "AUBERRY 2 NW, CA US",
 "44": "AUBURN, CA US",
 "45": "NORTH FORK RANGE R STATION, CA US",
 "46": "NEVADA CITY, CA US",
 "47": "REPRESA, CA US",
 "48": "SIERRAVILLE R S, CA US",
 "49": "CHESTER, CA US",
 "50": "NEW MELONES DAM HQ, CA US",
 "51": "CHICO UNIVERSITY FARM, CA US",
 "52": "EXCHEQUER DAM, CA US",
 "53": "BIG CREEK PH 1, CA US",
 "54": "SONORA, CA US",
 "55": "WEST POINT, CA US",
 "56": "SO ENTRANCE YOSEMITE N.P., CA US",
 "57": "HARRY ENGLEBRIGHT DM, CA US",
 "58": "HETCH HETCHY, CA US",
 "59": "SUTTER HILL CDF, CA US",
 "60": "FRIANT GOVERNMENT CAMP, CA US",
 "61": "GEORGETOWN R S, CA US",
 "62": "PINE FLAT DAM, CA US",
 "63": "INDEPENDENCE, CA US",
 "64": "FIDDLETOWN DEXTER RANCH, CA US",
 "65": "STRAWBERRY VALLEY, CA US",
 "66": "YOSEMITIE VILLAGE 12 W, CA US",
 "67": "KELSEY 1 N, CA US",
 "68": "PARADISE, CA US",
 "69": "TAHOE CITY, CA US",
 "70": "INYOKERN, CA US",
 "71": "SOUTH LAKE, CA US",
 "72": "YOSEMITIE PARK HEADQUARTERS, CA US",
 "73": "BISHOP CRK INTAKE 2, CA US",
 "74": "BOCA, CA US",
 "75": "HUNTINGTON LAKE, CA US",
 "76": "LODGEPOLE, CA US",
 "77": "PACIFIC HOUSE, CA US",
 "78": "MAMMOTH LAKES RANGER STATION, CA US",
 "79": "COLFAX, CA US",
 "80": "COLGATE POWERHOUSE, CA US",
 "81": "BODIE CALIFORNIA STATE HISTORIC PARK, CA US"
 },
 "COUNTY": {
 "0": "Tuolumne",
 "1": "Plumas",
 "2": "Kern",
 "3": "Nevada",
 "4": "Nevada",
 "5": "Butte",
 "6": "Tulare",
 "7": "Mono",
 "8": "Butte",
 "9": "Tulare",
 "10": "Calaveras",
 "11": "Plumas",
 "12": "Nevada",
 "13": "El Dorado",
 "14": "Tulare",
 "15": "Kern",
 "16": "Tuolumne",
 "17": "Placer",
 "18": "Mono",
 "19": "Tehama",
 "20": "Lassen",
 "21": "Mono",
 "22": "Placer",
 "23": "Placer",
 "24": "Shasta",
 "25": "Butte",
 "26": "Plumas",
 "27": "Tulare",
 "28": "Plumas",
 "29": "Fresno",
 "30": "Sierra",
 "31": "Inyo",
 "32": "Shasta",
 "33": "Calaveras",
 "34": "Mono",
 "35": "Nevada",
 "36": "Inyo",
 "37": "Butte",
 "38": "Inyo",
 "39": "Tulare",
 "40": "El Dorado",
 "41": "Tulare",
 "42": "Tuolumne",
 "43": "Fresno",
 "44": "Placer",
 "45": "Madera",
 "46": "Nevada",
 "47": "Sacramento",
 "48": "Sierra",
 "49": "Plumas",
 "50": "Tuolumne",
 "51": "Butte",
 "52": "Mariposa",
 "53": "Fresno",
 "54": "Tuolumne",
 "55": "Calaveras",
 "56": "Mariposa",
 "57": "Nevada",
 "58": "Tuolumne",
 "59": "Amador",
 "60": "Fresno",
 "61": "El Dorado",
 "62": "Fresno",
 "63": "Inyo",
 "64": "Amador",
 "65": "Yuba",
 "66": "Mariposa",
 "67": "El Dorado",
 "68": "Butte",
 "69": "Placer",
 "70": "Kern",
 "71": "Inyo",
 "72": "Mariposa",
 "73": "Inyo",
 "74": "Nevada",
 "75": "Fresno",
 "76": "Tulare",
 "77": "El Dorado",
 "78": "Mono",
 "79": "Placer",
 "80": "Yuba",
 "81": "Mono"
 },
 "ELEVATION": {
 "0": 853,
 "1": 1389.9,
 "2": 823.9,
 "3": 1809.6,
 "4": 1641.3,
 "5": 1085.1,
 "6": 2011.7,
 "7": 2071.7,
 "8": 57.9,
 "9": 156.4,
 "10": 1431.0,
 "11": 576.4,
 "12": 731.5,
 "13": 563.9,
 "14": 347.5,
 "15": 957.1,
 "16": 1374.6,
 "17": 1608.1,
 "18": 2734.1,
 "19": 1485.9,
 "20": 1283.8,
 "21": 1972.1,
 "22": 1011.9,
 "23": 919.0,
 "24": 1752.6,
 "25": 52.1,
 "26": 1042.4,
 "27": 106.7,
 "28": 1478.3,
 "29": 528.8,
 "30": 888.5,
 "31": 1165.9,
 "32": 676.7,
 "33": 200.6,
 "34": 2939.8,
 "35": 1774.9,
 "36": 1250.3,
 "37": 826.0,
 "38": 2763.0,
 "39": 520.6,
 "40": 1924.5,
 "41": 132.6,
 "42": 1452.4,
 "43": 637.0,
 "44": 393.8,
 "45": 806.2,
 "46": 847.6,
 "47": 89.9,
 "48": 1516.4,
 "49": 1380.7,
 "50": 292.6,
 "51": 56.4,
 "52": 134.7,
 "53": 1486.8,
 "54": 510.5,
 "55": 845.8,
 "56": 1538.3,
 "57": 243.8,
 "58": 1179.6,
 "59": 483.4,
 "60": 125.0,
 "61": 914.7,
 "62": 187.5,
 "63": 1204.0,
 "64": 658.4,
 "65": 1160.7,
 "66": 2017.8,
 "67": 609.6,
 "68": 151.2,
 "69": 1179.6,
 "70": 125.0,
 "71": 914.7,
 "72": 187.5,
 "73": 1204.0,
 "74": 658.4,
 "75": 1160.7,
 "76": 2017.8,
 "77": 609.6,
 "78": 151.2,
 "79": 1179.6,
 "80": 125.0,
 "81": 914.7,
 "82": 187.5,
 "83": 1204.0,
 "84": 658.4,
 "85": 1160.7,
 "86": 2017.8,
 "87": 609.6,
 "88": 151.2,
 "89": 1179.6,
 "90": 125.0,
 "91": 914.7,
 "92": 187.5,
 "93": 1204.0,
 "94": 658.4,
 "95": 1160.7,
 "96": 2017.8,
 "97": 609.6,
 "98": 151.2,
 "99": 1179.6
 }
}'

8":533.4,"69":1898.9,"70":740.7,"71":2920.0,"72":1224.7,"73":2485.3,"74":169  
 9.3,"75":2139.7,"76":2052.8,"77":1051.6,"78":2378.7,"79":725.4,"80":181.4,"8  
 1":2551.2}],"LATITUDE": {"0":37.8444,"1":40.1705,"2":35.7831,"3":39.3239,"4":3  
 9.4539,"5":39.6949,"6":36.7394,"7":37.9567,"8":39.49,"9":36.3817,"10":38.276  
 9,"11":39.9372,"12":39.2041,"13":38.6955,"14":36.465,"15":35.7269,"16":37.88  
 5,"17":39.2774,"18":37.7519,"19":40.3458,"20":40.4167,"21":38.2575,"22":39.16  
 5,"23":39.01,"24":40.54111,"25":39.5177,"26":39.9366,"27":36.3278,"28":39.805  
 3,"29":36.9092,"30":39.5633,"31":36.1388,"32":40.4583,"33":38.2486,"34":37.93  
 55,"35":39.333,"36":37.3711,"37":39.8716,"38":37.213,"39":36.4914,"40":38.898  
 3,"41":36.2032,"42":37.9747,"43":37.0919,"44":38.9072,"45":37.2329,"46":39.24  
 66,"47":38.6944,"48":39.5833,"49":40.3033,"50":38.0047,"51":39.6911,"52":37.5  
 85,"53":37.2064,"54":37.9672,"55":38.3775,"56":37.5122,"57":39.2372,"58":37.9  
 613,"59":38.3772,"60":36.9969,"61":38.933,"62":36.821,"63":36.798,"64":38.523  
 6,"65":39.563,"66":37.7592,"67":38.8088,"68":39.7538,"69":39.1678,"70":35.651  
 3,"71":37.1683,"72":37.75,"73":37.248,"74":39.3886,"75":37.2275,"76":36.604  
 4,"77":38.7583,"78":37.6478,"79":39.0911,"80":39.3308,"81":38.2119}],"LONGITUD  
 E": {"0": -120.2258,"1": -121.0886,"2": -118.4389,"3": -120.2331,"4": -120.655  
 6,"5": -121.3452,"6": -118.9631,"7": -119.1194,"8": -121.61833,"9": -119.0264,"1  
 0": -120.3113,"11": -121.3147,"12": -121.068,"13": -120.8244,"14": -118.8619,"15":  
 -118.7006,"16": -119.8561,"17": -120.7102,"18": -119.1402,"19": -121.6091,"20": -1  
 20.6631,"21": -119.2286,"22": -120.8566,"23": -120.8455,"24": -121.57667,"25": -12  
 1.553,"26": -120.9475,"27": -119.2994,"28": -120.4719,"29": -119.0883,"30": -120.8  
 238,"31": -117.9527,"32": -121.8663,"33": -120.8433,"34": -119.2305,"35": -120.17  
 3,"36": -118.358,"37": -121.6108,"38": -118.6136,"39": -118.8253,"40": -119.994  
 7,"41": -119.0545,"42": -119.9161,"43": -119.5128,"44": -121.0838,"45": -119.509  
 7,"46": -121.0008,"47": -121.1611,"48": -120.3705,"49": -121.2422,"50": -120.486  
 3,"51": -121.8211,"52": -120.2672,"53": -119.2419,"54": -120.3872,"55": -120.545  
 2,"56": -119.6331,"57": -121.2666,"58": -119.783,"59": -120.8008,"60": -119.707  
 2,"61": -120.8008,"62": -119.3374,"63": -118.2036,"64": -120.7061,"65": -121.107  
 7,"66": -119.8208,"67": -120.8208,"68": -121.6241,"69": -120.1428,"70": -117.821  
 3,"71": -118.5705,"72": -119.5897,"73": -118.5813,"74": -120.0936,"75": -119.2205  
 6,"76": -118.7325,"77": -120.503,"78": -118.9617,"79": -120.948,"80": -121.1922,"8  
 1": -119.0142}],"PRECIPITATION": {"0": 176.02,"1": 164.08,"2": 67.06,"3": 167.3  
 9,"4": 276.61,"5": 296.16,"6": 186.18,"7": 71.88,"8": 137.67,"9": 62.74,"10": 254.  
 0,"11": 282.7,"12": 229.11,"13": 170.69,"14": 114.05,"15": 95.25,"16": 152.4,"17": 2  
 68.22,"18": 81.53,"19": 217.93,"20": 45.21,"21": 41.4,"22": 224.28,"23": 214.88,"2  
 4": 132.84,"25": 123.7,"26": 181.86,"27": 46.99,"28": 98.3,"29": 144.27,"30": 267.2  
 1,"31": 37.59,"32": 127.0,"33": 98.04,"34": 93.98,"35": 126.49,"36": 21.59,"37": 29  
 5.15,"38": 68.83,"39": 122.68,"40": 66.55,"41": 54.61,"42": 220.47,"43": 117.86,"4  
 4": 159.51,"45": 155.45,"46": 268.48,"47": 104.65,"48": null,"49": 135.13,"50": 125.  
 73,"51": 112.27,"52": 90.17,"53": 148.34,"54": 147.83,"55": 162.56,"56": 198.37,"5  
 7": 150.11,"58": 149.1,"59": 140.21,"60": 66.04,"61": 227.08,"62": 95.25,"63": 25.6  
 5,"64": 155.96,"65": 346.71,"66": 198.12,"67": 162.05,"68": 255.52,"69": 144.53,"7  
 0": 30.23,"71": 83.57,"72": 169.16,"73": 49.02,"74": 86.11,"75": 204.22,"76": 214.3  
 8,"77": 220.22,"78": null,"79": 207.26,"80": 168.91,"81": 39.62}),"TEMPERATURE":  
 {"0": 6.1,"1": 1.4,"2": 8.9,"3": -0.9,"4": 2.9,"5": null,"6": 1.7,"7": 0.4,"8": 10.  
 3,"9": 11.3,"10": 2.7,"11": null,"12": 6.9,"13": 9.2,"14": 9.7,"15": 6.5,"16": nul  
 1,"17": 4.1,"18": null,"19": 0.9,"20": 2.4,"21": -2.2,"22": null,"23": null,"24": 0.  
 2,"25": 10.7,"26": 4.0,"27": 10.9,"28": 0.5,"29": 9.2,"30": 5.3,"31": 9.3,"32": nul  
 1,"33": 10.1,"34": null,"35": -1.1,"36": 5.7,"37": 7.1,"38": null,"39": 9.8,"40": -0.  
 6,"41": 11.1,"42": 4.4,"43": 8.8,"44": 9.7,"45": 7.2,"46": 6.7,"47": null,"48": 0.  
 8,"49": 0.7,"50": 10.1,"51": 9.8,"52": null,"53": 4.7,"54": 7.7,"55": null,"56": 2.  
 5,"57": null,"58": 4.9,"59": 9.3,"60": 10.8,"61": 7.2,"62": null,"63": 7.6,"64": nul  
 1,"65": 4.6,"66": 2.5,"67": null,"68": 9.3,"69": -0.8,"70": 9.8,"71": null,"72": 5.  
 0,"73": null,"74": -1.4,"75": 1.3,"76": -1.4,"77": null,"78": -2.3,"79": 7.3,"80": nu  
 ll,"81": -4.4}),"ELEVATION\_FT": {"0": 2799.8687664042,"1": 4560.0393700787,"2": 270  
 3.0839895013,"3": 5937.0078740157,"4": 5384.842519685,"5": 3560.0393700787,"6": 6

```
600.0656167979,"7":6796.9160104987,"8":189.9606299213,"9":513.1233595801,"1  
0":4694.8818897638,"11":1891.0761154856,"12":2399.9343832021,"13":1850.065616  
7979,"14":1140.0918635171,"15":3140.0918635171,"16":4509.842519685,"17":5275.  
9186351706,"18":8970.1443569554,"19":4875.0,"20":4211.9422572178,"21":6470.14  
43569554,"22":3319.8818897638,"23":3015.0918635171,"24":5750.0,"25":170.93175  
85302,"26":3419.9475065617,"27":350.0656167979,"28":4850.0656167979,"29":173  
4.9081364829,"30":2915.0262467192,"31":3825.1312335958,"32":2220.144356955  
4,"33":658.1364829396,"34":9645.0131233596,"35":5823.1627296588,"36":4102.034  
1207349,"37":2709.9737532808,"38":9064.9606299213,"39":1708.0052493438,"40":6  
313.9763779528,"41":435.0393700787,"42":4765.0918635171,"43":2089.895013123  
4,"44":1291.9947506562,"45":2645.0131233596,"46":2780.8398950131,"47":294.947  
5065617,"48":4975.0656167979,"49":4529.8556430446,"50":959.9737532808,"51":18  
5.0393700787,"52":441.9291338583,"53":4877.9527559055,"54":1674.8687664042,"5  
5":2774.9343832021,"56":5046.9160104987,"57":799.8687664042,"58":3870.0787401  
575,"59":1585.9580052493,"60":410.1049868766,"61":3000.9842519685,"62":615.15  
7480315,"63":3950.1312335958,"64":2160.1049868766,"65":3808.0708661417,"66":6  
620.0787401575,"67":2000.0,"68":1750.0,"69":6229.9868766404,"70":2430.1181102  
362,"71":9580.0524934383,"72":4018.0446194226,"73":8153.8713910761,"74":5575.  
1312335958,"75":7020.0131233596,"76":6734.9081364829,"77":3450.1312335958,"7  
8":7804.1338582677,"79":2379.9212598425,"80":595.1443569554,"81":8370.0787401  
575}}'
```

## Set the index

Set the index to the Station Name:

```
sierraFebNew = sierraFebNew.set_index("STATION_NAME")
```

In [ ]: #

Out[ ]:

	STATION_NAME	COUNTY	ELEVATION	LATITUDE	LONGITUDE	PRECIPITATION	TEMPERATURE
GROVELAND 2, CA US	Tuolumne	853.4	37.8444	-120.2258	176.02	6.1	
CANYON DAM, CA US	Plumas	1389.9	40.1705	-121.0886	164.08	1.4	
KERN RIVER PH 3, CA US	Kern	823.9	35.7831	-118.4389	67.06	8.9	
DONNER MEMORIAL ST PARK, CA US	Nevada	1809.6	39.3239	-120.2331	167.39	-0.9	
BOWMAN DAM, CA US	Nevada	1641.3	39.4539	-120.6556	276.61	2.9	
...	...	...	...	...	...	...	
PACIFIC HOUSE, CA US	El Dorado	1051.6	38.7583	-120.5030	220.22	NaN	
MAMMOTH LAKES RANGER STATION, CA US	Mono	2378.7	37.6478	-118.9617	NaN	-2.3	
COLFAX, CA US	Placer	725.4	39.0911	-120.9480	207.26	7.3	
COLGATE POWERHOUSE, CA US	Yuba	181.4	39.3308	-121.1922	168.91	NaN	
BODIE CALIFORNIA STATE HISTORIC PARK, CA US	Mono	2551.2	38.2119	-119.0142	39.62	-4.4	

82 rows × 7 columns



## Transpose

See what happens when you transpose `sierraFebNew` with `.transpose()`

In [ ]: #

Out[ ]:

STATION_NAME	GROVELAND 2, CA US	CANYON DAM, CA US	KERN RIVER PH 3, CA US	DONNER MEMORIAL ST PARK, CA US	BOWMAN DAM, CA US	BRUSH CREEK RANGER STATION, CA US	GR
<b>COUNTY</b>	Tuolumne	Plumas	Kern	Nevada	Nevada	Butte	
<b>ELEVATION</b>	853.4	1389.9	823.9	1809.6	1641.3	1085.1	
<b>LATITUDE</b>	37.8444	40.1705	35.7831	39.3239	39.4539	39.6949	
<b>LONGITUDE</b>	-120.2258	-121.0886	-118.4389	-120.2331	-120.6556	-121.3452	-
<b>PRECIPITATION</b>	176.02	164.08	67.06	167.39	276.61	296.16	
<b>TEMPERATURE</b>	6.1	1.4	8.9	-0.9	2.9	NaN	
<b>ELEVATION_FT</b>	2799.868766	4560.03937	2703.08399	5937.007874	5384.84252	3560.03937	660(

7 rows × 82 columns

## sf\_weather\_daily

Make sure you have the "sf\_weather\_daily.csv" file downloaded from iLearn and in the `pdData` folder which is in the folder where this Jupyter notebook resides. (Note you can still path to it as an argument you pass to the function, i.e "C:\Geog625\projects\py...")

▀ We'll create an object called `sf_weather` and load the sf weather csv into a dataframe.

```
#load csv into dataframe
sf_weather = pd.read_csv('pdData/sf_weather_daily.csv')
sf_weather
```

In [ ]: #

Out[ ]:

	date_time	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moone
0	1/1/2019 12:00	11	5	51.8	41.0	11	51.8	3:17 AM	2:10 PM
1	1/2/2019 12:00	10	5	50.0	41.0	9	48.2	4:17 AM	2:47 PM
2	1/3/2019 12:00	11	5	51.8	41.0	10	50.0	5:17 AM	3:28 PM
3	1/4/2019 12:00	12	6	53.6	42.8	11	51.8	6:13 AM	4:13 PM
4	1/5/2019 12:00	13	8	55.4	46.4	13	55.4	7:06 AM	5:02 PM
...	...	...	...	...	...	...	...	...	...
360	12/27/2019 12:00	15	8	59.0	46.4	12	53.6	8:44 AM	6:34 PM
361	12/28/2019 12:00	12	8	53.6	46.4	11	51.8	9:30 AM	7:33 PM
362	12/29/2019 12:00	11	10	51.8	50.0	11	51.8	10:09 AM	8:32 PM
363	12/30/2019 12:00	17	8	62.6	46.4	13	55.4	10:42 AM	9:31 PM
364	12/31/2019 12:00	15	9	59.0	48.2	13	55.4	11:12 AM	10:28 PM

365 rows × 16 columns

Notice how in the dataframe you see kind of a truncated version of all your data. Somewhere around maybe line 4 you see ..., well there is a function you can use to remedy this if you'd like to see all your data at once. We'll be using the Pandas "set\_option" function to expose all rows to this. You can also expose additional columns if you have a lot. I've included sample code below, but commented out the option for width and columns, only provided for reference.

```
#set the data frame display option
pd.set_option('display.max_rows', 5000)
```

Some other options you can set:

```
pd.set_option('display.max_columns', 100)
pd.set_option('display.width', 1000)
```

Make some of these changes and call up your sf\_weather dataframe to see the difference.

In [ ]:

#

Out[ ]:

	date_time	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moone
0	1/1/2019 12:00	11	5	51.8	41.0	11	51.8	3:17 AM	2:10 PM
1	1/2/2019 12:00	10	5	50.0	41.0	9	48.2	4:17 AM	2:47 PM
2	1/3/2019 12:00	11	5	51.8	41.0	10	50.0	5:17 AM	3:28 PM
3	1/4/2019 12:00	12	6	53.6	42.8	11	51.8	6:13 AM	4:13 PM
4	1/5/2019 12:00	13	8	55.4	46.4	13	55.4	7:06 AM	5:02 PM
...	...	...	...	...	...	...	...	...	...
360	12/27/2019 12:00	15	8	59.0	46.4	12	53.6	8:44 AM	6:34 PM
361	12/28/2019 12:00	12	8	53.6	46.4	11	51.8	9:30 AM	7:33 PM
362	12/29/2019 12:00	11	10	51.8	50.0	11	51.8	10:09 AM	8:32 PM
363	12/30/2019 12:00	17	8	62.6	46.4	13	55.4	10:42 AM	9:31 PM
364	12/31/2019 12:00	15	9	59.0	48.2	13	55.4	11:12 AM	10:28 PM

365 rows × 16 columns



- ▀ We can set the row label to the 'date\_time' field with `.set_index()`. Note that you'll still have integer indices, but this provides another way to reference a row, and that row label will stick with the data even after you subset it.

```
sf_weather = sf_weather.set_index('date_time')
sf_weather
```

```
In [ ]: #
Out[ ]: Index(['date_time', 'maxtempC', 'mintempC', 'maxtempF', 'mintempF', 'tempC', 'tempF', 'moonrise', 'moonset', 'sunrise', 'sunset', 'FeelsLikeC', 'WindGustKmph', 'winddirDegree', 'windspeedKmph', 'location'], dtype='object')
Out[ ]: RangeIndex(start=0, stop=365, step=1)
Out[ ]: Index(['1/1/2019 12:00', '1/2/2019 12:00', '1/3/2019 12:00', '1/4/2019 12:00', '1/5/2019 12:00', '1/6/2019 12:00', '1/7/2019 12:00', '1/8/2019 12:00', '1/9/2019 12:00', '1/10/2019 12:00',
               ...
               '12/22/2019 12:00', '12/23/2019 12:00', '12/24/2019 12:00', '12/25/2019 12:00', '12/26/2019 12:00', '12/27/2019 12:00', '12/28/2019 12:00', '12/29/2019 12:00', '12/30/2019 12:00', '12/31/2019 12:00'], dtype='object', name='date_time', length=365)
Out[ ]: pandas.core.frame.DataFrame
```

⌚ What is the datatype of the variable "sf\_weather"?

**Answer:** pandas.core.frame.DataFrame

💻 If we wanted to examine the first five lines of sf\_weather, we could use the ".head()" method.

`sf_weather.head()`

```
In [ ]: #
Out[ ]:
      maxtempC  mintempC  maxtempF  mintempF  tempC  tempF  moonrise  moonset  sun
date_time
1/1/2019 12:00    11       5     51.8     41.0    11     51.8   3:17 AM   2:10 PM
1/2/2019 12:00    10       5     50.0     41.0     9     48.2   4:17 AM   2:47 PM
1/3/2019 12:00    11       5     51.8     41.0    10     50.0   5:17 AM   3:28 PM
1/4/2019 12:00    12       6     53.6     42.8    11     51.8   6:13 AM   4:13 PM
1/5/2019 12:00    13       8     55.4     46.4    13     55.4   7:06 AM   5:02 PM
```

- ▀ How would we display the first 10 lines of sf\_weather? Figure this out by typing

```
help(sf_weather.head)
```

to see what parameters it accepts. There's only one.

- ▀ Then use that information to display the first 10 lines of sf\_weather.

*Note that even though sf\_weather is an object we created, its object type has associated methods.*

In [ ]: #

Out[ ]:

		maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
	date_time									
	1/1/2019 12:00	11	5	51.8	41.0	11	51.8	3:17 AM	2:10 PM	
	1/2/2019 12:00	10	5	50.0	41.0	9	48.2	4:17 AM	2:47 PM	
	1/3/2019 12:00	11	5	51.8	41.0	10	50.0	5:17 AM	3:28 PM	
	1/4/2019 12:00	12	6	53.6	42.8	11	51.8	6:13 AM	4:13 PM	
	1/5/2019 12:00	13	8	55.4	46.4	13	55.4	7:06 AM	5:02 PM	
	1/6/2019 12:00	13	8	55.4	46.4	12	53.6	7:54 AM	5:54 PM	
	1/7/2019 12:00	13	7	55.4	44.6	13	55.4	8:37 AM	6:48 PM	
	1/8/2019 12:00	13	9	55.4	48.2	13	55.4	9:15 AM	7:44 PM	
	1/9/2019 12:00	13	10	55.4	50.0	13	55.4	9:50 AM	8:40 PM	
	1/10/2019 12:00	12	9	53.6	48.2	12	53.6	10:20 AM	9:37 PM	

- ▀ If we wanted to examine the end of the sf\_weather data, then we could use the .tail() method. Try this with sf\_weather using otherwise the same syntax as .head and get the last 10 lines.

In [ ]: #

Out[ ]:

	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sui
date_time									
12/22/2019 12:00	14	10	57.2	50.0	12	53.6	3:33 AM	2:33 PM	
12/23/2019 12:00	13	5	55.4	41.0	11	51.8	4:41 AM	3:11 PM	
12/24/2019 12:00	11	8	51.8	46.4	11	51.8	5:48 AM	3:54 PM	
12/25/2019 12:00	12	6	53.6	42.8	11	51.8	6:52 AM	4:43 PM	
12/26/2019 12:00	14	8	57.2	46.4	12	53.6	7:52 AM	5:37 PM	
12/27/2019 12:00	15	8	59.0	46.4	12	53.6	8:44 AM	6:34 PM	
12/28/2019 12:00	12	8	53.6	46.4	11	51.8	9:30 AM	7:33 PM	
12/29/2019 12:00	11	10	51.8	50.0	11	51.8	10:09 AM	8:32 PM	
12/30/2019 12:00	17	8	62.6	46.4	13	55.4	10:42 AM	9:31 PM	
12/31/2019 12:00	15	9	59.0	48.2	13	55.4	11:12 AM	10:28 PM	

◀ ▶ ►

▀ You can call a series (column) of your dataframe as a property to see the values. Let's say we wanted to see the values of the `maxtempF` column you would use this syntax to access `maxtempF` as a variable `max_temp`:

```
max_temp = sf_weather.maxtempF
max_temp
```

In [ ]: #

Out[ ]: date\_time

1/1/2019 12:00	11
1/2/2019 12:00	10
1/3/2019 12:00	11
1/4/2019 12:00	12
1/5/2019 12:00	13
	..
12/27/2019 12:00	15
12/28/2019 12:00	12
12/29/2019 12:00	11
12/30/2019 12:00	17
12/31/2019 12:00	15

Name: maxtempC, Length: 365, dtype: int64

⌚ What is the *datatype* (hint: `.dtype`) of `max_temp` series just created? :

In [ ]: #

Out[ ]: `dtype('int64')`

**Answer:** `pandas.core.series.Series`

💻 In the following examples, we'll take a closer look at how we can explore the properties of dataframe. For example, if you want to see the number of rows or columns in your dataframe, use the ".shape" property.

`sf_weather.shape`

In [ ]: #

Out[ ]: `(365, 15)`

💻 What if we wanted to see the data types of each of fields? Sometimes this is very useful if you're trying to pass a field to a function that only accepts a certain datatype, for example integer, but you keep encountering errors. Examining the datatypes of your fields could be useful.

`sf_weather.dtypes`

In [ ]: #

Out[ ]: 

<code>maxtempC</code>	<code>int64</code>
<code>mintempC</code>	<code>int64</code>
<code>maxtempF</code>	<code>float64</code>
<code>mintempF</code>	<code>float64</code>
<code>tempC</code>	<code>int64</code>
<code>tempF</code>	<code>float64</code>
<code>moonrise</code>	<code>object</code>
<code>moonset</code>	<code>object</code>
<code>sunrise</code>	<code>object</code>
<code>sunset</code>	<code>object</code>
<code>FeelsLikeC</code>	<code>int64</code>
<code>WindGustKmph</code>	<code>int64</code>
<code>winddirDegree</code>	<code>int64</code>
<code>windspeedKmph</code>	<code>int64</code>
<code>location</code>	<code>object</code>
<code>dtype</code>	<code>object</code>

## Data selection in Pandas

In this section we can explore how to recreate dataframes by selecting only certain parts of your larger dataframe. One thing that's always important to remember that if you want to create a new version of your dataframe that you'll need to assign your operation to a new object.

## Series selection

This example shows how we can create a new dataframe with only the four columns; `maxtempF`, `mintempF`, `location`, and `sunset`. Obviously we'd still have the `date_time` field because it is our index.

```
sf_temp_sunset = sf_weather[['maxtempF', 'mintempF', 'location', 'sunset']]  
sf_temp_sunset
```

In [ ]: #

Out[ ]:

	maxtempC	mintempC	location	sunset
date_time				
1/1/2019 12:00	11	5	san_francisco	5:02 PM
1/2/2019 12:00	10	5	san_francisco	5:02 PM
1/3/2019 12:00	11	5	san_francisco	5:03 PM
1/4/2019 12:00	12	6	san_francisco	5:04 PM
1/5/2019 12:00	13	8	san_francisco	5:05 PM
...	...	...	...	...
12/27/2019 12:00	15	8	san_francisco	4:58 PM
12/28/2019 12:00	12	8	san_francisco	4:58 PM
12/29/2019 12:00	11	10	san_francisco	4:59 PM
12/30/2019 12:00	17	8	san_francisco	5:00 PM
12/31/2019 12:00	15	9	san_francisco	5:01 PM

365 rows × 4 columns

Create a new dataframe called `sf_mintemp_moon` that only contains the columns: `date_time`, `mintempC`, `tempC`, `moonrise`, and `location`. However, don't actually include `date_time` in your request since it's the named index and will already be there, and in fact create an error if you include it.

In [ ]: #

Out[ ]:

	date_time	mintempC	tempC	moonrise	location
	1/1/2019 12:00	5	11	3:17 AM	san_francisco
	1/2/2019 12:00	5	9	4:17 AM	san_francisco
	1/3/2019 12:00	5	10	5:17 AM	san_francisco
	1/4/2019 12:00	6	11	6:13 AM	san_francisco
	1/5/2019 12:00	8	13	7:06 AM	san_francisco
	...	...	...	...	...
	12/27/2019 12:00	8	12	8:44 AM	san_francisco
	12/28/2019 12:00	8	11	9:30 AM	san_francisco
	12/29/2019 12:00	10	11	10:09 AM	san_francisco
	12/30/2019 12:00	8	13	10:42 AM	san_francisco
	12/31/2019 12:00	9	13	11:12 AM	san_francisco

365 rows × 4 columns

## Selecting rows using indexing

.loc[] and .iloc[] are two very common methods of selecting data within a dataframe by indexing.

### Using loc to use named indices and iloc for numeric indices

loc uses the label of the column or the label of the row and iloc uses the index of the row or the index of the column. In this example below we're accessing the row with index date January 1st, 2019.

```
sf_weather.loc['1/1/2019 12:00']
```

In [ ]: #

```
Out[ ]: maxtempC           11
         mintempC          5
         maxtempF          51.8
         mintempF          41.0
         tempC              11
         tempF              51.8
         moonrise           3:17 AM
         moonset            2:10 PM
         sunrise             7:25 AM
         sunset              5:02 PM
         FeelsLikeC         11
         WindGustKmph        22
         winddirDegree       49
         windspeedKmph        12
         location            san_francisco
Name: 1/1/2019 12:00, dtype: object
```

▀ iloc alternative, getting a range of records.

```
print(sf_weather.iloc[1:4])
```

In [ ]: #

Out[ ]:

	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
--	----------	----------	----------	----------	-------	-------	----------	---------	-----

date_time	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
1/2/2019 12:00	10	5	50.0	41.0	9	48.2	4:17 AM	2:47 PM	
1/3/2019 12:00	11	5	51.8	41.0	10	50.0	5:17 AM	3:28 PM	
1/4/2019 12:00	12	6	53.6	42.8	11	51.8	6:13 AM	4:13 PM	

▀ You can also specify a range with named indices using loc.

```
Feb = sf_weather.loc['2/1/2019 12:00':'2/28/2019 12:00']
```

Then reference part of this subset with `iloc`, and we'll see that we have new integer indices, while the row labels stick with what they were, illustrating why creating row labels with `.set_index` was useful.

In [ ]: #

Out[ ]:

date_time	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
2/1/2019 12:00	13	10	55.4	50.0	13	55.4	5:02 AM	2:58 PM	
2/2/2019 12:00	12	10	53.6	50.0	12	53.6	5:51 AM	3:49 PM	
2/3/2019 12:00	12	10	53.6	50.0	12	53.6	6:35 AM	4:42 PM	
2/4/2019 12:00	9	8	48.2	46.4	9	48.2	7:15 AM	5:38 PM	
2/5/2019 12:00	10	6	50.0	42.8	10	50.0	7:51 AM	6:33 PM	
2/6/2019 12:00	10	5	50.0	41.0	9	48.2	8:22 AM	7:30 PM	
2/7/2019 12:00	10	5	50.0	41.0	10	50.0	8:52 AM	8:26 PM	
2/8/2019 12:00	11	7	51.8	44.6	10	50.0	9:21 AM	9:23 PM	
2/9/2019 12:00	11	8	51.8	46.4	11	51.8	9:49 AM	10:20 PM	
2/10/2019 12:00	10	7	50.0	44.6	9	48.2	10:17 AM	11:19 PM	
2/11/2019 12:00	11	6	51.8	42.8	10	50.0	10:47 AM	No moonset	
2/12/2019 12:00	11	7	51.8	44.6	11	51.8	11:21 AM	12:19 AM	
2/13/2019 12:00	14	11	57.2	51.8	14	57.2	11:59 AM	1:22 AM	
2/14/2019 12:00	14	11	57.2	51.8	11	51.8	12:43 PM	2:26 AM	
2/15/2019 12:00	11	9	51.8	48.2	11	51.8	1:37 PM	3:31 AM	
2/16/2019 12:00	11	9	51.8	48.2	10	50.0	2:38 PM	4:33 AM	
2/17/2019 12:00	9	7	48.2	44.6	9	48.2	3:47 PM	5:32 AM	
2/18/2019 12:00	12	5	53.6	41.0	10	50.0	5:00 PM	6:24 AM	
2/19/2019 12:00	11	5	51.8	41.0	10	50.0	6:15 PM	7:11 AM	
2/20/2019 12:00	11	6	51.8	42.8	11	51.8	7:29 PM	7:51 AM	
2/21/2019 12:00	11	6	51.8	42.8	10	50.0	8:40 PM	8:28 AM	
2/22/2019 12:00	10	5	50.0	41.0	9	48.2	9:50 PM	9:03 AM	

date_time	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
2/23/2019 12:00	12	5	53.6	41.0	11	51.8	10:57 PM	9:37 AM	↑
2/24/2019 12:00	11	8	51.8	46.4	11	51.8	No moonrise	10:11 AM	↑
2/25/2019 12:00	11	4	51.8	39.2	10	50.0	12:01 AM	10:47 AM	↑
2/26/2019 12:00	12	3	53.6	37.4	12	53.6	1:04 AM	11:26 AM	↑
2/27/2019 12:00	14	9	57.2	48.2	14	57.2	2:03 AM	12:08 PM	↑
2/28/2019 12:00	14	10	57.2	50.0	13	55.4	2:58 AM	12:55 PM	↑

▀ We'll do some of this with the simpler sierra data

sierra

In [ ]: #

Out[ ]:

	temperature	elevation	latitude
Oroville	10.7	52	39.52
Auburn	9.7	394	38.91
Sonora	7.7	510	37.97
Placerville	9.2	564	38.70
Colfax	7.3	725	39.09
Nevada City	6.7	848	39.25
Quincy	4.0	1042	39.94
Yosemite	5.0	1225	37.75
Sierraville	0.9	1516	40.35
Truckee	-1.1	1775	39.33
Tahoe City	-0.8	1899	39.17
Bodie	-4.4	2551	38.21

▀ How would you get the sierra data from 'Sonora' to 'Colfax'?

In [ ]: #

Out[ ]:

	temperature	elevation	latitude
Sonora	7.7	510	37.97
Placerville	9.2	564	38.70
Colfax	7.3	725	39.09

- ▀ If you wanted to select multiple rows specifically (such as the first day of the month) then you can pass those row indices. Note that the list is a single input to `.loc`.

```
sf_weather.loc[['1/1/2019 12:00','2/1/2019 12:00','3/1/2019 12:00']]
```

In [ ]: #

Out[ ]:

	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
date_time									
1/1/2019 12:00	11	5	51.8	41.0	11	51.8	3:17 AM	2:10 PM	
2/1/2019 12:00	13	10	55.4	50.0	13	55.4	5:02 AM	2:58 PM	
3/1/2019 12:00	12	8	53.6	46.4	12	53.6	3:49 AM	1:45 PM	



- ▀ Using the same method, get the sierra data for Sonora and Sierraville.

In [ ]: #

Out[ ]:

	temperature	elevation	latitude
Sonora	7.7	510	37.97
Sierraville	0.9	1516	40.35

- ▀ Back to `sf_weather`, the first set of labels you pass will only return the rows you're interested in, but if you wanted to refine your data data, you can also pass a set of column labels as the second argument:

```
sf_weather.loc['1/1/2019 12:00':'1/3/2019 12:00',['maxtempC','mintempC','location']]
```

In [ ]: #

Out[ ]:

	maxtempC	mintempC	location
date_time			
1/1/2019 12:00	11	5	san_francisco
1/2/2019 12:00	10	5	san_francisco
1/3/2019 12:00	11	5	san_francisco

▀▀ Create a dataframe that contains data for July 1-5. Show only the columns `mintempF` , `location` , `moonrise` , and `WindGustKmph` .

In [ ]: #

Out[ ]:

	mintempF	location	moonrise	WindGustKmph
date_time				
7/1/2019 12:00	57.2	san_francisco	3:50 AM	16
7/2/2019 12:00	59.0	san_francisco	4:44 AM	16
7/3/2019 12:00	57.2	san_francisco	5:46 AM	13
7/4/2019 12:00	57.2	san_francisco	6:54 AM	12
7/5/2019 12:00	55.4	san_francisco	8:06 AM	13

▀▀ Next, we can explore the `.iloc[]` method to select data. To differentiate these two methods I like to think of the "i" in the `iloc` method as "index", meaning we need to pass an index number to return the row or column. First set of indexes are rows and second set of indexes are columns:

```
sf_weather.iloc[0]
```

In [ ]: #

```
Out[ ]: maxtempC           11
         mintempC          5
         maxtempF          51.8
         mintempF          41.0
         tempC              11
         tempF              51.8
         moonrise            3:17 AM
         moonset             2:10 PM
         sunrise             7:25 AM
         sunset              5:02 PM
         FeelsLikeC          11
         WindGustKmph        22
         winddirDegree       49
         windspeedKmph        12
         location             san_francisco
Name: 1/1/2019 12:00, dtype: object
```

Let's see how we can get the first 5 rows of our data from, but using the `.iloc[]` method:

```
sf_weather.iloc[:5]
```

In [ ]: #

Out[ ]:

	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
--	----------	----------	----------	----------	-------	-------	----------	---------	-----

date_time	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
1/1/2019 12:00	11	5	51.8	41.0	11	51.8	3:17 AM	2:10 PM	
1/2/2019 12:00	10	5	50.0	41.0	9	48.2	4:17 AM	2:47 PM	
1/3/2019 12:00	11	5	51.8	41.0	10	50.0	5:17 AM	3:28 PM	
1/4/2019 12:00	12	6	53.6	42.8	11	51.8	6:13 AM	4:13 PM	
1/5/2019 12:00	13	8	55.4	46.4	13	55.4	7:06 AM	5:02 PM	

Now, let's see how we can return the first 3 rows and the first 10 columns in the dataframe:

```
sf_weather.iloc[[0,1,2],:10]
```

In [ ]: #

Out[ ]:

	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
--	----------	----------	----------	----------	-------	-------	----------	---------	-----

date\_time

1/1/2019  
12:00

11

5

51.8

41.0

11

51.8

3:17 AM

2:10 PM

1/2/2019  
12:00

10

5

50.0

41.0

9

48.2

4:17 AM

2:47 PM

1/3/2019  
12:00

11

5

51.8

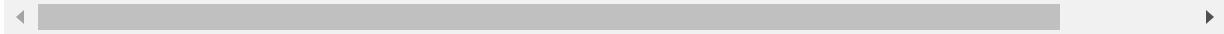
41.0

10

50.0

5:17 AM

3:28 PM



Now create the *last* 3 rows and the *last* 10 columns

In [ ]: #

Out[ ]:

	tempF	moonrise	moonset	sunrise	sunset	FeelsLikeC	WindGustKmph	winddirDegr
--	-------	----------	---------	---------	--------	------------	--------------	-------------

date\_time

12/29/2019  
12:00

51.8

10:09 AM

8:32 PM

7:25  
AM4:59  
PM

10

14

1

12/30/2019  
12:00

55.4

10:42 AM

9:31 PM

7:25  
AM5:00  
PM

13

4

3

12/31/2019  
12:00

55.4

11:12 AM

10:28  
PM7:25  
AM5:01  
PM

13

7

1



As you can see above, using the iloc method might be a little easier to select multiple rows/columns by using the slicing of indexes method. This way we wouldn't have to pass each column or row name by label.

Next, let's create a smaller dataframe by assigning the first twenty rows of sf\_weather with .head(n=20) to begin\_sf\_weather ...

In [ ]: #

Out[ ]:

date_time	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
1/1/2019 12:00	11	5	51.8	41.0	11	51.8	3:17 AM	2:10 PM	
1/2/2019 12:00	10	5	50.0	41.0	9	48.2	4:17 AM	2:47 PM	
1/3/2019 12:00	11	5	51.8	41.0	10	50.0	5:17 AM	3:28 PM	
1/4/2019 12:00	12	6	53.6	42.8	11	51.8	6:13 AM	4:13 PM	
1/5/2019 12:00	13	8	55.4	46.4	13	55.4	7:06 AM	5:02 PM	
1/6/2019 12:00	13	8	55.4	46.4	12	53.6	7:54 AM	5:54 PM	
1/7/2019 12:00	13	7	55.4	44.6	13	55.4	8:37 AM	6:48 PM	
1/8/2019 12:00	13	9	55.4	48.2	13	55.4	9:15 AM	7:44 PM	
1/9/2019 12:00	13	10	55.4	50.0	13	55.4	9:50 AM	8:40 PM	
1/10/2019 12:00	12	9	53.6	48.2	12	53.6	10:20 AM	9:37 PM	
1/11/2019 12:00	12	8	53.6	46.4	12	53.6	10:50 AM	10:33 PM	
1/12/2019 12:00	13	8	55.4	46.4	12	53.6	11:18 AM	11:30 PM	
1/13/2019 12:00	13	8	55.4	46.4	12	53.6	11:46 AM	No moonset	
1/14/2019 12:00	12	8	53.6	46.4	12	53.6	12:15 PM	12:28 AM	
1/15/2019 12:00	11	11	51.8	51.8	11	51.8	12:48 PM	1:29 AM	
1/16/2019 12:00	13	9	55.4	48.2	13	55.4	1:24 PM	2:32 AM	
1/17/2019 12:00	13	10	55.4	50.0	13	55.4	2:07 PM	3:38 AM	
1/18/2019 12:00	12	8	53.6	46.4	12	53.6	2:57 PM	4:45 AM	
1/19/2019 12:00	15	9	59.0	48.2	15	59.0	3:56 PM	5:52 AM	
1/20/2019 12:00	13	10	55.4	50.0	12	53.6	5:03 PM	6:54 AM	

... and write it out to a csv:

```
begin_sf_weather.to_csv("pdData/begin_sf_weather.csv")
```

In [ ]: #

Using either the `.loc[]` method or the `.iloc[]` method create a CSV (using the `.to_csv` method) from the `sf_weather` dataframe that contains rows 15-20 ( `15:21` ) and the first 10 columns. Name the output `ten1520.csv` .

Remember to save into the `pdData` folder as we did above.

In [ ]: #

## Selecting columns or rows?

The syntax for selecting columns looks a lot like selection rows. The key difference in the methods we've just looked at is if you're using

- `.loc / .iloc` for rows
- or not, for columns

The following examples illustrate this. First we'll create a simple DataFrame with x, y, z columns and a, b, c rows:

```
df = pd.DataFrame({"x":pd.Series({'a':1,'b':2,'c':3}),
                   "y":pd.Series({'a':4,'b':5,'c':6}),
                   "z":pd.Series({'a':7,'b':8,'c':9})})
df
```

In [ ]: #

Out[ ]:

	x	y	z
<b>a</b>	1	4	7
<b>b</b>	2	5	8
<b>c</b>	3	6	9

Now we'll select two columns with `df[['x','y']]`

```
In [ ]: #
```

Out[ ]:

	x	y
a	1	4
b	2	5
c	3	6

▀▀ Then select two rows with `df.loc[['b', 'c']]`

```
In [ ]: #
```

Out[ ]:

	x	y	z
b	2	5	8
c	3	6	9

As you can see above, the methods do look a lot alike, but working with rows uses some form of `Loc`.

▀▀ We could even do both to get an `intersection` of the data structure, with

```
df[['x', 'y']].loc[['b', 'c']]
```

```
In [ ]: #
```

Out[ ]:

	x	y
b	2	5
c	3	6

## Descriptive statistics

We can use various functions exposed to us through pandas to run get descriptive statistics for our *series*. A list of some of these statistics, most of which are obvious what they do:

- count
- sum
- mean
- median
- min
- max
- mode : returns a series of modes
- prod : product
- mad mean absolute deviation
- sem standard error of the mean
- std sample standard deviation
- var
- sem standard error of the mean
- skew skewness (3rd moment)
- kurt kurtosis (4th moment)
- cumsum cumulative sum
- cumprod
- cummax
- cummin

▀ First, let's create a series with modes and get its mean with:

```
aSeries = pd.Series([1,3,3,5,7,7,9,11,11,13])  
aSeries.mean()
```

... and also get the **mode** (which may return multiple values).

```
In [ ]: #
```

```
Out[ ]: 7.0
```

```
Out[ ]: 0      3  
        1      7  
        2     11  
       dtype: int64
```

▀ Then derive some other summary statistics.

```
In [ ]: #
```

```
Out[ ]: 1.2649110640673518
```

```
In [ ]: #
```

```
Out[ ]: 4.0
```

```
In [ ]: #
```

```
Out[ ]: 0      1
        1      4
        2      7
        3     12
        4     19
        5     26
        6     35
        7     46
        8     57
        9     70
       dtype: int64
```

All of the above were for a single series.

- ▀ If we use these methods with a DataFrame, all series are described using the specified statistic:

```
sf_weather.mean()
```

```
In [ ]: #
```

```
C:\Users\900008452\AppData\Local\Temp\ipykernel_24480\513922073.py:2: FutureWarning: Dropping of nuisance columns in DataFrame reductions (with 'numeric_only=None') is deprecated; in a future version this will raise TypeError. Select only valid columns before calling the reduction.
sf_weather.mean()
```

```
Out[ ]: maxtempC      18.150685
        mintempC     12.600000
        maxtempF      64.671233
        mintempF      54.680000
        tempC         17.353425
        tempF         63.236164
       FeelsLikeC    17.391781
        WindGustKmph   14.860274
        winddirDegree  203.501370
        windspeedKmph   11.863014
       dtype: float64
```

- ▀ Do the same for standard deviation (std):

```
In [ ]: #
```

```
C:\Users\900008452\AppData\Local\Temp\ipykernel_24480\194938136.py:2: FutureWarning: Dropping of nuisance columns in DataFrame reductions (with 'numeric_only=None') is deprecated; in a future version this will raise TypeError. Select only valid columns before calling the reduction.  
sf_weather.std()
```

```
Out[ ]: maxtempC      4.673608  
mintempC      3.796904  
maxtempF      8.412495  
mintempF      6.834428  
tempC         4.599752  
tempF         8.279553  
FeelsLikeC    5.008502  
WindGustKmph   9.659573  
winddirDegree  83.522506  
windspeedKmph  5.994308  
dtype: float64
```

- ▀ If we pass just the column to the function we can get the values returned on the column. For example, if I wanted to see the max time for sunset I would pass this function:

```
sf_weather['sunset'].max()
```

```
In [ ]: #
```

```
Out[ ]: '7:36 PM'
```

- ▀ Then to see the earliest sunset all year we'd use `min` instead

```
In [ ]: #
```

```
Out[ ]: '4:50 PM'
```

- ▀ What if we wanted to see the mean and standard deviation of lowest temperature all year:

```
In [ ]: #
```

```
Out[ ]: (54.67999999999999, 6.83442803703474)
```

- ▀ Now use the `.describe()` method on the same DataFrame series to show all these statistics for `mintempF`. (Hint: the command looks the same as deriving the `.mean()` or `.std()`, just uses `.describe()` .

```
In [ ]: #
```

```
Out[ ]: count    365.000000
         mean     54.680000
         std      6.834428
         min     37.400000
         25%    50.000000
         50%    55.400000
         75%    59.000000
         max     73.400000
Name: mintempF, dtype: float64
```

## Creating graphs with Pandas

There are some advanced operations with generating graphs using some other libraries, but Pandas provides some basic graphing capabilities right out of the box without needing to import additional libraries.

Below is a simple example of generating a new dataframe that contains only the January data from our sf\_weather dataframe and then plots it:

```
jan_sf_weather = sf_weather.head(n=31)
jan_sf_weather
```

In [ ]: #

Out[ ]:

date_time	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
1/1/2019 12:00	11	5	51.8	41.0	11	51.8	3:17 AM	2:10 PM	
1/2/2019 12:00	10	5	50.0	41.0	9	48.2	4:17 AM	2:47 PM	
1/3/2019 12:00	11	5	51.8	41.0	10	50.0	5:17 AM	3:28 PM	
1/4/2019 12:00	12	6	53.6	42.8	11	51.8	6:13 AM	4:13 PM	
1/5/2019 12:00	13	8	55.4	46.4	13	55.4	7:06 AM	5:02 PM	
1/6/2019 12:00	13	8	55.4	46.4	12	53.6	7:54 AM	5:54 PM	
1/7/2019 12:00	13	7	55.4	44.6	13	55.4	8:37 AM	6:48 PM	
1/8/2019 12:00	13	9	55.4	48.2	13	55.4	9:15 AM	7:44 PM	
1/9/2019 12:00	13	10	55.4	50.0	13	55.4	9:50 AM	8:40 PM	
1/10/2019 12:00	12	9	53.6	48.2	12	53.6	10:20 AM	9:37 PM	
1/11/2019 12:00	12	8	53.6	46.4	12	53.6	10:50 AM	10:33 PM	
1/12/2019 12:00	13	8	55.4	46.4	12	53.6	11:18 AM	11:30 PM	
1/13/2019 12:00	13	8	55.4	46.4	12	53.6	11:46 AM	No moonset	
1/14/2019 12:00	12	8	53.6	46.4	12	53.6	12:15 PM	12:28 AM	
1/15/2019 12:00	11	11	51.8	51.8	11	51.8	12:48 PM	1:29 AM	
1/16/2019 12:00	13	9	55.4	48.2	13	55.4	1:24 PM	2:32 AM	
1/17/2019 12:00	13	10	55.4	50.0	13	55.4	2:07 PM	3:38 AM	
1/18/2019 12:00	12	8	53.6	46.4	12	53.6	2:57 PM	4:45 AM	
1/19/2019 12:00	15	9	59.0	48.2	15	59.0	3:56 PM	5:52 AM	
1/20/2019 12:00	13	10	55.4	50.0	12	53.6	5:03 PM	6:54 AM	
1/21/2019 12:00	12	8	53.6	46.4	12	53.6	6:15 PM	7:50 AM	
1/22/2019 12:00	13	8	55.4	46.4	12	53.6	7:30 PM	8:39 AM	

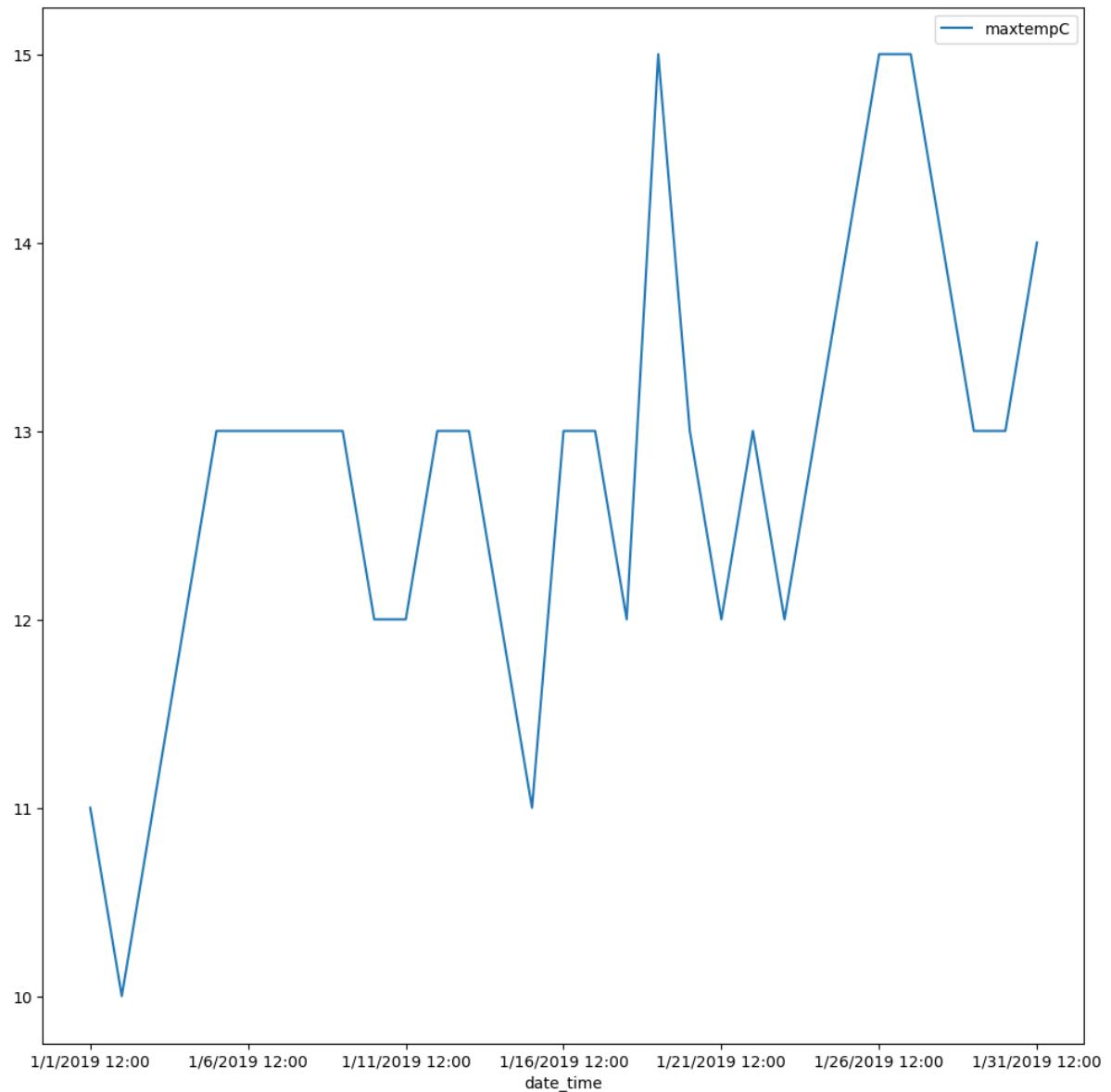
	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
date_time									
1/23/2019 12:00	12	7	53.6	44.6	12	53.6	8:42 PM	9:22 AM	
1/24/2019 12:00	13	9	55.4	48.2	12	53.6	9:53 PM	9:59 AM	
1/25/2019 12:00	14	7	57.2	44.6	13	55.4	11:01 PM		10:33 AM
1/26/2019 12:00	15	9	59.0	48.2	14	57.2	No moonrise		11:06 AM
1/27/2019 12:00	15	8	59.0	46.4	14	57.2	12:06 AM		11:39 AM
1/28/2019 12:00	14	10	57.2	50.0	13	55.4	1:10 AM		12:12 PM
1/29/2019 12:00	13	10	55.4	50.0	13	55.4	2:11 AM		12:49 PM
1/30/2019 12:00	13	10	55.4	50.0	13	55.4	3:11 AM		1:28 PM
1/31/2019 12:00	14	11	57.2	51.8	13	55.4	4:08 AM		2:11 PM

Plot the data setting the y axis to the maxtempF field from our data frame:

```
jan_sf_weather.plot(y = ['maxtempF'], figsize = (15,15))
```

```
In [ ]: #
```

```
Out[ ]: <AxesSubplot: xlabel='date_time'>
```



Plot the mintempF for the month of February.

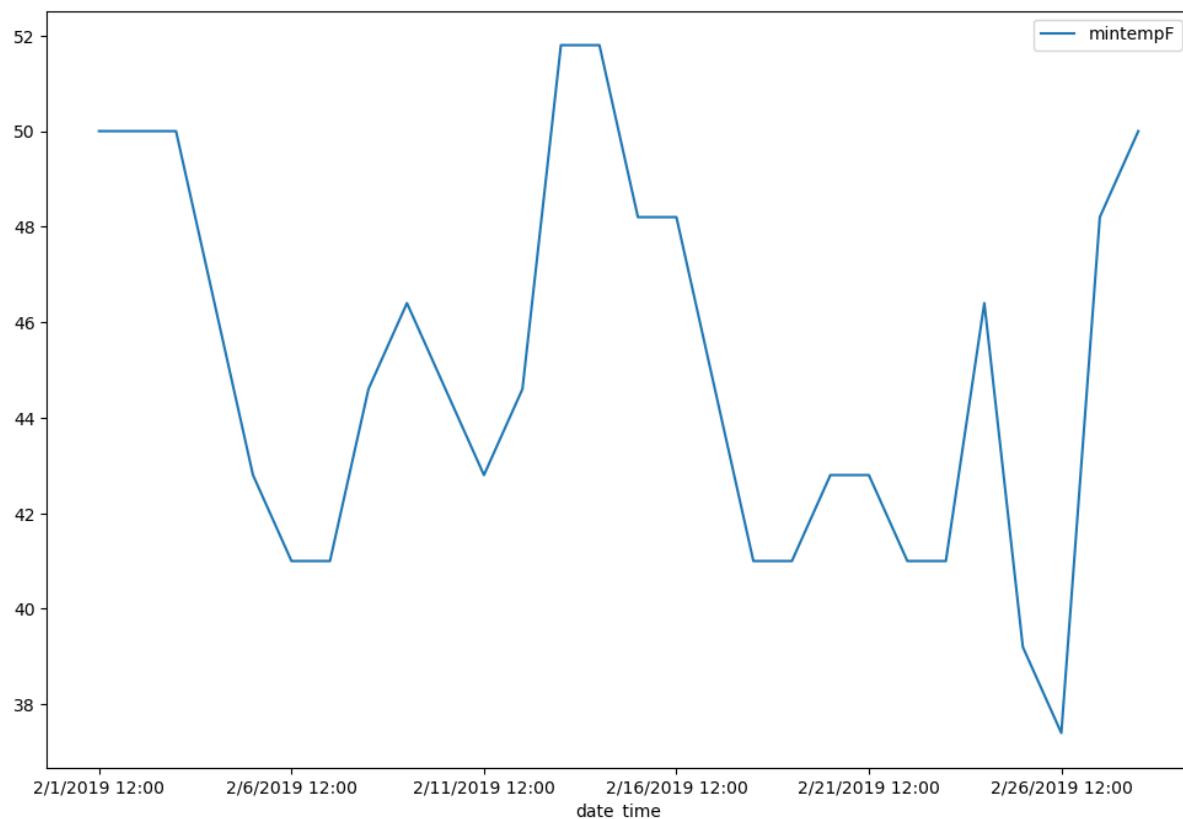
In [ ]: #

Out[ ]:

date_time	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
2/1/2019 12:00	13	10	55.4	50.0	13	55.4	5:02 AM	2:58 PM	
2/2/2019 12:00	12	10	53.6	50.0	12	53.6	5:51 AM	3:49 PM	
2/3/2019 12:00	12	10	53.6	50.0	12	53.6	6:35 AM	4:42 PM	
2/4/2019 12:00	9	8	48.2	46.4	9	48.2	7:15 AM	5:38 PM	
2/5/2019 12:00	10	6	50.0	42.8	10	50.0	7:51 AM	6:33 PM	
2/6/2019 12:00	10	5	50.0	41.0	9	48.2	8:22 AM	7:30 PM	
2/7/2019 12:00	10	5	50.0	41.0	10	50.0	8:52 AM	8:26 PM	
2/8/2019 12:00	11	7	51.8	44.6	10	50.0	9:21 AM	9:23 PM	
2/9/2019 12:00	11	8	51.8	46.4	11	51.8	9:49 AM	10:20 PM	
2/10/2019 12:00	10	7	50.0	44.6	9	48.2	10:17 AM	11:19 PM	
2/11/2019 12:00	11	6	51.8	42.8	10	50.0	10:47 AM	No moonset	
2/12/2019 12:00	11	7	51.8	44.6	11	51.8	11:21 AM	12:19 AM	
2/13/2019 12:00	14	11	57.2	51.8	14	57.2	11:59 AM	1:22 AM	
2/14/2019 12:00	14	11	57.2	51.8	11	51.8	12:43 PM	2:26 AM	
2/15/2019 12:00	11	9	51.8	48.2	11	51.8	1:37 PM	3:31 AM	
2/16/2019 12:00	11	9	51.8	48.2	10	50.0	2:38 PM	4:33 AM	
2/17/2019 12:00	9	7	48.2	44.6	9	48.2	3:47 PM	5:32 AM	
2/18/2019 12:00	12	5	53.6	41.0	10	50.0	5:00 PM	6:24 AM	
2/19/2019 12:00	11	5	51.8	41.0	10	50.0	6:15 PM	7:11 AM	
2/20/2019 12:00	11	6	51.8	42.8	11	51.8	7:29 PM	7:51 AM	
2/21/2019 12:00	11	6	51.8	42.8	10	50.0	8:40 PM	8:28 AM	
2/22/2019 12:00	10	5	50.0	41.0	9	48.2	9:50 PM	9:03 AM	

	maxtempC	mintempC	maxtempF	mintempF	tempC	tempF	moonrise	moonset	sun
date_time									
2/23/2019 12:00	12	5	53.6	41.0	11	51.8	10:57 PM	9:37 AM	1
2/24/2019 12:00	11	8	51.8	46.4	11	51.8	No moonrise	10:11 AM	1
2/25/2019 12:00	11	4	51.8	39.2	10	50.0	12:01 AM	10:47 AM	1
2/26/2019 12:00	12	3	53.6	37.4	12	53.6	1:04 AM	11:26 AM	1
2/27/2019 12:00	14	9	57.2	48.2	14	57.2	2:03 AM	12:08 PM	1
2/28/2019 12:00	14	10	57.2	50.0	13	55.4	2:58 AM	12:55 PM	1

Out[ ]: <AxesSubplot: xlabel='date\_time'>



## key

- ➔ This directs you to do something specific, maybe in the operating system or answer something conceptual.
- ☒ Code to just run, typically boilerplate.
- 💻 Coding you need to write, in the subsequent code cell.
- ❓ Questions to answer in the same markdown cell.
- Prompt for an interpretation or the answer to a question.

In [ ]:

# Exercise 05: Pandas Data Transformation

In the previous exercise we were introduced to the Pandas Python library and we explored the various ways of loading data into a dataframe and accessing certain rows and columns from that dataframe to create new dataframes. In this exercise we'll explore some more advanced methods in the Pandas library for accessing and manipulating data.

We'll also look at the various methods for combining and merging multiple dataframes.

Concepts covered in the exercise include:

- Adding and dropping columns and rows
- Working with null values in a dataframe
- Filtering dataframes with various queries
- Iterating over dataframes (looping over values)
- Group by, sum, sort and assign methods
- Join, Merge, and Concat multiple dataframes

## Data we'll be exploring

For this exercise, we'll make use of some open data from the City & County of San Francisco hosted at <https://datasf.org/opendata/>. Some former students from our program who've gone on to work for the city have been instrumental in making these data available. This is a great resource for your own research and coursework, and was suggested by another former student from our program, Dara O'Beirne, who's now working for the state in Sacramento and sometimes teaches this class and this lab is based on one he created for the class.

From that data source, we'll look at Bay Area unemployment data and we'll work on manipulating that data to be able to analyze questions, however we won't focus as much on the questions (those are for you to consider) as much as on the transformation methods. We'll loosely look at transformations that help to understand the effect of COVID on unemployment (and one of the variables is pandemic specific), but there's a lot more you can do with this and other data at the site. In the geopandas lab, we'll look at 311 data from that same source to look at parking incidents by neighborhood, and rely on latitude and longitude fields to map them.

## Dataframe Row/Column Manipulation and Filtering

Let's begin by importing both the pandas and matplotlib libraries:

```
import pandas as pd  
import matplotlib.pyplot as plt
```

In [ ]: #

## Parsing Dates

We'll start by simply reading unemployment data, including COVID-related claims, without any special settings or transformations. Explore information about this dataset at <https://data.sfgov.org/Economy-and-Community/Unemployment-Insurance-Weekly-Claims-for-Bay-Area-/d98w-yij4> (<https://data.sfgov.org/Economy-and-Community/Unemployment-Insurance-Weekly-Claims-for-Bay-Area-/d98w-yij4>) where you'll see that:

- UI\_Claims are the "Number of new weekly Unemployment Insurance (UI) claims filed with EDD (Includes new, additional, transitional, and PUA claims)"
  - EDD: California Economic Development Department
- PUA\_Claims : "Breakout of the number of new weekly Pandemic Unemployment Assistance (PUA) claims filed [sic] with EDD."



```
bayArea_unemployment_noparse = pd.read_csv('pdData/Unemployment_BA_Counties_2020_202105.csv')
bayArea_unemployment_noparse
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
0	1/11/2020	Alameda	1487	0
1	1/11/2020	Contra Costa	1073	0
2	1/11/2020	Marin	144	0
3	1/11/2020	Napa	162	0
4	1/11/2020	San Francisco	945	0
...	...	...	...	...
652	5/29/2021	San Francisco	3878	233
653	5/29/2021	San Mateo	2346	140
654	5/29/2021	Santa Clara	5307	419
655	5/29/2021	Solano	1580	140
656	5/29/2021	Sonoma	1410	126

657 rows × 4 columns

Next, we'll create a dataframe that contains the unemployment claims filed during Covid, for each county in the Bay Area, but parse the dates as dates.

```
bayArea_unemployment = pd.read_csv('pdData/Unemployment_BA_Counties_2020_202105.csv', parse_dates= ["Week_Ending"])
bayArea_unemployment
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
0	2020-01-11	Alameda	1487	0
1	2020-01-11	Contra Costa	1073	0
2	2020-01-11	Marin	144	0
3	2020-01-11	Napa	162	0
4	2020-01-11	San Francisco	945	0
...	...	...	...	...
652	2021-05-29	San Francisco	3878	233
653	2021-05-29	San Mateo	2346	140
654	2021-05-29	Santa Clara	5307	419
655	2021-05-29	Solano	1580	140
656	2021-05-29	Sonoma	1410	126

657 rows × 4 columns

Q Why do we pass "Week\_Ending" to the parse\_dates parameter?

**Answer:** We want to use Week\_Ending as a date. Without this, it's just a string.

## Dropping columns and rows

Earlier, we selected columns using a list desired, and used .iloc and .loc to select rows. Similarly, we can use .drop to drop either columns or rows, but here we use We used .select and The .drop method. Once we've loaded all the data into the dataframe, let's drop a column from the dataframe. I'll create a new dataframe called "bay\_unemployment\_noPUA", that doesn't contain the dropped column:

```
bayArea_unemployment_noPUA = bayArea_unemployment.drop(columns = ["PUA_Claims"])
bayArea_unemployment_noPUA
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims
0	2020-01-11	Alameda	1487
1	2020-01-11	Contra Costa	1073
2	2020-01-11	Marin	144
3	2020-01-11	Napa	162
4	2020-01-11	San Francisco	945
...	...	...	...
652	2021-05-29	San Francisco	3878
653	2021-05-29	San Mateo	2346
654	2021-05-29	Santa Clara	5307
655	2021-05-29	Solano	1580
656	2021-05-29	Sonoma	1410

657 rows × 3 columns

- Notice that if we re-examine the original dataframe it still contains the columns we dropped, remember it's important to reassign that operation you do onto a new dataframe:

bayArea\_unemployment

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
0	2020-01-11	Alameda	1487	0
1	2020-01-11	Contra Costa	1073	0
2	2020-01-11	Marin	144	0
3	2020-01-11	Napa	162	0
4	2020-01-11	San Francisco	945	0
...	...	...	...	...
652	2021-05-29	San Francisco	3878	233
653	2021-05-29	San Mateo	2346	140
654	2021-05-29	Santa Clara	5307	419
655	2021-05-29	Solano	1580	140
656	2021-05-29	Sonoma	1410	126

657 rows × 4 columns

## Drop a row using an index

First let's check the syntax usage for `.drop` by typing...

```
bayArea_unemployment.drop
```

and pressing shift-tab when the cursor is at the end of `.drop`.

(To get more detailed information, put the above into `help()`)

... then noting that `axis=0` (rows) is the default, just provide the numeric index `0` as the sole input to `bayArea_unemployment.drop(...)`, even though it may be confusing to say that the index `0` is a *label* (remember that if the parameter isn't specified, the first thing provided is assigned to the first parameter, which in this case is `labels`.)

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
1	2020-01-11	Contra Costa	1073	0
2	2020-01-11	Marin	144	0
3	2020-01-11	Napa	162	0
4	2020-01-11	San Francisco	945	0
5	2020-01-11	San Mateo	467	0
...	...	...	...	...
652	2021-05-29	San Francisco	3878	233
653	2021-05-29	San Mateo	2346	140
654	2021-05-29	Santa Clara	5307	419
655	2021-05-29	Solano	1580	140
656	2021-05-29	Sonoma	1410	126

656 rows × 4 columns

Checking the usage again with shift-tab, and noting the `index` parameter which applies to rows, we can see that we can use the `.drop()` method to drop a list of rows, so use this to drop `index = [0,1,2,3]`:

```
bayArea_unemployment_drop4 = bayArea_unemployment.drop(index = [0,1,2,3])
bayArea_unemployment_drop4
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
4	2020-01-11	San Francisco	945	0
5	2020-01-11	San Mateo	467	0
6	2020-01-11	Santa Clara	1443	0
7	2020-01-11	Solano	678	0
8	2020-01-11	Sonoma	558	0
...	...	...	...	...
652	2021-05-29	San Francisco	3878	233
653	2021-05-29	San Mateo	2346	140
654	2021-05-29	Santa Clara	5307	419
655	2021-05-29	Solano	1580	140
656	2021-05-29	Sonoma	1410	126

653 rows × 4 columns

Then we can drop the first 50 rows by using a slicing method on the index. Assign the result to bayArea\_unemployment\_drop50 with our same input, but instead of listing the indices, use a slice: (index = bayArea\_unemployment.index[:50])

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
50	2020-02-15	San Mateo	509	0
51	2020-02-15	Santa Clara	1395	0
52	2020-02-15	Solano	535	0
53	2020-02-15	Sonoma	411	0
54	2020-02-22	Alameda	1046	0
...	...	...	...	...
652	2021-05-29	San Francisco	3878	233
653	2021-05-29	San Mateo	2346	140
654	2021-05-29	Santa Clara	5307	419
655	2021-05-29	Solano	1580	140
656	2021-05-29	Sonoma	1410	126

607 rows × 4 columns

## Insert a column

This is a common need when doing analysis: creating a new variable (column) from existing data. There are multiple ways of doing this.

- One simple way of inserting a new column and populating it with data is to essentially assign something to a new variable (column) that we name in the `[]` accessor of the data frame. It inserts that new column at the end of the dataframe. In this example every row within our new column will have the same value (which we've provided hard-coded, but this method would also work if you were to derive that from another source/input), and we might be later merging rows with data from other regions.

```
region = "Bay Area"
bayArea_unemployment["Region Name"] = region
bayArea_unemployment
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims	Region Name
0	2020-01-11	Alameda	1487	0	Bay Area
1	2020-01-11	Contra Costa	1073	0	Bay Area
2	2020-01-11	Marin	144	0	Bay Area
3	2020-01-11	Napa	162	0	Bay Area
4	2020-01-11	San Francisco	945	0	Bay Area
...	...	...	...	...	...
652	2021-05-29	San Francisco	3878	233	Bay Area
653	2021-05-29	San Mateo	2346	140	Bay Area
654	2021-05-29	Santa Clara	5307	419	Bay Area
655	2021-05-29	Solano	1580	140	Bay Area
656	2021-05-29	Sonoma	1410	126	Bay Area

657 rows × 5 columns

- In addition we can use the `.insert()` method to insert a field into a dataframe at a specific location, in this case as the second column (at the 1 position):

```
state = "California"
bayArea_unemployment.insert(1, "State", state)
bayArea_unemployment
```

In [ ]: #

Out[ ]:

	Week_Ending	State	County	UI_Claims	PUA_Claims	Region Name
0	2020-01-11	California	Alameda	1487	0	Bay Area
1	2020-01-11	California	Contra Costa	1073	0	Bay Area
2	2020-01-11	California	Marin	144	0	Bay Area
3	2020-01-11	California	Napa	162	0	Bay Area
4	2020-01-11	California	San Francisco	945	0	Bay Area
...	...	...	...	...	...	...
652	2021-05-29	California	San Francisco	3878	233	Bay Area
653	2021-05-29	California	San Mateo	2346	140	Bay Area
654	2021-05-29	California	Santa Clara	5307	419	Bay Area
655	2021-05-29	California	Solano	1580	140	Bay Area
656	2021-05-29	California	Sonoma	1410	126	Bay Area

657 rows × 6 columns

▀ Insert another column called Country and set it to "United States". Put Country just after State .

In [ ]: #

Out[ ]:

	Week_Ending	Country	State	County	UI_Claims	PUA_Claims	Region Name
0	2020-01-11	United States	California	Alameda	1487	0	Bay Area
1	2020-01-11	United States	California	Contra Costa	1073	0	Bay Area
2	2020-01-11	United States	California	Marin	144	0	Bay Area
3	2020-01-11	United States	California	Napa	162	0	Bay Area
4	2020-01-11	United States	California	San Francisco	945	0	Bay Area
...	...	...	...	...	...	...	...
652	2021-05-29	United States	California	San Francisco	3878	233	Bay Area
653	2021-05-29	United States	California	San Mateo	2346	140	Bay Area
654	2021-05-29	United States	California	Santa Clara	5307	419	Bay Area
655	2021-05-29	United States	California	Solano	1580	140	Bay Area
656	2021-05-29	United States	California	Sonoma	1410	126	Bay Area

657 rows × 7 columns

## Adding rows

- Adding a new row by using the append method. The new row of data should be formatted with key:value pairs representing each column heading and row value, respectively:

```
new_row = {"Week_Ending": "2020-11-28", "Country":"United States", "State":"California",
           "County":"Marin", "UI_Claims": 450, "PUA_Claims":50, "Region Name":"Bay
Area"}
```

In [ ]: #

- Then check the usage of .append using either shift-tab or help. Note what it says about the usage and the first parameter other :

Append rows of other to the end of caller, returning a new object.

... and what it says about the second parameter ignore\_index which by default is False but we want to set to True.

- Note that we'll assign the output to the same object (while the documentation says it will be a new object), so it replaces it:

```
bay_area_unemployment = bayArea_unemployment.append(new_row, ignore_index=True)
bay_area_unemployment
```

In [ ]:

```
C:\Users\900008452\AppData\Local\Temp\ipykernel_10588\4119025374.py:2: Future
Warning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
```

```
bay_area_unemployment = bayArea_unemployment.append(new_row, ignore_index=True)
```

Out[ ]:

	Week_Ending	Country	State	County	UI_Claims	PUA_Claims	Region Name
0	2020-01-11 00:00:00	United States	California	Alameda	1487	0	Bay Area
1	2020-01-11 00:00:00	United States	California	Contra Costa	1073	0	Bay Area
2	2020-01-11 00:00:00	United States	California	Marin	144	0	Bay Area
3	2020-01-11 00:00:00	United States	California	Napa	162	0	Bay Area
4	2020-01-11 00:00:00	United States	California	San Francisco	945	0	Bay Area
...	...	...	...	...	...	...	...
653	2021-05-29 00:00:00	United States	California	San Mateo	2346	140	Bay Area
654	2021-05-29 00:00:00	United States	California	Santa Clara	5307	419	Bay Area
655	2021-05-29 00:00:00	United States	California	Solano	1580	140	Bay Area
656	2021-05-29 00:00:00	United States	California	Sonoma	1410	126	Bay Area
657	2020-11-28	United States	California	Marin	450	50	Bay Area

658 rows × 7 columns

Q What is the data type of the "new\_row" above?

Answer: dict

## Filtering

Filtering lets you select rows, based either on indices or values in fields.

■ Moving forward, recreate the dataframe `bayArea_unemployment` from the source CSV. Use the code we created earlier that reads the CSV and parses the dates.

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
0	2020-01-11	Alameda	1487	0
1	2020-01-11	Contra Costa	1073	0
2	2020-01-11	Marin	144	0
3	2020-01-11	Napa	162	0
4	2020-01-11	San Francisco	945	0
...	...	...	...	...
652	2021-05-29	San Francisco	3878	233
653	2021-05-29	San Mateo	2346	140
654	2021-05-29	Santa Clara	5307	419
655	2021-05-29	Solano	1580	140
656	2021-05-29	Sonoma	1410	126

657 rows × 4 columns

- Define a filter (mask) and assign it to the alameda variable, this is called creating a mask. We are creating the mask that we can apply to the dataframe that will conduct the desired filter:

```
alameda = bayArea_unemployment["County"]=="Alameda"
alameda
```

In [ ]: #

Out[ ]:

```
0      True
1     False
2     False
3     False
4     False
      ...
652    False
653    False
654    False
655    False
656    False
Name: County, Length: 657, dtype: bool
```

- Pass the filter to the larger dataframe to get only Alameda data returned:

```
alameda_unemployment = bayArea_unemployment[alameda]
alameda_unemployment
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
0	2020-01-11	Alameda	1487	0
9	2020-01-18	Alameda	1817	0
18	2020-01-25	Alameda	1350	0
27	2020-02-01	Alameda	1265	0
36	2020-02-08	Alameda	1328	0
...	...	...	...	...
612	2021-05-01	Alameda	5789	567
621	2021-05-08	Alameda	5544	580
630	2021-05-15	Alameda	5538	587
639	2021-05-22	Alameda	6892	507
648	2021-05-29	Alameda	5664	495

73 rows × 4 columns

Filtering using the .eq() method:

```
alameda_unemployment = bayArea_unemployment[bayArea_unemployment.County.eq("Alameda")]
alameda_unemployment
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
0	2020-01-11	Alameda	1487	0
9	2020-01-18	Alameda	1817	0
18	2020-01-25	Alameda	1350	0
27	2020-02-01	Alameda	1265	0
36	2020-02-08	Alameda	1328	0
...	...	...	...	...
612	2021-05-01	Alameda	5789	567
621	2021-05-08	Alameda	5544	580
630	2021-05-15	Alameda	5538	587
639	2021-05-22	Alameda	6892	507
648	2021-05-29	Alameda	5664	495

73 rows × 4 columns

Using either approach, create a `san_francisco_unemployment`

Be careful with spaces and underscores...

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
4	2020-01-11	San Francisco	945	0
13	2020-01-18	San Francisco	965	0
22	2020-01-25	San Francisco	739	0
31	2020-02-01	San Francisco	801	0
40	2020-02-08	San Francisco	824	0
...	...	...	...	...
616	2021-05-01	San Francisco	4040	336
625	2021-05-08	San Francisco	3791	367
634	2021-05-15	San Francisco	3909	397
643	2021-05-22	San Francisco	4619	281
652	2021-05-29	San Francisco	3878	233

73 rows × 4 columns

Filtering on multiple values using `.isin`

The `.isin` method is nice when you have a list of values for a given field you want.

Let's try this with county names. We'll create a list of two county names with

```
marin_sf = ["San Francisco", "Marin"]
```

... then see what you get with

```
bayArea_unemployment.County.isin(marin_sf)
```

In [ ]:

```
Out[ ]: 0    False
        1    False
        2     True
        3    False
        4     True
       ...
      652    True
      653   False
      654   False
      655   False
      656   False
Name: County, Length: 657, dtype: bool
```

So this is a *mask* of rows, which can be used to filter the rows by applying it to `bayArea_unemployment[ ... ]`. We'll then assign the output to `marin_sf_unemployment`.

It may seem a little weird to seem to access the same data twice with

`bayArea_unemployment[bayArea_unemployment.County.isin(marin_sf)]` but this is commonly done and it's just a way of nesting operations.

To maybe help visualize this, the following methods are equivalent, so use either one:

```
theMask = bayArea_unemployment.County.isin(marin_sf)
marin_sf_unemployment = bayArea_unemployment[theMask]

marin_sf_unemployment = bayArea_unemployment[bayArea_unemployment.County.isin(marin_sf)]
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
<b>2</b>	2020-01-11	Marin	144	0
<b>4</b>	2020-01-11	San Francisco	945	0
<b>11</b>	2020-01-18	Marin	202	0
<b>13</b>	2020-01-18	San Francisco	965	0
<b>20</b>	2020-01-25	Marin	146	0
...	...	...	...	...
<b>634</b>	2021-05-15	San Francisco	3909	397
<b>641</b>	2021-05-22	Marin	812	79
<b>643</b>	2021-05-22	San Francisco	4619	281
<b>650</b>	2021-05-29	Marin	635	70
<b>652</b>	2021-05-29	San Francisco	3878	233

146 rows × 4 columns

Filtering on a not ( != ) condition, meaning returning everything that does not meet a value or values:

```
#select everything except that does not equal two counties
not_marin_sf_unemployment = bayArea_unemployment[(bayArea_unemployment.County != "San Francisco") & (bayArea_unemployment.County != "Marin")]
not_marin_sf_unemployment
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
0	2020-01-11	Alameda	1487	0
1	2020-01-11	Contra Costa	1073	0
3	2020-01-11	Napa	162	0
5	2020-01-11	San Mateo	467	0
6	2020-01-11	Santa Clara	1443	0
...	...	...	...	...
651	2021-05-29	Napa	392	42
653	2021-05-29	San Mateo	2346	140
654	2021-05-29	Santa Clara	5307	419
655	2021-05-29	Solano	1580	140
656	2021-05-29	Sonoma	1410	126

511 rows × 4 columns

## Create a dataframe from filtered data

- Create a dataframe of the unemployment data that **does not** include Alameda County and Santa Clara.

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
1	2020-01-11	Contra Costa	1073	0
2	2020-01-11	Marin	144	0
3	2020-01-11	Napa	162	0
4	2020-01-11	San Francisco	945	0
5	2020-01-11	San Mateo	467	0
...	...	...	...	...
651	2021-05-29	Napa	392	42
652	2021-05-29	San Francisco	3878	233
653	2021-05-29	San Mateo	2346	140
655	2021-05-29	Solano	1580	140
656	2021-05-29	Sonoma	1410	126

511 rows × 4 columns

# Dataframe Iterating, Groupby, Summary Stats, Sort, and Assigning

## Groupby combined with summary statistic

One important use of the groupby method is to derive a summary statistic for a grouping. You might use it to derive the mean values by county or some other categorical field in the data frame. Let's create a simple example so we can see how it works.

Start by building the `sierra` data we built in the previous exercise. Use the version with dictionaries so you'll have named row indices for the stations, and end up with columns in the order elevation,temperature,latitude.

In [ ]: #

Out[ ]:

	elevation	temperature	latitude
Oroville	52	10.7	39.52
Auburn	394	9.7	38.91
Sonora	510	7.7	37.97
Placerville	564	9.2	38.70
Colfax	725	7.3	39.09
Nevada City	848	6.7	39.25
Quincy	1042	4.0	39.94
Yosemite	1225	5.0	37.75
Sierraville	1516	0.9	40.35
Truckee	1775	-1.1	39.33
Tahoe City	1899	-0.8	39.17
Bodie	2551	-4.4	38.21

We don't have a categorical variable in our data, so we'll create one "highElev" that has True for elevations > 1000 and False for not. (In R, we'd call this a factor.)

In [ ]: #

Out[ ]:

	elevation	temperature	latitude	highElev
Oroville	52	10.7	39.52	False
Auburn	394	9.7	38.91	False
Sonora	510	7.7	37.97	False
Placerville	564	9.2	38.70	False
Colfax	725	7.3	39.09	False
Nevada City	848	6.7	39.25	False
Quincy	1042	4.0	39.94	True
Yosemite	1225	5.0	37.75	True
Sierraville	1516	0.9	40.35	True
Truckee	1775	-1.1	39.33	True
Tahoe City	1899	-0.8	39.17	True
Bodie	2551	-4.4	38.21	True

Now we'll create a small dataframe where a summary statistic is derived for each variable by group. We'll just derive the mean values, but we could instead derive other statistics.

```
sierraElevGroup = sierra.groupby("highElev").mean()
sierraElevGroup
```

In [ ]: #

Out[ ]:

	elevation	temperature	latitude
highElev			
False	515.5	8.55	38.906667
True	1668.0	0.60	39.125000

## Continuing with the bayArea\_unemployment data

■ Sorting a dataframe on a field:

```
#sorting dataframe
sorted_unemployment = bayArea_unemployment.sort_values(by='UI_Claims', ascending=True)
sorted_unemployment
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims
57	2020-02-22	Napa	100	0
48	2020-02-15	Napa	115	0
75	2020-03-07	Napa	115	0
39	2020-02-08	Napa	130	0
66	2020-02-29	Napa	130	0
...	...	...	...	...
100	2020-03-28	Contra Costa	32426	0
114	2020-04-04	Santa Clara	38511	0
108	2020-04-04	Alameda	38645	0
99	2020-03-28	Alameda	46056	0
105	2020-03-28	Santa Clara	49472	0

657 rows × 4 columns

## Iterating a dataframe

▀ One other way to scroll through a dataframe by row, probably to do something with each step, is to *iterate* it with a `for` loop and the `.iterrows()` method, which you should learn about with either Shift-tab or help() to understand what the following is doing. Each iteration provides the row index and the row of data as a series with each of field names as indices. For example, to display the rows of sierra:

```
for index, row in sierra.iterrows():
    print(index)
    print(row)
```

In [ ]:

```
Oroville
elevation      52
temperature    10.7
latitude       39.52
highElev       False
Name: Oroville, dtype: object
Auburn
elevation      394
temperature    9.7
latitude       38.91
highElev       False
Name: Auburn, dtype: object
Sonora
elevation      510
temperature    7.7
latitude       37.97
highElev       False
Name: Sonora, dtype: object
Placerville
elevation      564
temperature    9.2
latitude       38.7
highElev       False
Name: Placerville, dtype: object
Colfax
elevation      725
temperature    7.3
latitude       39.09
highElev       False
Name: Colfax, dtype: object
Nevada City
elevation      848
temperature    6.7
latitude       39.25
highElev       False
Name: Nevada City, dtype: object
Quincy
elevation      1042
temperature    4.0
latitude       39.94
highElev       True
Name: Quincy, dtype: object
Yosemite
elevation      1225
temperature    5.0
latitude       37.75
highElev       True
Name: Yosemite, dtype: object
Sierraville
elevation      1516
temperature    0.9
latitude       40.35
highElev       True
Name: Sierraville, dtype: object
Truckee
elevation      1775
temperature    -1.1
```

```
latitude      39.33
highElev     True
Name: Truckee, dtype: object
Tahoe City
elevation    1899
temperature   -0.8
latitude     39.17
highElev     True
Name: Tahoe City, dtype: object
Bodie
elevation    2551
temperature   -4.4
latitude     38.21
highElev     True
Name: Bodie, dtype: object
```

### Using a condition in an `iterrows`:

Now we'll look at our bayArea\_unemployment data, which is much larger so we might want to only print (or do something else with) by setting a condition. Note in the code below how you're doing something `if` the condition is true. In this case, we're just printing something out, but you might imagine a situation where you *do something else* with each selected record, which is where iteration becomes most useful. We'll see examples of doing this in arcpy when we're using cursors to manipulate individual geometries.

```
for index, row in bayArea_unemployment.iterrows():
    if row["UI_Claims"] > 20000:
        print (row["County"], row["UI_Claims"])
```

In [ ]: #

```
Alameda 46056
Contra Costa 32426
San Francisco 26889
Santa Clara 49472
Alameda 38645
Contra Costa 26648
San Francisco 24040
Santa Clara 38511
Alameda 27574
Santa Clara 27906
Alameda 22842
Santa Clara 22920
Alameda 26991
Santa Clara 25577
Alameda 26274
Alameda 28748
```

## Using groupby to create stratified summary statistics

- Using the `groupby` function to create a new dataframe based on groups (and next we'll sum those values). Note that the groups are displayed here as a list of indices that belong to that group.

```
#groupby in a dataframe  
unemployment_by_county = bayArea_unemployment.groupby("County")  
unemployment_by_county.groups
```

In [ ]: #

```
Out[ ]: {'Alameda': [0, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99, 108, 117, 126, 135, 144, 153, 162, 171, 180, 189, 198, 207, 216, 225, 234, 243, 252, 261, 270, 279, 288, 297, 306, 315, 324, 333, 342, 351, 360, 369, 378, 387, 396, 405, 414, 423, 432, 441, 450, 459, 468, 477, 486, 495, 504, 513, 522, 531, 540, 549, 558, 567, 576, 585, 594, 603, 612, 621, 630, 639, 648], 'Contra Costa': [1, 10, 19, 28, 37, 46, 55, 64, 73, 82, 91, 100, 109, 118, 127, 136, 145, 154, 163, 172, 181, 190, 199, 208, 217, 226, 235, 244, 253, 262, 271, 280, 289, 298, 307, 316, 325, 334, 343, 352, 361, 370, 379, 388, 397, 406, 415, 424, 433, 442, 451, 460, 469, 478, 487, 496, 505, 514, 523, 532, 541, 550, 559, 568, 577, 586, 595, 604, 613, 622, 631, 640, 649], 'Marin': [2, 11, 20, 29, 38, 47, 56, 65, 74, 83, 92, 101, 110, 119, 128, 137, 146, 155, 164, 173, 182, 191, 200, 209, 218, 227, 236, 245, 254, 263, 272, 281, 290, 299, 308, 317, 326, 335, 344, 353, 362, 371, 380, 389, 398, 407, 416, 425, 434, 443, 452, 461, 470, 479, 488, 497, 506, 515, 524, 533, 542, 551, 560, 569, 578, 587, 596, 605, 614, 623, 632, 641, 650], 'Napa': [3, 12, 21, 30, 39, 48, 57, 66, 75, 84, 93, 102, 111, 120, 129, 138, 147, 156, 165, 174, 183, 192, 201, 210, 219, 228, 237, 246, 255, 264, 273, 282, 291, 300, 309, 318, 327, 336, 345, 354, 363, 372, 381, 390, 399, 408, 417, 426, 435, 444, 453, 462, 471, 480, 489, 498, 507, 516, 525, 534, 543, 552, 561, 570, 579, 588, 597, 606, 615, 624, 633, 642, 651], 'San Francisco': [4, 13, 22, 31, 40, 49, 58, 67, 76, 85, 94, 103, 112, 121, 130, 139, 148, 157, 166, 175, 184, 193, 202, 211, 220, 229, 238, 247, 256, 265, 274, 283, 292, 301, 310, 319, 328, 337, 346, 355, 364, 373, 382, 391, 400, 409, 418, 427, 436, 445, 454, 463, 472, 481, 490, 499, 508, 517, 526, 535, 544, 553, 562, 571, 580, 589, 598, 607, 616, 625, 634, 643, 652], 'San Mateo': [5, 14, 23, 32, 41, 50, 59, 68, 77, 86, 95, 104, 113, 122, 131, 140, 149, 158, 167, 176, 185, 194, 203, 212, 221, 230, 239, 248, 257, 266, 275, 284, 293, 302, 311, 320, 329, 338, 347, 356, 365, 374, 383, 392, 401, 410, 419, 428, 437, 446, 455, 464, 473, 482, 491, 500, 509, 518, 527, 536, 545, 554, 563, 572, 581, 590, 599, 608, 617, 626, 635, 644, 653], 'Santa Clara': [6, 15, 24, 33, 42, 51, 60, 69, 78, 87, 96, 105, 114, 123, 132, 141, 150, 159, 168, 177, 186, 195, 204, 213, 222, 231, 240, 249, 258, 267, 276, 285, 294, 303, 312, 321, 330, 339, 348, 357, 366, 375, 384, 393, 402, 411, 420, 429, 438, 447, 456, 465, 474, 483, 492, 501, 510, 519, 528, 537, 546, 555, 564, 573, 582, 591, 600, 609, 618, 627, 636, 645, 654], 'Solano': [7, 16, 25, 34, 43, 52, 61, 70, 79, 88, 97, 106, 115, 124, 133, 142, 151, 160, 169, 178, 187, 196, 205, 214, 223, 232, 241, 250, 259, 268, 277, 286, 295, 304, 313, 322, 331, 340, 349, 358, 367, 376, 385, 394, 403, 412, 421, 430, 439, 448, 457, 466, 475, 484, 493, 502, 511, 520, 529, 538, 547, 556, 565, 574, 583, 592, 601, 610, 619, 628, 637, 646, 655], 'Sonoma': [8, 17, 26, 35, 44, 53, 62, 71, 80, 89, 98, 107, 116, 125, 134, 143, 152, 161, 170, 179, 188, 197, 206, 215, 224, 233, 242, 251, 260, 269, 278, 287, 296, 305, 314, 323, 332, 341, 350, 359, 368, 377, 386, 395, 404, 413, 422, 431, 440, 449, 458, 467, 476, 485, 494, 503, 512, 521, 530, 539, 548, 557, 566, 575, 584, 593, 602, 611, 620, 629, 638, 647, 656]}
```

Q Using `type()`, what type of object did we create for the `.groupby` result and the `.groups` result? :

• Interpret the readout above, and consider whether it's any different from a regular dictionary. Remember that a list can be a value. Review our earlier discussion about the two uses of dictionaries in GIS data analysis. What does this one represent? :

```
In [ ]: #
```

```
Out[ ]: pandas.io.formats.printing.PrettyDict
```

### Derive a summary statistic from the group

Various summary statistics can be derived, such as mean, median, std, sum, max, min, by applying that to the `groupby` object just created. For instance, the mean is `unemployment_by_count.mean()`.

- Derive the sum of the claims, probably the most useful, since it'll be the total claims per county, and assign it to `unemployment_by_count`.

```
In [ ]: #
```

```
Out[ ]:
```

	UI_Claims	PUA_Claims
County		
<b>Alameda</b>	740263	170528
<b>Contra Costa</b>	503377	111380
<b>Marin</b>	87088	18838
<b>Napa</b>	64668	9219
<b>San Francisco</b>	416169	84599
<b>San Mateo</b>	282089	43713
<b>Santa Clara</b>	691576	116923
<b>Solano</b>	217828	48366
<b>Sonoma</b>	207900	36821

County	UI_Claims	PUA_Claims
<b>Alameda</b>	740263	170528
<b>Contra Costa</b>	503377	111380
<b>Marin</b>	87088	18838
<b>Napa</b>	64668	9219
<b>San Francisco</b>	416169	84599
<b>San Mateo</b>	282089	43713
<b>Santa Clara</b>	691576	116923
<b>Solano</b>	217828	48366
<b>Sonoma</b>	207900	36821

- We can also combine the operations to go directly from the unemployment data to the sum:

```
bayArea_unemployment.groupby("County").sum()
```

In [ ]: #

Out[ ]:

UI\_Claims PUA\_Claims

County		
<b>Alameda</b>	740263	170528
<b>Contra Costa</b>	503377	111380
<b>Marin</b>	87088	18838
<b>Napa</b>	64668	9219
<b>San Francisco</b>	416169	84599
<b>San Mateo</b>	282089	43713
<b>Santa Clara</b>	691576	116923
<b>Solano</b>	217828	48366
<b>Sonoma</b>	207900	36821

... then we can sort to see the county with the most unemployment claims in order:

```
sorted_unemployment_by_county = unemployment_by_county.sort_values(by='UI_Claims',
ascending=False)
sorted_unemployment_by_county
```

In [ ]: #

Out[ ]:

UI\_Claims PUA\_Claims

County		
<b>Alameda</b>	740263	170528
<b>Santa Clara</b>	691576	116923
<b>Contra Costa</b>	503377	111380
<b>San Francisco</b>	416169	84599
<b>San Mateo</b>	282089	43713
<b>Solano</b>	217828	48366
<b>Sonoma</b>	207900	36821
<b>Marin</b>	87088	18838
<b>Napa</b>	64668	9219

## Calculating fields:

### The .assign method

- Using the assign method, we will calculate the difference of UI Claims to PUA Claims for each row:

```
#Dataframe assign method
bayArea_unemployment.assign( PUA_UI_DIFF = bayArea_unemployment.UI_Claims - bayArea_unemployment.PUA_Claims)
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims	PUA_UI_DIFF
0	2020-01-11	Alameda	1487	0	1487
1	2020-01-11	Contra Costa	1073	0	1073
2	2020-01-11	Marin	144	0	144
3	2020-01-11	Napa	162	0	162
4	2020-01-11	San Francisco	945	0	945
...	...	...	...	...	...
652	2021-05-29	San Francisco	3878	233	3645
653	2021-05-29	San Mateo	2346	140	2206
654	2021-05-29	Santa Clara	5307	419	4888
655	2021-05-29	Solano	1580	140	1440
656	2021-05-29	Sonoma	1410	126	1284

657 rows × 5 columns

### Assigning a Boolean

- With the assign method, we can assign `True` or `False` by testing a condition, in this case whether there are any `PUA_claims`:

```
bayArea_unemployment.assign(has_pua_claims = bayArea_unemployment['PUA_Claims'] > 0)
```

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims	has_pua_claims
0	2020-01-11	Alameda	1487	0	False
1	2020-01-11	Contra Costa	1073	0	False
2	2020-01-11	Marin	144	0	False
3	2020-01-11	Napa	162	0	False
4	2020-01-11	San Francisco	945	0	False
...	...	...	...	...	...
652	2021-05-29	San Francisco	3878	233	True
653	2021-05-29	San Mateo	2346	140	True
654	2021-05-29	Santa Clara	5307	419	True
655	2021-05-29	Solano	1580	140	True
656	2021-05-29	Sonoma	1410	126	True

657 rows × 5 columns

- Write an assign statement that calculates a true value on a column called "high\_pua\_claims" where the the value is greater than 500.

In [ ]: #

Out[ ]:

	Week_Ending	County	UI_Claims	PUA_Claims	high_pua_claims
0	2020-01-11	Alameda	1487	0	False
1	2020-01-11	Contra Costa	1073	0	False
2	2020-01-11	Marin	144	0	False
3	2020-01-11	Napa	162	0	False
4	2020-01-11	San Francisco	945	0	False
...	...	...	...	...	...
652	2021-05-29	San Francisco	3878	233	False
653	2021-05-29	San Mateo	2346	140	False
654	2021-05-29	Santa Clara	5307	419	False
655	2021-05-29	Solano	1580	140	False
656	2021-05-29	Sonoma	1410	126	False

657 rows × 5 columns

## Concat, Plot, Join, and Merge Dataframes

In this section we'll take a look at conducting concats, merges, and joins on various tables. When you work with data, sometimes the data might exist in multiple tables and being able to conduct joins on your data is powerful. In this section we'll be looking at population data from two different time periods, that include multiple geographic regions. The tables that begin with "us\_pop" contain population data from all 50 states within the United States. And the tables that begin with "americas\_pop" contain population data from various Countries throughout North, Central, and South America.

- First, we'll create four separate dataframes from our source data. Make sure you've downloaded these csv files from iLearn and have placed them your py\Ex10 folder:

```
us_pop_00_09 = pd.read_csv("pdData/us_population_2000_to_09.csv", index_col="Name")
us_pop_10_19 = pd.read_csv("pdData/us_population_2010_to_19.csv", index_col="Name")
americas_pop_00_09 = pd.read_csv("pdData/america_population_2000_to_09.csv", index_
                                col="Name")
americas_pop_10_19 = pd.read_csv("pdData/america_population_2010_to_19.csv", index_
                                col="Name")
```

In [ ]: #

- Have a look at the dataframe of Americas population from 2000 to 2009.

In [ ]: #

Out[ ]:

	Region	2000	2001	2002	2003	2004	2005	2006
Name		2000	2001	2002	2003	2004	2005	2006
<b>Mexico</b>	North America	98899845	100298153	101684758	103081020	104514932	106005203	107560000
<b>Canada</b>	North America	30588383	30880073	31178263	31488048	31815494	32164309	325360000
<b>Colombia</b>	South America	39629968	40255967	40875360	41483869	42075955	42647723	432000000
<b>Venezuela</b>	South America	24192446	24646472	25100408	25551624	25996594	26432447	268500000
<b>Costa Rica</b>	Central America	3962372	4034074	4100925	4164053	4225155	4285502	434500000
<b>Guatemala</b>	Central America	11650743	11924946	12208848	12500478	12796925	13096028	133970000
<b>Brazil</b>	South America	174790340	177196054	179537520	181809246	184006481	186127103	188167000
<b>Argentina</b>	South America	36870787	37275652	37681749	38087868	38491972	38892931	392890000
<b>Chile</b>	South America	15342353	15516113	15684409	15849652	16014971	16182721	163540000
<b>Belize</b>	Central America	247315	255063	262378	269425	276504	283800	291000000
<b>El Salvador</b>	Central America	5887936	5927006	5962136	5994077	6023797	6052123	607900000
<b>Honduras</b>	Central America	6574509	6751912	6929265	7106319	7282953	7458985	763400000
<b>Nicaragua</b>	Central America	5069302	5145366	5219328	5292118	5364935	5438690	551300000
<b>Panama</b>	Central America	3030328	3089648	3149188	3209048	3269356	3330217	339100000
<b>Bolivia</b>	South America	8418264	8580235	8742814	8905823	9069039	9232306	939500000
<b>Ecuador</b>	South America	12681123	12914667	13143465	13369678	13596388	13825847	140590000
<b>Guyana</b>	South America	746715	745206	744789	745143	745737	746163	746900000
<b>Paraguay</b>	South America	5323201	5428444	5531962	5632983	5730549	5824096	591300000
<b>Peru</b>	South America	26459944	26799285	27100968	27372226	27624213	27866145	281020000
<b>Suriname</b>	South America	470949	476579	482235	487942	493679	499464	505300000
<b>Uruguay</b>	South America	3319736	3325473	3326040	3323668	3321476	3321803	332500000
<b>Venezuela</b>	South America	24192446	24646472	25100408	25551624	25996594	26432447	268500000

- To see the datatypes of each field in your dataframe:

```
americas_pop_00_09.dtypes
```

```
In [ ]: #
```

```
Out[ ]: Region    object
         2000     int64
         2001     int64
         2002     int64
         2003     int64
         2004     int64
         2005     int64
         2006     int64
         2007     int64
         2008     int64
         2009     int64
        dtype: object
```

Then we can examine all the fields within our dataframes. In the earlier Pandas lab, we looked at indices and columns as a `pandas.core.indexes.base.Index` object.

- Use `.columns` to get the column names of `americas_pop_00_09`. We're not needing this now, but as a refresher, use `.index` to see those.

```
In [ ]: #
```

```
Out[ ]: Index(['Region', '2000', '2001', '2002', '2003', '2004', '2005', '2006',
               '2007', '2008', '2009'],
              dtype='object')
```

```
In [ ]: #
```

```
Out[ ]: Index(['Mexico', 'Canada', 'Colombia', 'Venezuela', 'Costa Rica', 'Guatemala',
               'Brazil', 'Argentina', 'Chile', 'Belize', 'El Salvador', 'Honduras',
               'Nicaragua', 'Panama', 'Bolivia', 'Ecuador', 'Guyana', 'Paraguay',
               'Peru', 'Suriname', 'Uruguay', 'Venezuela'],
              dtype='object', name='Name')
```

- Let's make a dataframe that just contains the south american populations from 2000 - 2009

```
south_america_00_09 = americas_pop_00_09[americas_pop_00_09.Region.eq("South America")]
print(south_america_00_09.dtypes)
south_america_00_09
```

In [ ]:

```
#  
Region      object  
2000       int64  
2001       int64  
2002       int64  
2003       int64  
2004       int64  
2005       int64  
2006       int64  
2007       int64  
2008       int64  
2009       int64  
dtype: object
```

Out[ ]:

	Region	2000	2001	2002	2003	2004	2005	2006
	Name							
<b>Colombia</b>	South America	39629968	40255967	40875360	41483869	42075955	42647723	43200
<b>Venezuela</b>	South America	24192446	24646472	25100408	25551624	25996594	26432447	26850
<b>Brazil</b>	South America	174790340	177196054	179537520	181809246	184006481	186127103	188167
<b>Argentina</b>	South America	36870787	37275652	37681749	38087868	38491972	38892931	39289
<b>Chile</b>	South America	15342353	15516113	15684409	15849652	16014971	16182721	16354
<b>Bolivia</b>	South America	8418264	8580235	8742814	8905823	9069039	9232306	9395
<b>Ecuador</b>	South America	12681123	12914667	13143465	13369678	13596388	13825847	14059
<b>Guyana</b>	South America	746715	745206	744789	745143	745737	746163	746
<b>Paraguay</b>	South America	5323201	5428444	5531962	5632983	5730549	5824096	5913
<b>Peru</b>	South America	26459944	26799285	27100968	27372226	27624213	27866145	28102
<b>Suriname</b>	South America	470949	476579	482235	487942	493679	499464	505
<b>Uruguay</b>	South America	3319736	3325473	3326040	3323668	3321476	3321803	3325
<b>Venezuela</b>	South America	24192446	24646472	25100408	25551624	25996594	26432447	26850



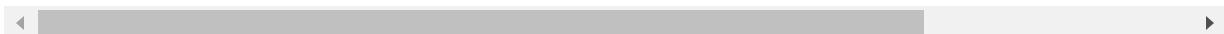
Next we're going to create a new dataframe for the purpose of plotting our data:

```
south_america_forplot = south_america_00_09.drop(columns=["Region"])
south_america_forplot
```

In [ ]: #

Out[ ]:

Name	2000	2001	2002	2003	2004	2005	2006
Colombia	39629968	40255967	40875360	41483869	42075955	42647723	43200897
Venezuela	24192446	24646472	25100408	25551624	25996594	26432447	26850194
Brazil	174790340	177196054	179537520	181809246	184006481	186127103	188167356
Argentina	36870787	37275652	37681749	38087868	38491972	38892931	39289878
Chile	15342353	15516113	15684409	15849652	16014971	16182721	16354504
Bolivia	8418264	8580235	8742814	8905823	9069039	9232306	9395446
Ecuador	12681123	12914667	13143465	13369678	13596388	13825847	14059384
Guyana	746715	745206	744789	745143	745737	746163	746343
Paraguay	5323201	5428444	5531962	5632983	5730549	5824096	5913209
Peru	26459944	26799285	27100968	27372226	27624213	27866145	28102056
Suriname	470949	476579	482235	487942	493679	499464	505295
Uruguay	3319736	3325473	3326040	3323668	3321476	3321803	3325401
Venezuela	24192446	24646472	25100408	25551624	25996594	26432447	26850194

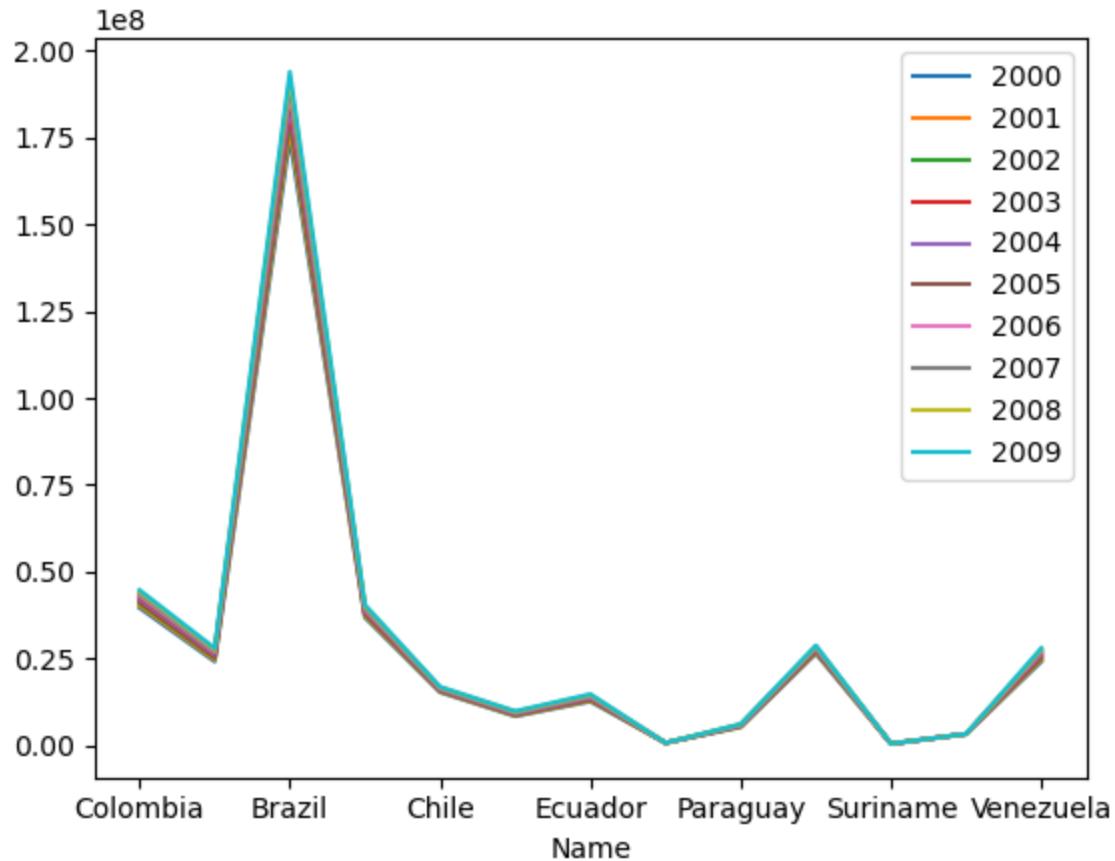


Now just call the .plot() method against the data.

```
south_america_forplot.plot()
```

```
In [ ]: #
```

```
Out[ ]: <AxesSubplot: xlabel='Name'>
```



## Transposing to improve the plot

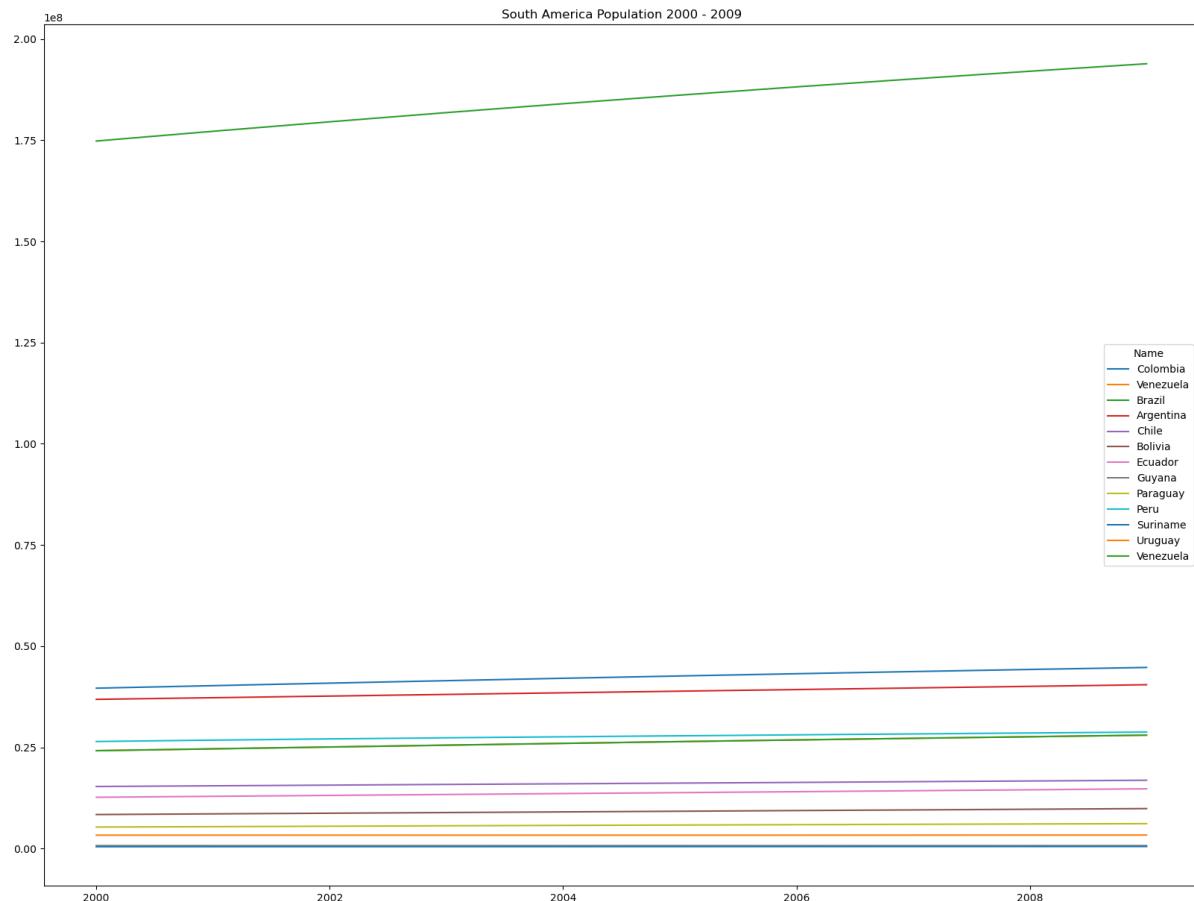
We notice this plot isn't very useful, as a line plot varying by country, and the differences from year to year are barely visible, so instead we'll transpose the data:

```
south_america_forplot.transpose().plot(title='South America Population 2000 - 2009', figsize=(20,15))
```

In [ ]:

```
#  
south_america_forplot.transpose().plot(title='South America Population 2000 -  
2009', figsize=(20,15))
```

Out[ ]: &lt;AxesSubplot: title={'center': 'South America Population 2000 - 2009'}&gt;

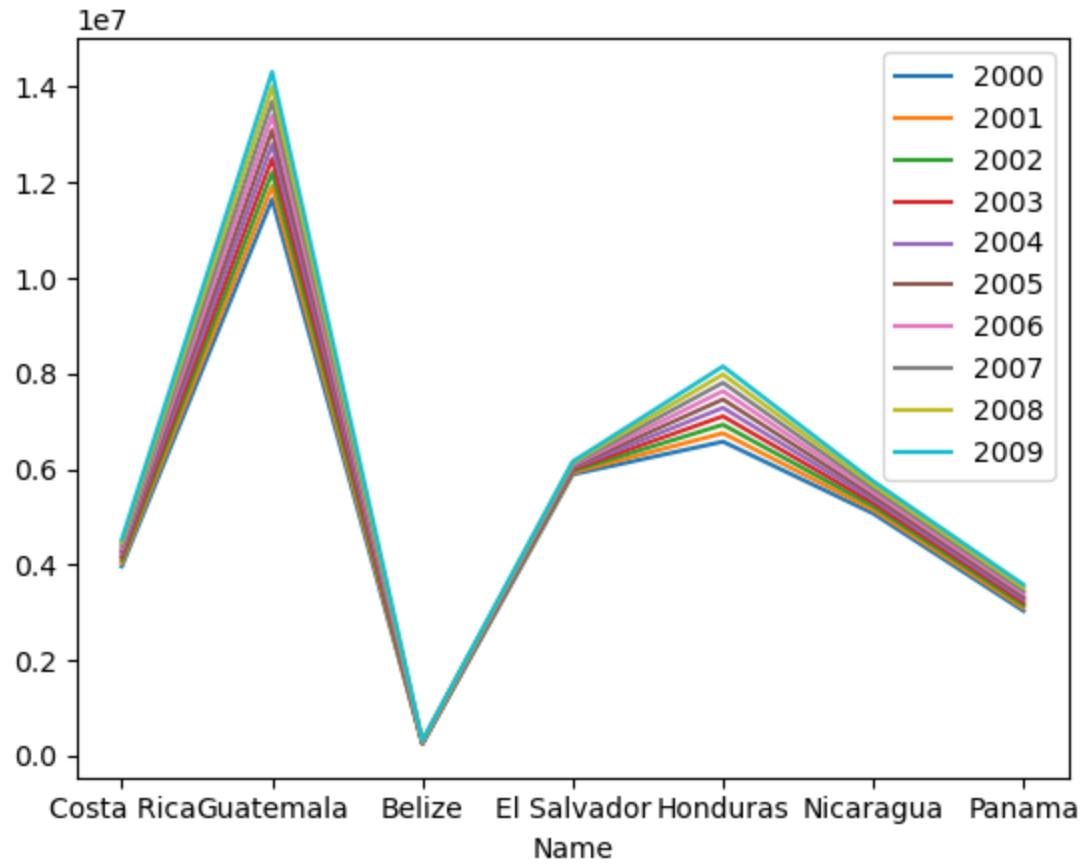


## Create plot

Now create the same plot that we did above, but this time create it with the data from Central America.

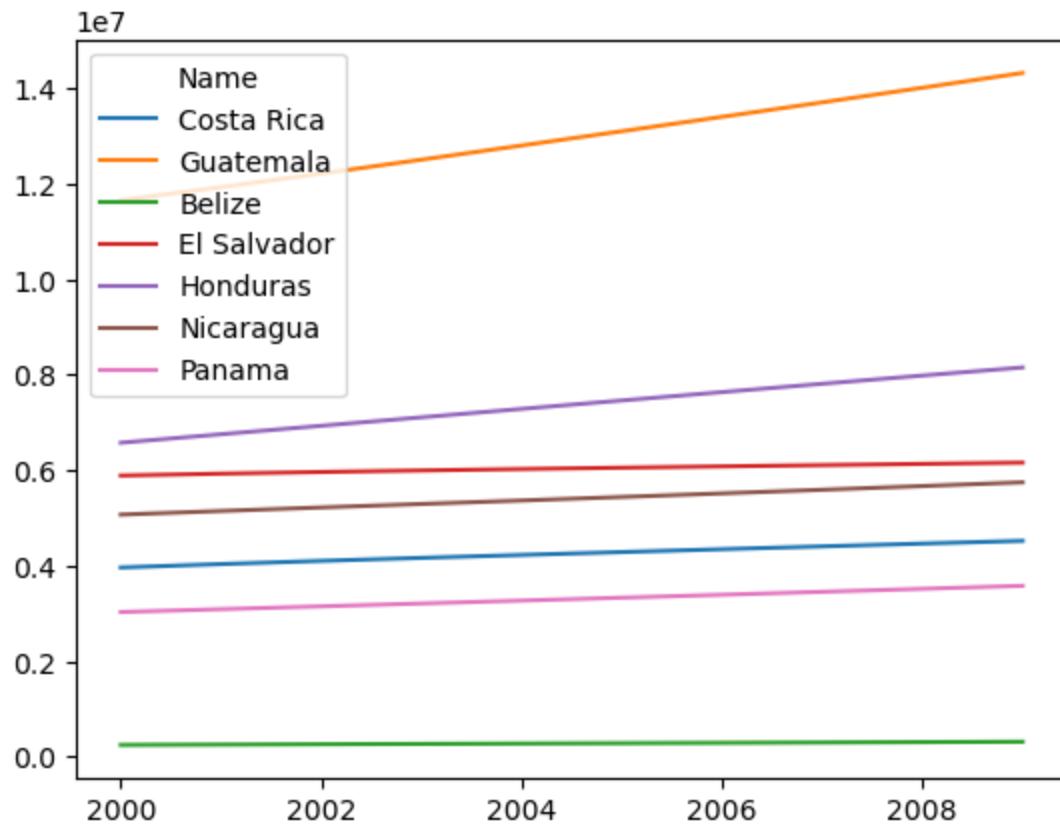
In [ ]: #

Out[ ]: &lt;AxesSubplot: xlabel='Name'&gt;



In [ ]: #

Out[ ]: &lt;AxesSubplot: &gt;



## Using concat and groupby to derive the total population by year for the North America 2000-2009

The `concat` method lets you combine rows from datasets that are structured similarly.

For our task, we'll use data from two different tables (Americas and US) for the first decade of the 21st century.

We'll use:

- `americas_pop_00_09`, which contains population for countries in North, South and Central America (excluding the United States)
- `us_pop_00_09`, which has United States population by state.

And for that decade, start with pulling out North America from the Americas data:

```
north_america_00_09 = americas_pop_00_09[americas_pop_00_09.Region.eq("North America")]
north_america_00_09
```

In [ ]: #

Out[ ]:

	Region	2000	2001	2002	2003	2004	2005	2006
	Name							
Mexico	North America	98899845	100298153	101684758	103081020	104514932	106005203	107560153
Canada	North America	30588383	30880073	31178263	31488048	31815494	32164309	32536987

Let's take a look at the US Population data now:

```
us_pop_00_09
```

In [ ]: #

Out[ ]:

	Region	2000	2001	2002	2003	2004	2005	2006
Name								
	Alabama	North America	4452173	4467634	4480089	4503491	4530729	4569805
	Alaska	North America	627963	633714	642337	648414	659286	666946
	Arizona	North America	5160586	5273477	5396255	5510364	5652404	5839077
	Arkansas	North America	2678588	2691571	2705927	2724816	2749686	2781097
	California	North America	33987977	34479458	34871843	35253159	35574576	35827943
	Colorado	North America	4326921	4425687	4490406	4528732	4575013	4631888
	Connecticut	North America	3411777	3432835	3458749	3484336	3496094	3506956
	Delaware	North America	786373	795699	806169	818003	830803	845150
	District of Columbia	North America	572046	574504	573158	568502	567754	567136
	Florida	North America	16047515	16356966	16689370	17004085	17415318	17842038
	Georgia	North America	8227303	8377038	8508256	8622793	8769252	8925922
	Hawaii	North America	1213519	1225948	1239613	1251154	1273569	1292729
	Idaho	North America	1299430	1319962	1340372	1363380	1391802	1428241
	Illinois	North America	12434161	12488445	12525556	12556006	12589773	12609903
	Indiana	North America	6091866	6127760	6155967	6196638	6233007	6278616
	Iowa	North America	2929067	2931997	2934234	2941999	2953635	2964454
	Kansas	North America	2693681	2702162	2713535	2723004	2734373	2745299
	Kentucky	North America	4049021	4068132	4089875	4117170	4146101	4182742
	Louisiana	North America	4471885	4477875	4497267	4521042	4552238	4576628
	Maine	North America	1277072	1285692	1295960	1306513	1313688	1318787
	Maryland	North America	5311034	5374691	5440389	5496269	5546935	5592379
	Massachusetts	North America	6361104	6397634	6417206	6422565	6412281	6403290

	Region	2000	2001	2002	2003	2004	2005	2006
Name								
<b>Michigan</b>	North America	9952450	9991120	10015710	10041152	10055315	10051137	10036081
<b>Minnesota</b>	North America	4933692	4982796	5018935	5053572	5087713	5119598	5163555
<b>Mississippi</b>	North America	2848353	2852994	2858681	2868312	2889010	2905943	2904978
<b>Missouri</b>	North America	5607285	5641142	5674825	5709403	5747741	5790300	5842704
<b>Montana</b>	North America	903773	906961	911667	919630	930009	940102	952692
<b>Nebraska</b>	North America	1713820	1719836	1728292	1738643	1749370	1761497	1772693
<b>Nevada</b>	North America	2018741	2098399	2173791	2248850	2346222	2432143	2522658
<b>New Hampshire</b>	North America	1239882	1255517	1269089	1279840	1290121	1298492	1308389
<b>New Jersey</b>	North America	8430621	8492671	8552643	8601402	8634561	8651974	8661679
<b>New Mexico</b>	North America	1821204	1831690	1855309	1877574	1903808	1932274	1962137
<b>New York</b>	North America	19001780	19082838	19137800	19175939	19171567	19132610	19104631
<b>North Carolina</b>	North America	8081614	8210122	8326201	8422501	8553152	8705407	8917270
<b>North Dakota</b>	North America	642023	639062	638168	638817	644705	646089	649422
<b>Ohio</b>	North America	11363543	11387404	11407889	11434788	11452251	11463320	11481213
<b>Oklahoma</b>	North America	3454365	3467100	3489080	3504892	3525233	3548597	3594090
<b>Oregon</b>	North America	3429708	3467937	3513424	3547376	3569463	3613202	3670883
<b>Pennsylvania</b>	North America	12284173	12298970	12331031	12374658	12410722	12449990	12510809
<b>Rhode Island</b>	North America	1050268	1057142	1065995	1071342	1074579	1067916	1063096
<b>South Carolina</b>	North America	4024223	4064995	4107795	4150297	4210921	4270150	4357847
<b>South Dakota</b>	North America	755844	757972	760020	763729	770396	775493	783033
<b>Tennessee</b>	North America	5703719	5750789	5795918	5847812	5910809	5991057	6088766
<b>Texas</b>	North America	20944499	21319622	21690325	22030931	22394023	22778123	23359580
<b>Utah</b>	North America	2244502	2283715	2324815	2360137	2401580	2457719	2525507

	Region	2000	2001	2002	2003	2004	2005	2006
Name								
<b>Vermont</b>	North America	609618	612223	615442	617858	619920	621215	622892
<b>Virginia</b>	North America	7105817	7198362	7286873	7366977	7475575	7577105	7673725
<b>Washington</b>	North America	5910512	5985722	6052349	6104115	6178645	6257305	6370753
<b>West Virginia</b>	North America	1807021	1801481	1805414	1812295	1816438	1820492	1827912
<b>Wisconsin</b>	North America	5373999	5406835	5445162	5479203	5514026	5546166	5577655
<b>Wyoming</b>	North America	494300	494657	500017	503453	509106	514157	522667

■ We can use the group by function to group all data in "us\_pop\_00\_09"

```
all_us_00_09 = us_pop_00_09.groupby("Region", as_index = False).sum()
all_us_00_09
```

In [ ]: #

Out[ ]:

	Region	2000	2001	2002	2003	2004	2005	2006
0	North America	282162411	284968955	287625193	290107933	292805298	295516599	298379912

■ Notice when we did the group by and sum, it set the index to the Region column and we lost "Name", the previous index. So in order to concat the two tables, let's set the index in our north\_america\_00\_09 table to "Region".

```
all_us_00_09.set_index("Region")
```

In [ ]: #

Out[ ]:

	2000	2001	2002	2003	2004	2005	2006	1
Region								
North America	282162411	284968955	287625193	290107933	292805298	295516599	298379912	301231

■ Set index on north america data

```
north_america_00_09.set_index("Region")
```

In [ ]: #

Out[ ]:

	2000	2001	2002	2003	2004	2005	2006	2007
<b>Region</b>								

Region	2000	2001	2002	2003	2004	2005	2006	2007
North America	98899845	100298153	101684758	103081020	104514932	106005203	107560153	109170100
North America	30588383	30880073	31178263	31488048	31815494	32164309	32536987	32930100



■ Now we can concat the two datasets to get Canada, Mexico and the US:

```
all_nAmerica = pd.concat([north_america_00_09, all_us_00_09])
all_nAmerica.set_index("Region")
```

In [ ]: #

Out[ ]:

	2000	2001	2002	2003	2004	2005	2006	2007
--	------	------	------	------	------	------	------	------

Region	2000	2001	2002	2003	2004	2005	2006	2007
North America	98899845	100298153	101684758	103081020	104514932	106005203	107560153	109170100
North America	30588383	30880073	31178263	31488048	31815494	32164309	32536987	32930100
North America	282162411	284968955	287625193	290107933	292805298	295516599	298379912	301230100



■ Then do a final sum to get all population in North America:

```
total_north_america_pop = all_nAmerica.groupby("Region", as_index = False).sum()
total_north_america_pop
```

In [ ]:

Out[ ]:

Region	2000	2001	2002	2003	2004	2005	2006	
North America	411650639	416147181	420488214	424677001	429135724	433686111	438477052	443301000



## Combine the decades with .merge

Next, we'll concat the two tables for each set of date ranges. So, we'll first concat the Americas population from 2000 - 2009 with the US population from 2000-2009:

```
all_00_09 = pd.concat([americas_pop_00_09, us_pop_00_09])
all_00_09
```

In [ ]: #

Out[ ]:

	Region	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009
	Name										
	<b>Mexico</b>	North America	98899845	100298153	101684758	103081020	104514932	106005203	1075600	1091600	1107600
	<b>Canada</b>	North America	30588383	30880073	31178263	31488048	31815494	32164309	3253600	3291600	3329600
	<b>Colombia</b>	South America	39629968	40255967	40875360	41483869	42075955	42647723	4320000	4376000	4432000
	<b>Venezuela</b>	South America	24192446	24646472	25100408	25551624	25996594	26432447	2685000	2726000	2767000
	<b>Costa Rica</b>	Central America	3962372	4034074	4100925	4164053	4225155	4285502	4345000	4405000	4465000
...	...	...	...	...	...	...	...	...	...	...	...
	<b>Virginia</b>	North America	7105817	7198362	7286873	7366977	7475575	7577105	7673000	7771000	7871000
	<b>Washington</b>	North America	5910512	5985722	6052349	6104115	6178645	6257305	6370000	6470000	6570000
	<b>West Virginia</b>	North America	1807021	1801481	1805414	1812295	1816438	1820492	1827000	1837000	1847000
	<b>Wisconsin</b>	North America	5373999	5406835	5445162	5479203	5514026	5546166	5577000	5677000	5777000
	<b>Wyoming</b>	North America	494300	494657	500017	503453	509106	514157	5220000	5320000	5420000

73 rows × 11 columns



Then we'll concat data tables for the same regions from 2010 to 2019:

```
all_10_19 = pd.concat([americas_pop_10_19, us_pop_10_19])
all_10_19
```

In [ ]: #

Out[ ]:

	Region	2010	2011	2012	2013	2014	2015	...
	Name							
<b>Mexico</b>	North America	114092963	115695473	117274155	118827161	120355128	121858258	12333
<b>Canada</b>	North America	34147564	34539159	34922030	35296528	35664337	36026676	3638
<b>Colombia</b>	South America	45222700	45662748	46075718	46495493	46967696	47520667	4817
<b>Venezuela</b>	South America	28439940	28887874	29360837	29781040	30042968	30081829	2985
<b>Costa Rica</b>	Central America	4577378	4633086	4688000	4742107	4795396	4847804	489
...	...	...	...	...	...	...	...	...
<b>Virginia</b>	North America	8023699	8101155	8185080	8252427	8310993	8361808	841
<b>Washington</b>	North America	6742830	6826627	6897058	6963985	7054655	7163657	729
<b>West Virginia</b>	North America	1854239	1856301	1856872	1853914	1849489	1842050	183
<b>Wisconsin</b>	North America	5690475	5705288	5719960	5736754	5751525	5760940	577
<b>Wyoming</b>	North America	564487	567299	576305	582122	582531	585613	58

73 rows × 11 columns



Now that we have these two new datasets, we can run a pd.merge() operation, which has the usage:

```
result = pd.merge(left DataFrame, right DataFrame, left_index=False, right_index=False, how='inner')
```

With the merge function, the first two parameters will always be the left and right dataframes. For our purpose, we want to set "left\_index=True" and "right\_index=True" to specify that the indices will be our key values and we can retrain the Countries as the index. Finally, we pass in "how='right'" to indicate a right join.

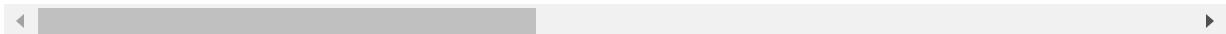
```
merged_df = pd.merge(all_00_09, all_10_19, left_index=True, right_index=True, how='right')
merged_df
```

In [ ]: #

Out[ ]:

	Name	Region_x	2000	2001	2002	2003	2004	2005	2006
	<b>Alabama</b>	North America	4452173	4467634	4480089	4503491	4530729	4569805	4628981
	<b>Alaska</b>	North America	627963	633714	642337	648414	659286	666946	675302
	<b>Argentina</b>	South America	36870787	37275652	37681749	38087868	38491972	38892931	39289878
	<b>Arizona</b>	North America	5160586	5273477	5396255	5510364	5652404	5839077	6029141
	<b>Arkansas</b>	North America	2678588	2691571	2705927	2724816	2749686	2781097	2821761
	...	...	...	...	...	...	...	...	...
	<b>Virginia</b>	North America	7105817	7198362	7286873	7366977	7475575	7577105	7673725
	<b>Washington</b>	North America	5910512	5985722	6052349	6104115	6178645	6257305	6370753
	<b>West Virginia</b>	North America	1807021	1801481	1805414	1812295	1816438	1820492	1827912
	<b>Wisconsin</b>	North America	5373999	5406835	5445162	5479203	5514026	5546166	5577655
	<b>Wyoming</b>	North America	494300	494657	500017	503453	509106	514157	522667

75 rows × 22 columns



## .join()

☰ Above was the merge method, which gives you a lot of ability to do different types of joins and to join on different fields if you want. Next, we'll look at the very straightforward and simple Join operation.

It is useful to use the ".join()" method *if you know you want to join on the index field* and your data is relatively clean and straightforward, with the usage:

```
result = DataFrame.join([other DataFrame], how='inner', on=None)
```

The DataFrame.join() method lets us use dot notation on our left table, then pass in the right table and how as an argument. This eliminates the need to specify the right and left index arguments like we did in the previous function. If on=None, the join key will be the row index. Let's observe how the nulls are affecting our analysis by taking a look at the DataFrame head.

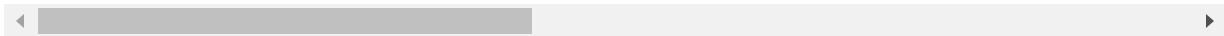
```
joined_df = all_00_09.join(all_10_19, how='right', lsuffix='_left', rsuffix='_right')
joined_df
```

In [ ]: #

Out[ ]:

	Name	Region_left	2000	2001	2002	2003	2004	2005	2006
	<b>Alabama</b>	North America	4452173	4467634	4480089	4503491	4530729	4569805	4628981
	<b>Alaska</b>	North America	627963	633714	642337	648414	659286	666946	675302
	<b>Argentina</b>	South America	36870787	37275652	37681749	38087868	38491972	38892931	39289878
	<b>Arizona</b>	North America	5160586	5273477	5396255	5510364	5652404	5839077	6029141
	<b>Arkansas</b>	North America	2678588	2691571	2705927	2724816	2749686	2781097	2821761
	...	...	...	...	...	...	...	...	...
	<b>Virginia</b>	North America	7105817	7198362	7286873	7366977	7475575	7577105	7673725
	<b>Washington</b>	North America	5910512	5985722	6052349	6104115	6178645	6257305	6370755
	<b>West Virginia</b>	North America	1807021	1801481	1805414	1812295	1816438	1820492	1827912
	<b>Wisconsin</b>	North America	5373999	5406835	5445162	5479203	5514026	5546166	5577655
	<b>Wyoming</b>	North America	494300	494657	500017	503453	509106	514157	522667

75 rows × 22 columns

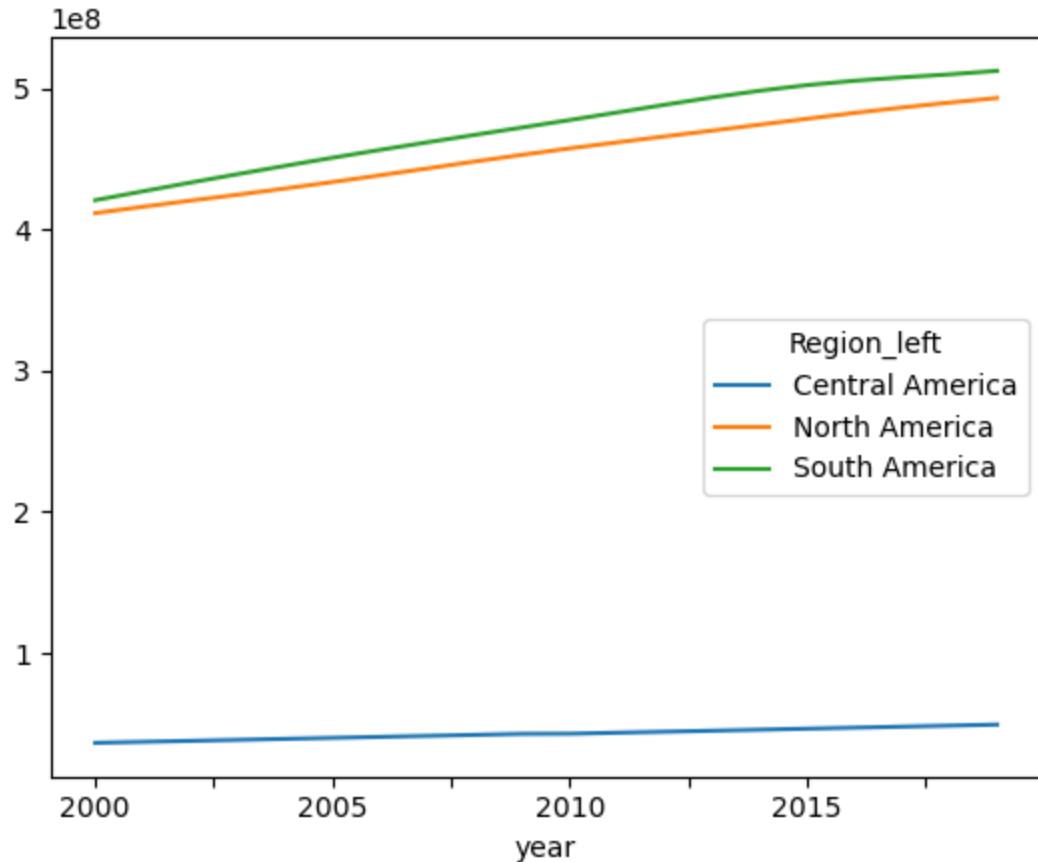


## Create and plot a joined dataframe

- >Create a dataframe that is a sum of joined\_df by Region and plot it using the matplotlib library.

```
In [ ]: #  
joined_plot = joined_df.groupby("Region_left").sum().transpose().rename_axis  
("year")  
joined_plot.plot()
```

```
Out[ ]: <AxesSubplot: xlabel='year'>
```



## Review of what we've learned

In this exercise, we've used several methods to transform data:

- Along the way we've also looked at
- Parsing dates
-

## key

- ➔ This directs you to do something specific, maybe in the operating system or answer something conceptual.
- ☒ Code to just run, typically boilerplate.
- 💻 Coding you need to write, in the subsequent code cell.
- ❓ Questions to answer in the same markdown cell.
- Similar to a question, but requesting an interpretation you need to provide, in the same markdown cell

In [ ]:

# Introduction to GeoPandas

Before you begin this exercise make sure you have all of the installation working; this should have happened the first week. Also, make sure you have downloaded the data from iLearn and placed the data in the "geodata" folder where the project folder is with the python files". To complete this exercise make sure you have this data in that geodata folder:

- owid-covid-data.csv
- World\_Map.shp
- BA\_Counties.shp
- sf\_neighborhoods.shp
- SF\_Nov20\_311.csv

In this exercise we will be working on the GeoPandas library. GeoPandas is an effective open source python library for analyzing large amounts of tabular and in particular, spatial data. GeoPandas adds a spatial geometry data type to Pandas and enables spatial operations on these types, using [shapely](#) (<https://github.com/Toblerity/Shapely>). GeoPandas leverages Pandas together with several core open source geospatial packages and practices to provide a uniquely simple and convenient framework for handling geospatial feature data, operating on both geometries and attributes jointly, and as with Pandas, largely eliminating the need to iterate over features (rows).

GeoPandas builds on mature, stable and widely used packages (Pandas, shapely, etc). It is being supported more and more as a preferred Python data structure for geospatial vector data, and a useful tool for exploratory *spatial* data analysis.

```
In [ ]: from IPython.core.interactiveshell import InteractiveShell  
InteractiveShell.ast_node_interactivity = "all"
```

█ First, import the pandas library as pd and matplotlib.pyplot as plt

```
import pandas as pd  
import matplotlib.pyplot as plt
```

```
In [ ]: #
```

## I. Mapping World COVID Data by Country

In this first section we will be using Global COVID data to make a map of cases and deaths by country. The data were downloaded from [Our World in Data] on 5 May 2022 (<https://ourworldindata.org/coronavirus-source-data> (<https://ourworldindata.org/coronavirus-source-data>)) as `owid-covid-data.csv`. You should have downloaded this file from iLearn and placed it in your working project directory for this exercise, in the `geodata` folder.

## Initial dataframe processing

>Create a dataframe called 'all\_data' by using `pd.read_csv` with that CSV (remember to include the folder "geodata" in the path):

In [ ]: #

Out[ ]:

	iso_code	continent	location	date	total_cases	new_cases	new_cases_smoothed	tc
0	AFG	Asia	Afghanistan	2020-02-24	5.0	5.0	5.0	NaN
1	AFG	Asia	Afghanistan	2020-02-25	5.0	0.0	0.0	NaN
2	AFG	Asia	Afghanistan	2020-02-26	5.0	0.0	0.0	NaN
3	AFG	Asia	Afghanistan	2020-02-27	5.0	0.0	0.0	NaN
4	AFG	Asia	Afghanistan	2020-02-28	5.0	0.0	0.0	NaN
...	...	...	...	...	...	...	...	...
166093	ZWE	Africa	Zimbabwe	2022-02-28	236380.0	577.0	401.286	
166094	ZWE	Africa	Zimbabwe	2022-03-01	236871.0	491.0	413.000	
166095	ZWE	Africa	Zimbabwe	2022-03-02	237503.0	632.0	416.286	
166096	ZWE	Africa	Zimbabwe	2022-03-03	237503.0	0.0	362.286	
166097	ZWE	Africa	Zimbabwe	2022-03-04	238739.0	1236.0	467.429	

166098 rows × 67 columns

What is the data type for the "all\_data" object above?

..

## Group by year with sums

We'll group by country next to start our analysis, but we it might be useful to also look at the covid data grouped by year as the total for each year and then displayed by country. We'll also want to **filter** the columns (axis=1) to not get sums of totals which are already accumulated values. We could have also used the **.drop()** method, but filtering columns is less code since there are fewer to retain than to drop. We'll also have a new dataframe with a new name. We'll also rename the columns since now the new cases and new deaths represent totals.

Note that both location and date are indices.

```
all_data['date'] = pd.to_datetime(all_data['date'])
all_data = all_data.filter(['location', "date", "new_cases", "new_deaths"], axis=1)
year_grpd = all_data.groupby(['location', all_data['date'].dt.year]).sum()
year_grpd.columns = ['TotalCases', 'TotalDeaths']
year_grpd
```

In [ ]: #

Out[ ]:

		TotalCases	TotalDeaths
location	date		
Afghanistan	2020	52332.0	2189.0
	2021	105754.0	5167.0
	2022	16136.0	263.0
Africa	2020	2760926.0	65507.0
	2021	6977423.0	163031.0
	...	...	...
Zambia	2021	233549.0	3346.0
	2022	59339.0	224.0
Zimbabwe	2020	13873.0	363.0
	2021	199391.0	4641.0
	2022	25481.0	393.0

693 rows × 2 columns

## Reduce the dataframe to totals for locations

- Use another `.groupby()` and `.sum()` to end up with total cases and deaths by location.

```
grouped_data = year_grpd.groupby('location').sum()  
grouped_data
```

In [ ]: #

Out[ ]:

location	TotalCases	TotalDeaths
Afghanistan	174222.0	7619.0
Africa	11247473.0	248869.0
Albania	272030.0	3478.0
Algeria	265186.0	6852.0
Andorra	38434.0	151.0
...	...	...
Wallis and Futuna	454.0	7.0
World	442613253.0	5964020.0
Yemen	11775.0	2135.0
Zambia	313613.0	3958.0
Zimbabwe	238745.0	5397.0

238 rows × 2 columns

- If we want to rename our dataframe's index then we can use the `.index.names` property to assign the desired value to our index column (note that this is different from `.set_index` which makes a field the index; we're just renaming the existing index). In this example below, we're going to use the word 'NAME' for the location field.

```
world_covid_cases.index.names = ['NAME']  
world_covid_cases
```

In [ ]: #

Out[ ]:

	TotalCases	TotalDeaths
NAME		
Afghanistan	174222.0	7619.0
Africa	11247473.0	248869.0
Albania	272030.0	3478.0
Algeria	265186.0	6852.0
Andorra	38434.0	151.0
...	...	...
Wallis and Futuna	454.0	7.0
World	442613253.0	5964020.0
Yemen	11775.0	2135.0
Zambia	313613.0	3958.0
Zimbabwe	238745.0	5397.0

238 rows × 2 columns

- █ Enter and execute the code that would show all the datatypes ( .dtype ) in the world\_covid\_cases object above.

In [ ]: #

Out[ ]: TotalCases float64  
TotalDeaths float64  
dtype: object

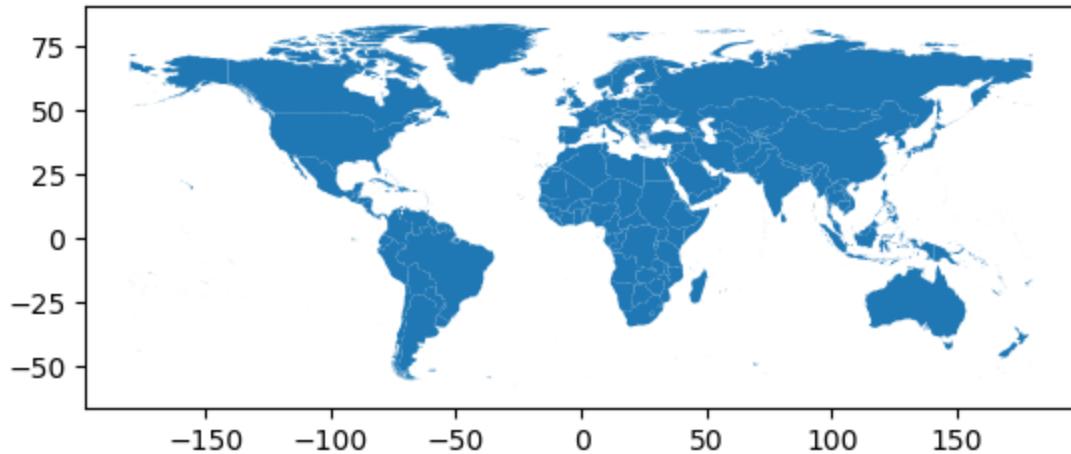
## Load spatial data for geopandas and merge with our COVID data

- █ Before we move forward in our data manipulation, let's load the shapefile of all countries in the world and then plot this shape file. We'll now start using geopandas so we'll need to import it.

```
import geopandas as gpd
world_data = gpd.read_file('geodata/World_Map.shp')
world_data.plot()
```

```
In [ ]: #
```

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



Then as a dataframe, just `world_data`

```
In [ ]: #
```

```
Out[ ]:
```

	NAME	geometry
0	Antigua and Barbuda	MULTIPOLYGON (((-61.68667 17.02444, -61.73806 ...
1	Algeria	POLYGON ((2.96361 36.80222, 2.98139 36.80694, ...
2	Azerbaijan	MULTIPOLYGON (((45.08332 39.76804, 45.26639 39...)
3	Albania	POLYGON ((19.43621 41.02107, 19.45055 41.06000...)
4	Armenia	MULTIPOLYGON (((45.57305 40.63249, 45.52888 40...)
...	...	...
240	Saint Barthelemy	POLYGON ((-63.02834 18.01555, -63.03334 18.015...
241	Guernsey	POLYGON ((-2.59083 49.42249, -2.59722 49.42249...)
242	Jersey	POLYGON ((-2.01500 49.21416, -2.02111 49.17722...)
243	South Georgia South Sandwich Islands	MULTIPOLYGON (((-27.32584 -59.42722, -27.29806...))
244	Taiwan	MULTIPOLYGON (((121.57639 22.00139, 121.57027 ...))

245 rows × 2 columns

Next, we'd like to compare the values that exist within our World shape country names with the names in our World Covid list. First, we'll loop over each Name within the "world\_covid\_cases" dataframe, by passing the index field to the `.tolist()` method. Then we'll generate a list from the World Shpae file and only print out the values that do not exist within both lists.

```
world_data_list = world_data['NAME'].tolist()
for item in world_covid_cases.index.tolist():
    if not(item in world_data_list):
        print(item + ' is not in the world data list')
```

In [ ]:

```
#  
Africa is not in the world data list  
Asia is not in the world data list  
Bonaire Sint Eustatius and Saba is not in the world data list  
Brunei is not in the world data list  
Curacao is not in the world data list  
Czechia is not in the world data list  
Democratic Republic of Congo is not in the world data list  
Eswatini is not in the world data list  
Europe is not in the world data list  
European Union is not in the world data list  
Faeroe Islands is not in the world data list  
Falkland Islands is not in the world data list  
High income is not in the world data list  
International is not in the world data list  
Iran is not in the world data list  
Kosovo is not in the world data list  
Laos is not in the world data list  
Libya is not in the world data list  
Low income is not in the world data list  
Lower middle income is not in the world data list  
Macao is not in the world data list  
Micronesia (country) is not in the world data list  
Moldova is not in the world data list  
Myanmar is not in the world data list  
North America is not in the world data list  
North Macedonia is not in the world data list  
Northern Cyprus is not in the world data list  
Oceania is not in the world data list  
Pitcairn is not in the world data list  
Sint Maarten (Dutch part) is not in the world data list  
South America is not in the world data list  
South Korea is not in the world data list  
South Sudan is not in the world data list  
Syria is not in the world data list  
Tanzania is not in the world data list  
Timor is not in the world data list  
Upper middle income is not in the world data list  
Vatican is not in the world data list  
Vietnam is not in the world data list  
Wallis and Futuna is not in the world data list  
World is not in the world data list
```

Just to see the types of names in our World shape file let's loop over and print each item in our "world\_data\_list".

```
for name in world_data_list:  
    print(name)
```

While some of the missing records are continental and other summaries, we have identified some of the major countries that we need to replace in our world\_data in order to make sure we can conduct a merge on our index field from the world\_covid\_cases Geodataframe.

```
world_data.replace('Korea, Republic of', 'S. Korea', inplace = True)  
world_data.replace('Iran (Islamic Republic of)', 'Iran', inplace = True)  
world_data.replace('Viet Nam', 'Vietnam', inplace = True)  
world_data.replace('Micronesia, Federated States of', 'Micronesia (country)', inplace = True)  
world_data.replace('Czech Republic', 'Czechia', inplace = True)  
world_data.replace('The former Yugoslav Republic of Macedonia', 'North Macedonia', inplace = True)  
world_data.replace('United Republic of Tanzania', 'Tanzania', inplace = True)  
world_data.replace('Democratic Republic of the Congo', 'Democratic Republic of Congo', inplace = True)  
world_data.replace('Libyan Arab Jamahiriya', 'Libya', inplace = True)
```

In [ ]: #

Now we can use the .merge() method to join our tabular World covid cases to our World shapefile.

```
combined = world_data.merge(world_covid_cases, on = 'NAME')  
combined
```

In [ ]: #

Out[ ]:

	NAME	geometry	TotalCases	TotalDeaths
0	Antigua and Barbuda	MULTIPOLYGON (((-61.68667 17.02444, -61.73806 ...	7451.0	135.0
1	Algeria	POLYGON ((2.96361 36.80222, 2.98139 36.80694, ...	265186.0	6852.0
2	Azerbaijan	MULTIPOLYGON (((45.08332 39.76804, 45.26639 39...)	788525.0	9488.0
3	Albania	POLYGON ((19.43621 41.02107, 19.45055 41.06000...)	272030.0	3478.0
4	Armenia	MULTIPOLYGON (((45.57305 40.63249, 45.52888 40...)	421008.0	8518.0
...	...	...	...	...
200	Turks and Caicos Islands	MULTIPOLYGON (((-71.14029 21.43194, -71.14584 ...	5868.0	36.0
201	Serbia	POLYGON ((20.07142 42.56091, 20.10583 42.64278...)	1921831.0	15378.0
202	Guernsey	POLYGON ((-2.59083 49.42249, -2.59722 49.42249...)	0.0	0.0
203	Jersey	POLYGON ((-2.01500 49.21416, -2.02111 49.17722...)	0.0	0.0
204	Taiwan	MULTIPOLYGON (((121.57639 22.00139, 121.57027 ...	20719.0	853.0

205 rows × 4 columns

Then we can plot our total Covid Cases World Wide. You might notice some parameters below that we have not discussed in class. There is a lot of variables and information about how to plot using the matplotlib library. Here is some documentation in the GeoPandas documentation [plotting with GeoPandas](https://geopandas.org/docs/user_guide/mapping.html) ([https://geopandas.org/docs/user\\_guide/mapping.html](https://geopandas.org/docs/user_guide/mapping.html))

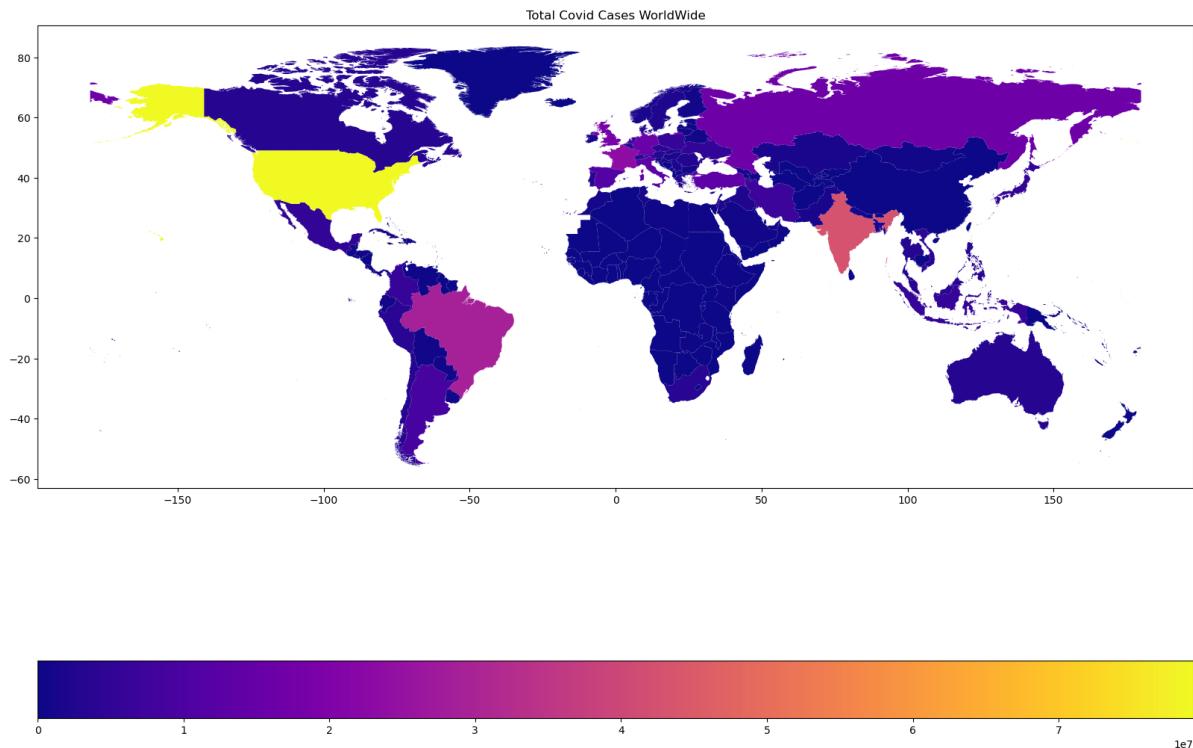
```
f, ax1 = plt.subplots(1, figsize=(20,20))
ax1.set_title("Total Covid Cases WorldWide")
combined.plot(ax=ax1, cmap='plasma', column="TotalCases", legend=True, legend_kwds=
{'orientation':'horizontal'})
plt.axis()
```

```
In [ ]: #
```

```
Out[ ]: Text(0.5, 1.0, 'Total Covid Cases WorldWide')
```

```
Out[ ]: <AxesSubplot: title={'center': 'Total Covid Cases WorldWide'}>
```

```
Out[ ]: (-197.99999999999997, 198.0, -62.896889999999914, 90.60076200000007)
```



❓ Can you describe what the "f" and "ax1" represent in the above block of code?

⋮

❓ While renaming some countries resulted in a pretty convincing map, yet there remain some issues resulting from geopolitical differences. For instance, what's happening with Western Sahara?

⋮

## II. Geopandas Geometric Operations

We'll explore geometric operations in Geopandas by looking at some **Bay Area counties** data. As we've seen above, GeoPandas can read shapefiles directly. Behind the scenes, this operation is using the `GDAL` package which contains the binaries capable of understanding geospatial data, the `fiona` package, which allows Python to interact nicely with `GDAL` libraries, and the `shapely` package which has functions for operating with feature classes in a Pythonic way. GeoPandas coordinate reference systems can use the "European Petroleum Survey Group" (EPSG) and other codes (e.g. ESRI) as shorthand for various standard systems. Complete documentation is available here [GeoPandas \(\[https://geopandas.org/docs/user\\\_guide/data\\\_structures.html\]\(https://geopandas.org/docs/user\_guide/data\_structures.html\)\)](https://geopandas.org/docs/user_guide/data_structures.html)

■ Read in the BA\_Counties.shp feature class, and then view it

```
import geopandas as gpd
import pandas as pd
ba_counties = gpd.read_file('geodata/BA_Counties.shp')
ba_counties
```

In [ ]: #

Out[ ]:

	OBJECTID	Acres	County	Shape_Leng	Shape_Area	geometry
0	1	4.763008e+05	Alameda	351719.536494	1.927521e+09	MULTIPOLYGON (((6033505.488 2112049.179, 60331...))
1	2	4.811194e+05	Contra Costa	292676.346086	1.947021e+09	MULTIPOLYGON (((6027405.300 2153185.401, 60272...))
2	3	3.360223e+05	Marin	419122.293142	1.359834e+09	MULTIPOLYGON (((5988028.191 2211673.341, 59879...))
3	4	5.054966e+05	Napa	278766.105739	2.045672e+09	POLYGON ((6021779.628 2506831.778, 6021843.051...))
4	5	3.021589e+04	San Francisco	106843.644006	1.222794e+08	MULTIPOLYGON (((6006624.799 2128525.324, 60061...))
5	6	2.902726e+05	San Mateo	306693.656488	1.174692e+09	MULTIPOLYGON (((6068478.537 2017484.502, 60691...))
6	7	8.312778e+05	Santa Clara	380752.936529	3.364062e+09	POLYGON ((6136976.940 1994399.503, 6137343.878...))
7	8	5.437972e+05	Solano	330514.170440	2.200669e+09	MULTIPOLYGON (((6121744.455 2214795.253, 61216...))
8	9	1.015939e+06	Sonoma	400983.408337	4.111360e+09	POLYGON ((5987868.159 2233054.492, 5987850.662...))

❓ ■ How many features in the dataset?

```
len(ba_counties)
```

We may have not used `len()` before with dataframes, but it's a base Python function that can return the length of strings and lists, or with numpy the total size, but with dataframes returns the number of records (rows) which for spatial data is the number of features. What do you get if you use `.size` with a dataframe?

```
In [ ]: #
```

```
Out[ ]: 9
```

- >List the unique geometry types in this geodataframe.

```
ba_counties.type.unique()
```

```
In [ ]: #
```

```
Out[ ]: array(['MultiPolygon', 'Polygon'], dtype=object)
```

- Why do you think there are these geometry types?

..

- Examine the attributes for the first feature

```
ba_counties.iloc[0]
```

```
In [ ]: #
```

```
Out[ ]: OBJECTID          1
        Acres            476300.784182
        County           Alameda
        Shape_Leng       351719.536494
        Shape_Area       1927520887.16
        geometry    (POLYGON ((6033505.487613098 2112049.179188581...
        Name: 0, dtype: object
```

## Projections and coordinate referencing systems in GeoPandas

Geospatial data area by definition always in a map projection and an associated coordinate referencing system (crs). We can detect the coordinate reference system of a spatial data frame with `.crs`.

- Use this with `world_data` and `ba_counties` and check <http://epsg.io> (<http://epsg.io>) to look up others (there are many thousands of these, since projections also have parameters that produce many variants, and these are needed for optimal display of geospatial data).

```
In [ ]: #
```

```
Out[ ]: <Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

```
Out[ ]: <Projected CRS: EPSG:2227>
Name: NAD83 / California zone 3 (ftUS)
Axis Info [cartesian]:
- E[east]: Easting (US survey foot)
- N[north]: Northing (US survey foot)
Area of Use:
- undefined
Coordinate Operation:
- name: unnamed
- method: Lambert Conic Conformal (2SP)
Datum: North_American_Datum_1983
- Ellipsoid: GRS 1980
- Prime Meridian: Greenwich
```

■ We can reproject our data in GeoPandas. Here we'll reproject our NAD83 data to UTM Zone 10 N, which has an EPSG code of 32610.

```
ba_counties_UTM = ba_counties.to_crs('EPSG:32610')
```

```
In [ ]: #
```

■ View the crs after we've assigned the projection

```
ba_counties_UTM.crs
```

```
In [ ]: #
```

```
Out[ ]: <Projected CRS: EPSG:32610>
Name: WGS 84 / UTM zone 10N
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Between 126°W and 120°W, northern hemisphere between equator and 84°N, onshore and offshore. Canada - British Columbia (BC); Northwest Territories (NWT); Nunavut; Yukon. United States (USA) - Alaska (AK).
- bounds: (-126.0, 0.0, -120.0, 84.0)
Coordinate Operation:
- name: UTM zone 10N
- method: Transverse Mercator
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

## The Geometry Field and Spatial Abstraction

The key to Geopandas is ability to work with geospatial data is by adding a new data type to the standard Pandas DataFrame: this is stored in the geometry field. Complete documentation on the geometry object is here: [http://geopandas.org/geometric\\_manipulations.html](http://geopandas.org/geometric_manipulations.html) ([http://geopandas.org/geometric\\_manipulations.html](http://geopandas.org/geometric_manipulations.html)).

Let's explore this field. The following code will show the first 5 values in the geometry field: this is actually a GeoSeries...

```
ba_counties_UTM['geometry'][0:5]
```

Note: if you open a shapefile in ArcGIS, you probably won't see a `geometry` field. Instead you'll see a `Shape` field, which represents the same thing, but it doesn't display its specific contents in the attribute table.

```
In [ ]: #
```

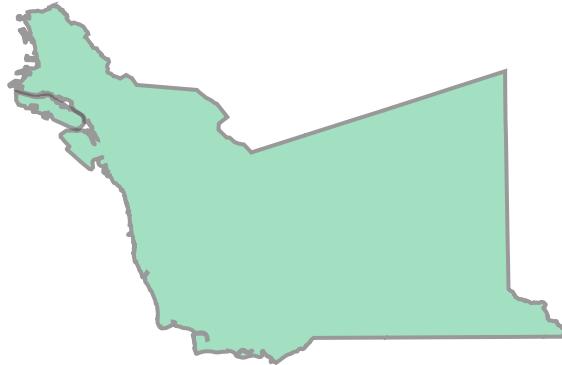
```
Out[ ]: 0    MULTIPOLYGON (((559203.322 4181745.002, 559093...
1    MULTIPOLYGON (((557010.199 4194225.573, 556971...
2    MULTIPOLYGON (((544539.414 4211720.068, 544529...
3    POLYGON ((552408.536 4301893.277, 552428.567 4...
4    MULTIPOLYGON (((550881.251 4186544.834, 550741...
Name: geometry, dtype: geometry
```

We can show just the first value, which will appear as a shape.

```
ba_counties_UTM['geometry'][0]
```

In [ ]: #

Out[ ]:

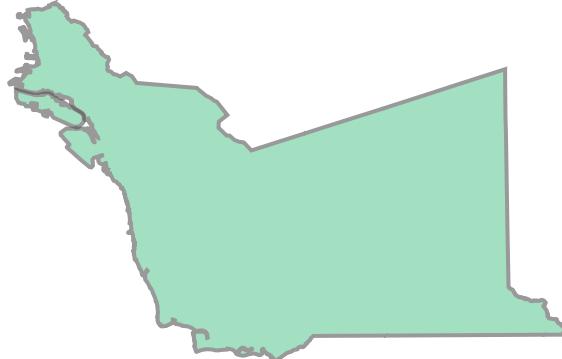


⌚ What does the following code produce and why?

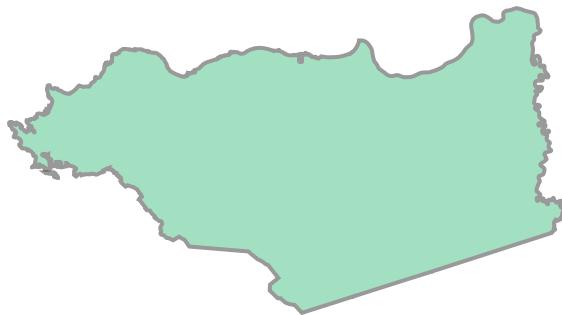
```
for i in range(len(ba_counties_UTM)):  
    ba_counties_UTM['geometry'][i]  
  
..
```

In [ ]: #

Out[ ]:



Out[ ]:



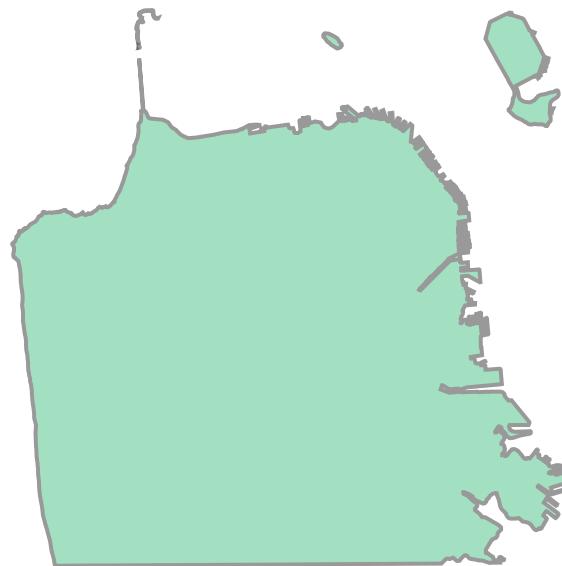
Out[ ]:



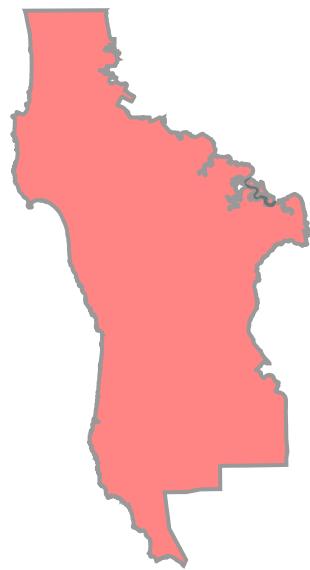
Out[ ]:



Out[ ]:



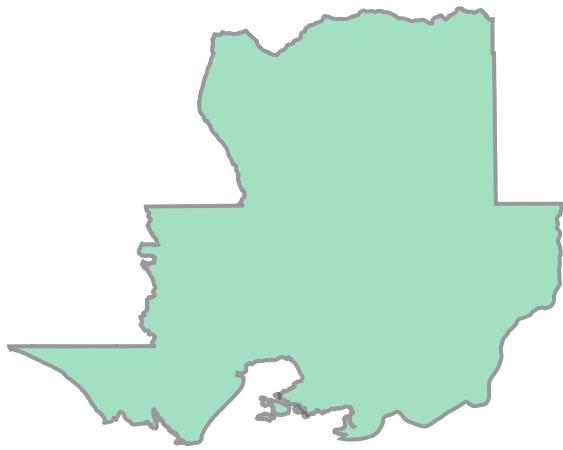
Out[ ]:



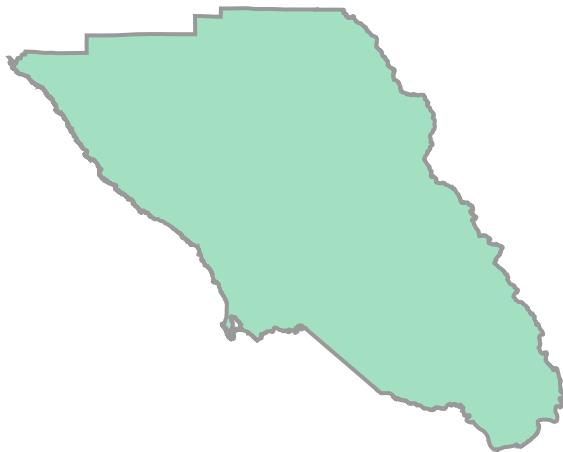
Out[ ]:



Out[ ]:



Out[ ]:



- Extract one feature geometry to a variable and show its type (not the same as dtype).

```
thePoly = ba_counties_UTM['geometry'][0]  
type(thePoly)
```

In [ ]: #

Out[ ]: shapely.geometry.multipolygon.MultiPolygon

- So what is `shapely`? Search online about `shapely` and `geopandas` to research your answer.

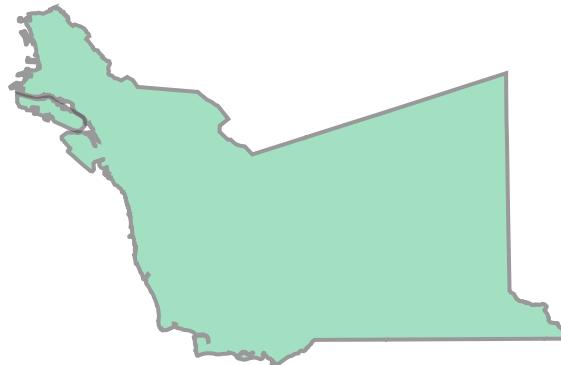
..

- Show `thePoly` and you should see the individual polygon (not surprising based on what we saw before).

```
thePoly
```

In [ ]: #

Out[ ]:



- Look at other geometric properties for `thePoly` using the properties `.area` and `.length`.

What do you think this length represents?

In [ ]: #

```
Area (m2): 1927520887  
Perimeter (m): 351719
```

We can also abstract parts of the geometry.

- Assign the result of the property `.boundary` to `theBoundary`, display it and list its geometry type .

In [ ]: #

Out[ ]:



Out[ ]: `shapely.geometry.multipolygon.MultiLineString`

## Spatial Analysis with GeoPandas

We've already started this with spatial abstractions like the boundary, but there are a lot more spatial analysis methods we can apply that you'll recognize from your experience with GIS.

### Centroid

- Create the centroid with the `.centroid` property, print it to see its value, and show its type,

In [ ]: #

POINT (598249.2377859637 4167169.885425969)

Out[ ]: `shapely.geometry.point.Point`

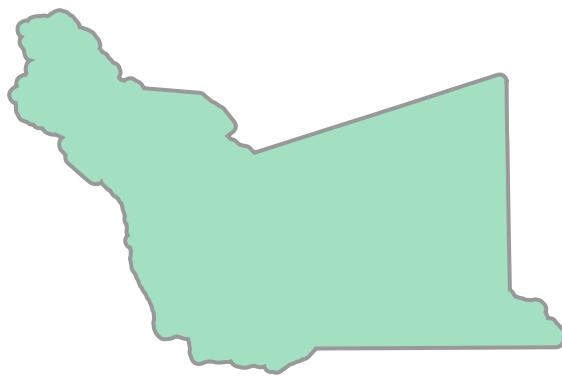
```
print(theCentroid)
```

## Buffer

- We can also do something you've certainly done before -- create a buffer polygon. Use the `.buffer()` method and provide 1000 (1 km) as the sole `distance` parameter.

```
In [ ]: #
```

Out[ ]:



## Distance Analysis

We can also compute distances fairly easily with GeoPandas objects.

- Here we'll compute the distance (in km) of each feature to the center point of all (`unary_union`) features.

```
#Compute the center of all features combined  
theCenter = ba_counties_UTM.unary_union.centroid  
theDistances = ba_counties_UTM.distance(theCenter)/1000
```

```
In [ ]: #
```

- Then get the mean distance using the `mean()` *method* -- note that it's a method not a property.

```
theDistances.mean()
```

```
In [ ]: #
```

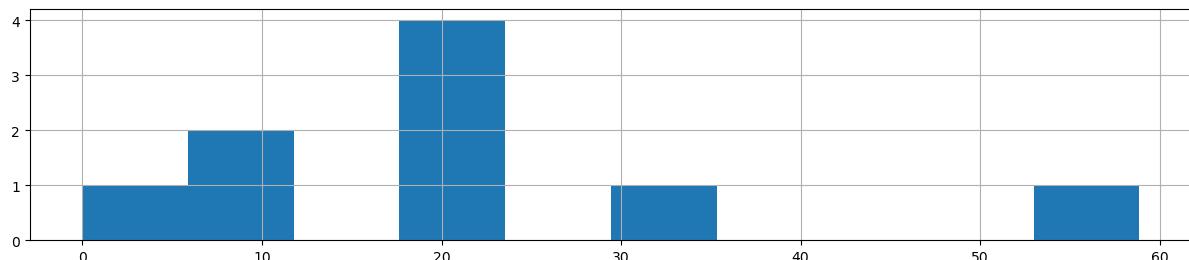
Out[ ]: 21.463153699118223

Then we can derive a histogram of these distances.

```
theDistances.hist(figsize=(15,3))
```

In [ ]: #

Out[ ]: <AxesSubplot: >



Here we will buffer the centroid of a feature and then intersect that with the feature.

We begin by selecting a feature. We'll pick Alameda County...

```
countyMask = ba_counties_UTM['County'] == 'Alameda'
Alameda = ba_counties_UTM[countyMask]
Alameda
```

You might have expected to see a map, but what we've just created is a record of the spatial dataframe, so it shows in as a row. Earlier we just pulled out the geometry.

In [ ]: #

Out[ ]:

	OBJECTID	Acres	County	Shape_Leng	Shape_Area	geometry
0	1	476300.784182	Alameda	351719.536494	1.927521e+09	MULTIPOLYGON ((559203.322 4181745.002, 559093...))

What do you get if you pull out the `geometry` field?

```
feature_geometry = Alameda['geometry']
type(feature_geometry)
features_geometry = ba_counties_UTM['geometry']
type(features_geometry)
```

Interpretation:

```
In [ ]: #
```

```
Out[ ]: geopandas.geoseries.GeoSeries
```

```
Out[ ]: geopandas.geoseries.GeoSeries
```

## Replacing geometries

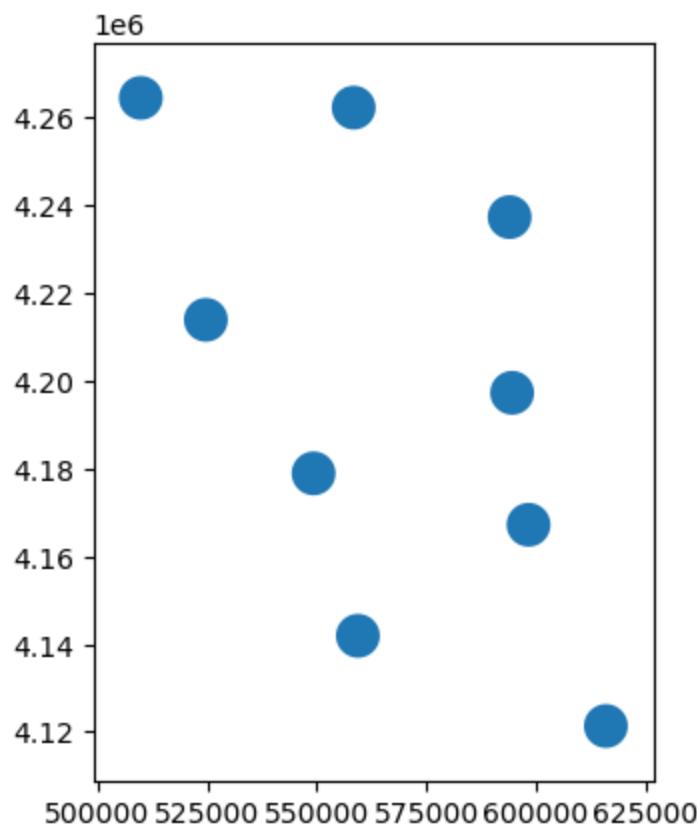
We can modify the geometry of our data in various ways.

Here's a simple example that will replace the shape with a 5000-m buffered centroid.

```
ba_counties_copy = ba_counties_UTM.copy()  
ba_counties_copy['geometry'] = ba_counties_copy['geometry'].centroid.buffer(5000)  
ba_counties_copy.plot()
```

```
In [ ]: #
```

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



### III. Making Maps with GeoPandas and Tabular Data

In this section we'll be taking a look at San Francisco's 311 data from November 2020. The file `SF_Nov20_311.csv` should be in your `geodata` folder. In particular we're going to make a simple map that contains all the Parking complaints reported through 311 in November 2020 and then using the `contextily` library, we'll be adding a basemap to our map.

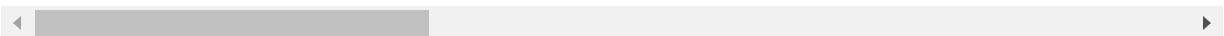
- Use `pd.read_csv()` to read this file and assign it to a new dataframe `sf_311`.

In [ ]: #

Out[ ]:

	CaseID	Opened	Closed	Updated	Status	Status_Notes	Responsible_Agent
0	13138553	11/13/2020 7:21	NaN	11/13/2020 14:09	Open	NaN	Port Author
1	13119442	11/8/2020 9:40	11/8/2020 9:45	11/8/2020 9:45	Closed	Case Resolved - Police Officer responded to re...	Parking Enforc Dispatch C
2	13091973	11/1/2020 0:08	11/1/2020 1:29	11/1/2020 1:29	Closed	Case Resolved - POLICE MATTER. PLEASE NOTIFY ...	311 Supervisor C
3	13155970	11/18/2020 6:27	11/18/2020 7:27	11/18/2020 7:27	Closed	Case Resolved - Officer responded to request u...	Parking Enforc Dispatch C
4	13180413	11/24/2020 14:19	11/24/2020 17:06	11/24/2020 17:06	Closed	Case Resolved - Officer responded to request u...	Parking Enforc Dispatch C
...	...	...	...	...	...	...	...
47991	13096891	11/2/2020 11:07	11/4/2020 10:08	11/4/2020 10:08	Closed	Comment Noted - To view the status, see\n13097...	
47992	13096288	11/2/2020 9:58	11/3/2020 13:28	11/3/2020 13:28	Closed	Case Transferred - DPH Environmental Health\n1...	
47993	13094573	11/1/2020 19:52	11/2/2020 8:00	11/2/2020 8:00	Closed	Case is a Duplicate - \n13094578,11/01/2020 0...	
47994	13093656	11/1/2020 14:13	11/17/2020 9:20	11/17/2020 9:20	Closed	Cancelled - \n13041136,10/19/2020 11:25:00 AM...	
47995	13092881	11/1/2020 10:22	11/6/2020 7:37	11/6/2020 7:37	Closed	Comment Noted - To view the status, see\n11470...	

47996 rows × 19 columns



Display all the field names within our dataframe using the `.columns` property.

In [ ]: #

Out[ ]: Index(['CaseID', 'Opened', 'Closed', 'Updated', 'Status', 'Status\_Notes', 'Responsible\_Agency', 'Category', 'Request\_Type', 'Request\_Details', 'Address', 'Street', 'Supervisor\_District', 'Neighborhood', 'Police District', 'Latitude', 'Longitude', 'Point', 'Source'], dtype='object')

- Next, because we might not be too familiar with our input data, we can do a .groupby() operation on "Request\_Type" and sort the results to see the 10 most common request types in our 311 data.

```
sf_311.groupby('Request_Type').Request_Type.count().sort_values(ascending=False).head(10)
```

- Interpret the various methods we've used in this statement and how they provide the result we're seeking.

..

In [ ]: #

Request_Type	
Bulky Items	10085
General Cleaning	7666
request_for_service	2180
Encampment Reports	1886
Human or Animal Waste	1855
Other_Illegal_Parking	1725
Graffiti on Other_enter_additional_details_below	1536
City_garbage_can_overflowing	1479
Illegal Postings - Affixed_Improperly	1399
Parking_on_Sidewalk	1188
Name: Request_Type, dtype: int64	

- Detect missing coordinates if any of the records are missing either Latitude or Longitude .

```
missing_coordinates = sf_311.Latitude.isnull() | sf_311.Longitude.isnull()
```

In [ ]: #

- How many missing coordinates?

```
missing_coordinates.sum()
```

In [ ]: #

Out[ ]: 46

- Here we can reassign the sf\_311 dataframe with just those that have coordinates by selecting with `.notna()` for either Latitude, Longitude, or Point (which holds a tuple of latitude and longitude, probably redundant for the other two fields.)

```
sf_311 = sf_311[sf_311['Latitude'].notna() & sf_311['Longitude'].notna()]
```

In [ ]: #

- Use the `missing_coordinates` method from above to confirm that there are no missing coordinates.

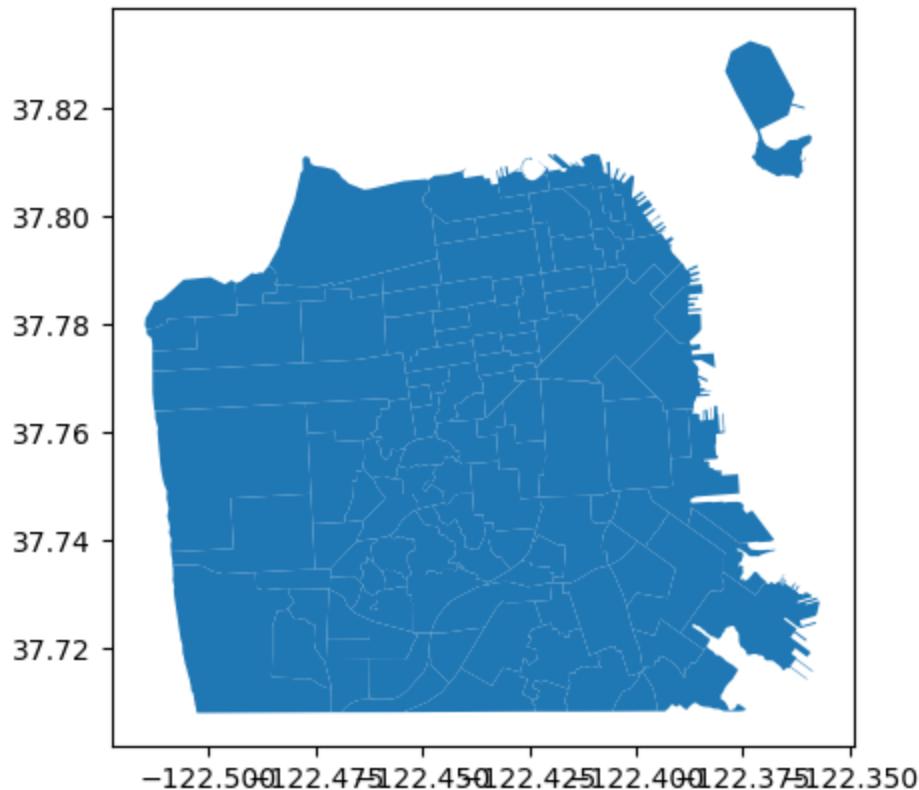
In [ ]: #

Out[ ]: 0

- Next, use `gpd.readfile()` and `.plot()` to assign a new GeoDataFrame `sf_neighborhoods` from `geodata/sf_neighborhoods.shp` and plot it.

In [ ]: #

Out[ ]: <AxesSubplot: >



Examine the `sf_neighborhoods` GeoDataFrame as a table.

`sf_neighborhoods`

In [ ]: #

Out[ ]:

			link	name	geometry
0	http://en.wikipedia.org/wiki/Sea_Cliff,_San_Fr...			Seacliff	POLYGON ((-122.49346 37.78352, -122.49373 37.7...)
1		None		Lake Street	POLYGON ((-122.48715 37.78379, -122.48729 37.7...)
2	http://www.nps.gov/prsf/index.htm			Presidio National Park	POLYGON ((-122.47758 37.81099, -122.47712 37.8...)
3		None	Presidio Terrace		POLYGON ((-122.47241 37.78735, -122.47100 37.7...)
4	http://www.sfgate.com/neighborhoods/sf/innerri...		Inner Richmond		POLYGON ((-122.47263 37.78631, -122.46683 37.7...)
...	...	...	...	...	...
112	http://en.wikipedia.org/wiki/Corona_Heights,_S...			Corona Heights	POLYGON ((-122.43519 37.76267, -122.43532 37.7...)
113	http://en.wikipedia.org/wiki/Haight-Ashbury	Ashbury Heights			POLYGON ((-122.45196 37.76148, -122.45210 37.7...)
114	http://en.wikipedia.org/wiki/Eureka_Valley,_Sa...			Eureka Valley	POLYGON ((-122.43734 37.76235, -122.43704 37.7...)
115	http://en.wikipedia.org/wiki/St._Francis_Wood,...			St. Francis Wood	POLYGON ((-122.47157 37.73471, -122.46831 37.7...)
116	http://en.wikipedia.org/wiki/Neighborhoods_in_...			Sherwood Forest	POLYGON ((-122.45890 37.74054, -122.45877 37.7...)

117 rows × 3 columns

What's the crs of `sf_neighborhoods` ?

..

In [ ]: #

Out[ ]: <Geographic 2D CRS: EPSG:4326>  
 Name: WGS 84  
 Axis Info [ellipsoidal]:  
 - Lat[north]: Geodetic latitude (degree)  
 - Lon[east]: Geodetic longitude (degree)  
 Area of Use:  
 - name: World.  
 - bounds: (-180.0, -90.0, 180.0, 90.0)  
 Datum: World Geodetic System 1984 ensemble  
 - Ellipsoid: WGS 84  
 - Prime Meridian: Greenwich

## Get the extent

- Use the `.total_bounds` property to assign the extent to `sf_extent`.

In [ ]: #

Out[ ]: array([-122.51489723, 37.70808921, -122.35698199, 37.83239598])

- What is the data type of the `sf_extent`?

..

- Realizing that the four extent values are ordered this way -- `[xmin,y_min,x_max,y_max]` -- assign those four values as variables `x_min`, etc.

In [ ]: #

- In the code above we created 4 variables that contains the x/y max and min. Next, we can set bounding extent that is too far outside our `sf_neighborhoods` data. We'll do this to exclude any 311 calls that are beyond our `sf_neighborhoods` extents.

```
too_far_ns = (sf_311.Latitude < y_min) | (sf_311.Latitude > y_max)
too_far_we = (sf_311.Longitude < x_min) | (sf_311.Longitude > x_max)
outside = too_far_ns | too_far_we
```

In [ ]: #

- Now, we can see how many records are beyond our extent.

```
outside.sum()
```

```
In [ ]: #
```

```
Out[ ]: 8
```

Let's filter our data, below is a new operation that you might not have seen in class yet. The `~` in the code below is a boolean operator in Pandas that mean "not". So, in this code we are saying return the values from the `sf_311` dataframe that are not outside our extent.

```
sf_311 = sf_311[~outside]
```

```
In [ ]: #
```

Once we've filtered out our data, we can now convert our Pandas dataframe into a GeoPandas Geodataframe. Here notice how we pass the `.points_from_xy()` method into the geometry option for loading into a Geopandas Geodataframe.

```
sf_311 = gpd.GeoDataFrame(sf_311, geometry=gpd.points_from_xy(sf_311.Longitude,  
sf_311.Latitude))  
sf_311
```

```
In [ ]: #
```

Before we plot our points, we'll want to add a basemap, and for that we need to use the `contextily` app. This needs some introduction...

## Basemaps from contextily

The `contextily` app provides basemaps -- *context*. This is something we take for granted when using ArcGIS Pro, but is a really useful capability, and the sheer number of basemaps available in ArcGIS is a big cartographic advantage. Part of what you're paying for with ArcGIS is access to all these basemap tile services. But there's an increasing number of open tile services, such as the well-known `OpenStreetMap`, which by its very name tells you it's open data.

However, one common source of error with basemaps is when a service provider goes off-line, or starts charging for their tiles, and I've had this happen with `contextily` when its default basemap provider Stamen started charging. So we'll explicitly tell it to access `OpenStreetMap`.

Before we plot our points we'll want to import the `contextily` python library as "cx", so we can add a basemap to our plot.

```
import contextily as cx
```

```
In [ ]: #
```

- Since the default projection is 4326 (GCS WGS84), we don't need to use `.set_crs` to set that, but we'd like to put it in 3857 ("Web Mercator") using `.to_crs()` to make sure the data will draw on top of our base map tiles from contextily.

```
sf_311 = sf_311.to_crs("EPSG:3857")
```

Note that we reassigned `sf_311` to be in this new crs. If we needed to work with it further in GCS, we'd need to run all of our code again to build it. We might have instead assigned it to something like `sf_311_webm` to separate it, but there's no particular advantage.

In [ ]: #

- To confirm this worked, get the crs property of `sf_311` with:

```
sf_311.crs
```

In [ ]: #

Out[ ]: <Projected CRS: EPSG:3857>  
Name: WGS 84 / Pseudo-Mercator  
Axis Info [cartesian]:  
- X[east]: Easting (metre)  
- Y[north]: Northing (metre)  
Area of Use:  
- name: World between 85.06°S and 85.06°N.  
- bounds: (-180.0, -85.06, 180.0, 85.06)  
Coordinate Operation:  
- name: Popular Visualisation Pseudo-Mercator  
- method: Popular Visualisation Pseudo Mercator  
Datum: World Geodetic System 1984 ensemble  
- Ellipsoid: WGS 84  
- Prime Meridian: Greenwich

- Set the basemap and basemap extent from contextily. Note that we'll specify the basemap tile provider with `source=cx.providers.OpenStreetMap.Mapnik`. To see other services that may be available, see [\(https://contextily.readthedocs.io/en/latest/providers\\_deepdive.html\)](https://contextily.readthedocs.io/en/latest/providers_deepdive.html).

```
basemap, basemap_extent = cx.bounds2img(*sf_311.total_bounds, zoom=15, ll=False,  
                                         source=cx.providers.OpenStreetMap.Mapnik)
```

Note that we needed to use `.total_bounds` again here since these need to be in web mercator.

In [ ]:

Plot all the 311 calls with a basemap set.

```
f, ax1 = plt.subplots(1, figsize=(20,20))
ax1.set_title("All 311 Calls November 2020 San Francisco")
plt.imshow(basemap, extent=basemap_extent)
sf_311.plot(ax=plt.gca(), marker='.', markersize=2, alpha=.75)
ax1.set_axis_off()
plt.axis()
```

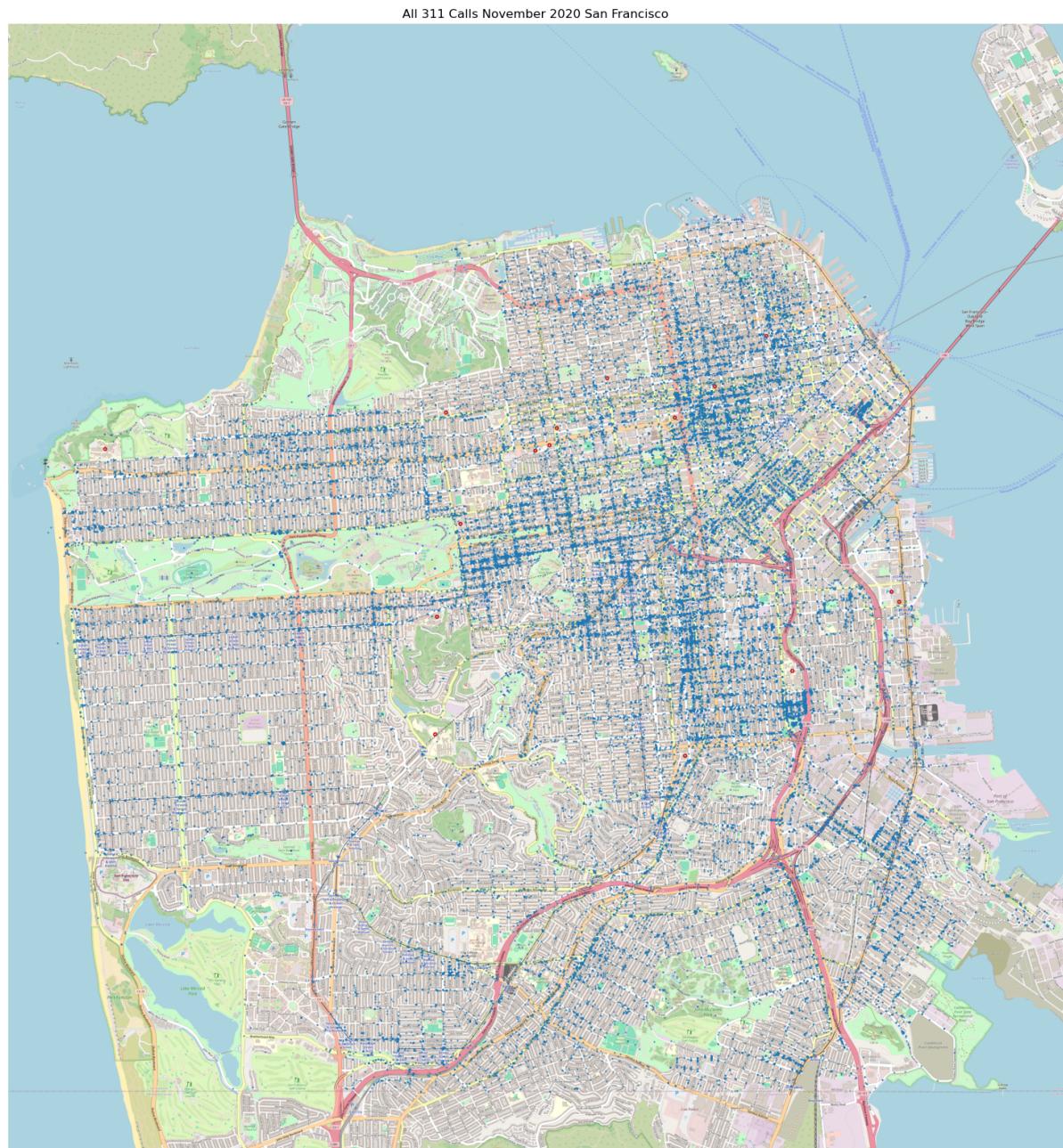
```
In [ ]: #
```

```
Out[ ]: Text(0.5, 1.0, 'All 311 Calls November 2020 San Francisco')
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x1dcaada2190>
```

```
Out[ ]: <AxesSubplot: title={'center': 'All 311 Calls November 2020 San Francisco'}>
```

```
Out[ ]: (-13638811.83098057, -13621689.93664469, 4537301.999008061, 4555646.88579650  
4)
```



⌚ In the above block of code, what does the "alpha" parameter control?

⋮

- Before we can decide how to best filter our data down, let's take a look at all the unique values in the "Category" field.

```
sf_311.Category.unique()
```

```
In [ ]: #  
Out[ ]: array(['Rec and Park Requests', 'Parking Enforcement', 'Noise Report',  
   'Street and Sidewalk Cleaning', 'Sewer Issues', 'Street Defects',  
   'Damaged Property', 'Sidewalk or Curb', 'Streetlights',  
   'General Request - PUC', 'Muni Employee Feedback',  
   'Blocked Street or Sidewalk', 'General Request - PUBLIC WORKS',  
   'Tree Maintenance', 'SFHA Requests',  
   'General Request - ANIMAL CARE CONTROL', 'Sign Repair',  
   'Encampments', 'Residential Building Request',  
   'Litter Receptacles', 'Graffiti', 'General Request - MTA',  
   'General Request - DPH', 'General Request - SFPD',  
   'Illegal Postings', 'General Request - BUILDING INSPECTION',  
   'General Request - ', 'General Request - ASSESSOR RECORDER',  
   'General Request - CHILDREN YOUTH FAMILIES',  
   'Muni Service Feedback', 'Homeless Concerns',  
   'DPW Volunteer Programs', 'Catch Basin Maintenance',  
   'General Request - PORT AUTHORITY', '311 External Request',  
   'Temporary Sign Request',  
   'General Request - ENTERTAINMENT COMMISSION',  
   'General Request - RENT BOARD',  
   'General Request - BOARD OF SUPERVISORS',  
   'General Request - CITYADMINISTRATOR GSA',  
   'General Request - PLANNING', 'General Request - FIRE DEPARTMENT',  
   'General Request - MOD', 'General Request - COUNTY CLERK',  
   'General Request - ENVIRONMENT',  
   'General Request - HUMAN SERVICES AGENCY',  
   'General Request - DISTRICT ATTORNEY', 'General Request - RPD',  
   'General Request - ELECTIONS', 'General Request - HSH',  
   'General Request - SHORT TERM RENTALS', 'General Request - DTIS',  
   'General Request - MOH', 'General Request - MOCD',  
   'General Request - 311CUSTOMERSERVICECENTER',  
   'General Request - ECONOMICS AND WORKFORCE DEVELOPMENT',  
   'Abandoned Vehicle'], dtype=object)
```

- We can create a new geodataframe that only contains the Category 'Parking Enforcement' and call it "parking\_issues".

```
parking_issues = sf_311.query("Category == 'Parking Enforcement'")  
parking_issues
```

In [ ]: #

Out[ ]:	<bound method NDFrame.head of osed			CaseID	Opened	C1
	Updated	Status	\			
1	13119442	11/8/2020 9:40	11/8/2020 9:45	11/8/2020 9:45	11/8/2020 9:45	Closed
3	13155970	11/18/2020 6:27	11/18/2020 7:27	11/18/2020 7:27	11/18/2020 7:27	Closed
4	13180413	11/24/2020 14:19	11/24/2020 17:06	11/24/2020 17:06	11/24/2020 17:06	Closed
5	13119107	11/8/2020 8:01	11/8/2020 8:22	11/8/2020 8:22	11/8/2020 8:22	Closed
6	13119507	11/8/2020 10:04	11/8/2020 10:12	11/8/2020 10:12	11/8/2020 10:12	Closed
...	...	...	...	...	...	...
47687	13182171	11/25/2020 6:45	11/25/2020 6:50	11/25/2020 6:50	11/25/2020 6:50	Closed
47688	13182140	11/25/2020 6:21	11/25/2020 7:17	11/25/2020 7:17	11/25/2020 7:17	Closed
47689	13106664	11/4/2020 13:44	11/4/2020 14:03	11/4/2020 14:03	11/4/2020 14:03	Closed
47737	13134578	11/12/2020 8:38	11/12/2020 8:44	11/12/2020 8:44	11/12/2020 8:44	Closed
47924	13097975	11/2/2020 13:37	11/2/2020 14:23	11/2/2020 14:23	11/2/2020 14:23	Closed
			Status_Notes \			
1	Case Resolved - Police Officer responded to re...					
3	Case Resolved - Officer responded to request u...					
4	Case Resolved - Officer responded to request u...					
5	Case Resolved - Officer responded to request u...					
6	Case Resolved - Police Officer responded to re...					
...			...			
47687	Case is a Duplicate - This issue has already b...					
47688	Case Resolved - Officer responded to request u...					
47689	Case Resolved - Officer responded to request u...					
47737	Case Resolved - Police Officer responded to re...					
47924	Case Resolved - Police Officer responded to re...					
			Responsible_Agency \			
1	Parking Enforcement Dispatch Queue	Parking Enforcement				
3	Parking Enforcement Dispatch Queue	Parking Enforcement				
4	Parking Enforcement Dispatch Queue	Parking Enforcement				
5	Parking Enforcement Dispatch Queue	Parking Enforcement				
6	Parking Enforcement Dispatch Queue	Parking Enforcement				
...		...	...			
47687	Parking Enforcement Dispatch Queue	Parking Enforcement				
47688	Parking Enforcement Dispatch Queue	Parking Enforcement				
47689	Parking Enforcement Dispatch Queue	Parking Enforcement				
47737	Parking Enforcement Dispatch Queue	Parking Enforcement				
47924	Parking Enforcement Dispatch Queue	Parking Enforcement				
			Request_Type \			
1	Other_Illegal_Parking		Parking Enforcement			
3	Other_Illegal_Parking		Blue - Chrysler - 5lrx746			
4	Other_Illegal_Parking		Blue - Honda - 8duj830			
5	Other_Illegal_Parking		Blue - Honda - Aw70c09			
6	Other_Illegal_Parking		Blue - Honda - Aw70c09			
...		...	...			
47687	Parking_on_Sidewalk	silver - cadillac escalade - 7wxn339				
47688	Parking_on_Sidewalk	silver - cadillac escalade - 7wxn339				
47689	Parking_on_Sidewalk	silver - cadillac escalade - 7wxn339				
47737	Other_Illegal_Parking	White - Chevy van - None				
47924	Other_Illegal_Parking	Blue - Ford suburban - Aksj				
			Address \			
1	57 INNES CT, SAN FRANCISCO, CA, 94124	INNES CT				
3	51 INNES CT, SAN FRANCISCO, CA, 94124	INNES CT				
4	51 INNES CT, SAN FRANCISCO, CA, 94124	INNES CT				

5	51 INNES CT, SAN FRANCISCO, CA, 94124	INNES CT	
6	51 INNES CT, SAN FRANCISCO, CA, 94124	INNES CT	
...	...	...	
47687	1380 LA PLAYA, SAN FRANCISCO, CA, 94122	LA PLAYA	
47688	1380 LA PLAYA, SAN FRANCISCO, CA, 94122	LA PLAYA	
47689	1380 LA PLAYA, SAN FRANCISCO, CA, 94122	LA PLAYA	
47737	Intersection of 48TH AVE and FULTON ST	48TH AVE	
47924	640 GREAT HWY, SAN FRANCISCO, CA, 94121	GREAT HWY	
1	Supervisor_District	Neighborhood Police District	Latitude \
3	10.0	Hunters Point	BAYVIEW 37.727079
4	10.0	Hunters Point	BAYVIEW 37.727363
5	10.0	Hunters Point	BAYVIEW 37.727356
6	10.0	Hunters Point	BAYVIEW 37.727256
...	...	...	...
47687	4.0	Outer Sunset	TARAVAL 37.760715
47688	4.0	Outer Sunset	TARAVAL 37.760715
47689	4.0	Outer Sunset	TARAVAL 37.760715
47737	1.0	Golden Gate Park	RICHMOND 37.771371
47924	1.0	Sutro Heights	RICHMOND 37.775573
1	Longitude	Point	Source \
3	-122.367097	(37.72707865, -122.36709653)	Mobile/Open311
4	-122.367930	(37.72736328, -122.36792978)	Mobile/Open311
5	-122.367940	(37.72735638, -122.36793984)	Mobile/Open311
6	-122.367988	(37.72725562, -122.36798812)	Mobile/Open311
...	...	...	...
47687	-122.509000	(37.7607147, -122.5090004)	Web
47688	-122.509000	(37.7607147, -122.5090004)	Web
47689	-122.509000	(37.7607147, -122.5090004)	Web
47737	-122.509327	(37.77137094, -122.50932645)	Mobile/Open311
47924	-122.511296	(37.77557256, -122.51129619)	Mobile/Open311
1	geometry		
3	POINT (-13621842.872 4540942.599)		
4	POINT (-13621935.635 4540982.659)		
5	POINT (-13621936.748 4540981.687)		
6	POINT (-13621942.125 4540967.506)		
...	...	...	
47687	POINT (-13637639.542 4545677.756)		
47688	POINT (-13637639.542 4545677.756)		
47689	POINT (-13637639.542 4545677.756)		
47737	POINT (-13637675.843 4547178.350)		
47924	POINT (-13637895.109 4547770.075)		

[5054 rows x 20 columns]>

Let's plot the parking issues only and then set the extent of our map to the total extent of the parking issues geodataframe.

```
f, ax1 = plt.subplots(1, figsize=(20,20))
ax1.set_title("Parking Related 311 Calls November 2020 San Francisco")
plt.imshow(basemap, extent=basemap_extent)
parking_issues.plot(ax=plt.gca(), marker='.', markersize=10, alpha=1)
ax1.set_axis_off()
plt.axis()
```

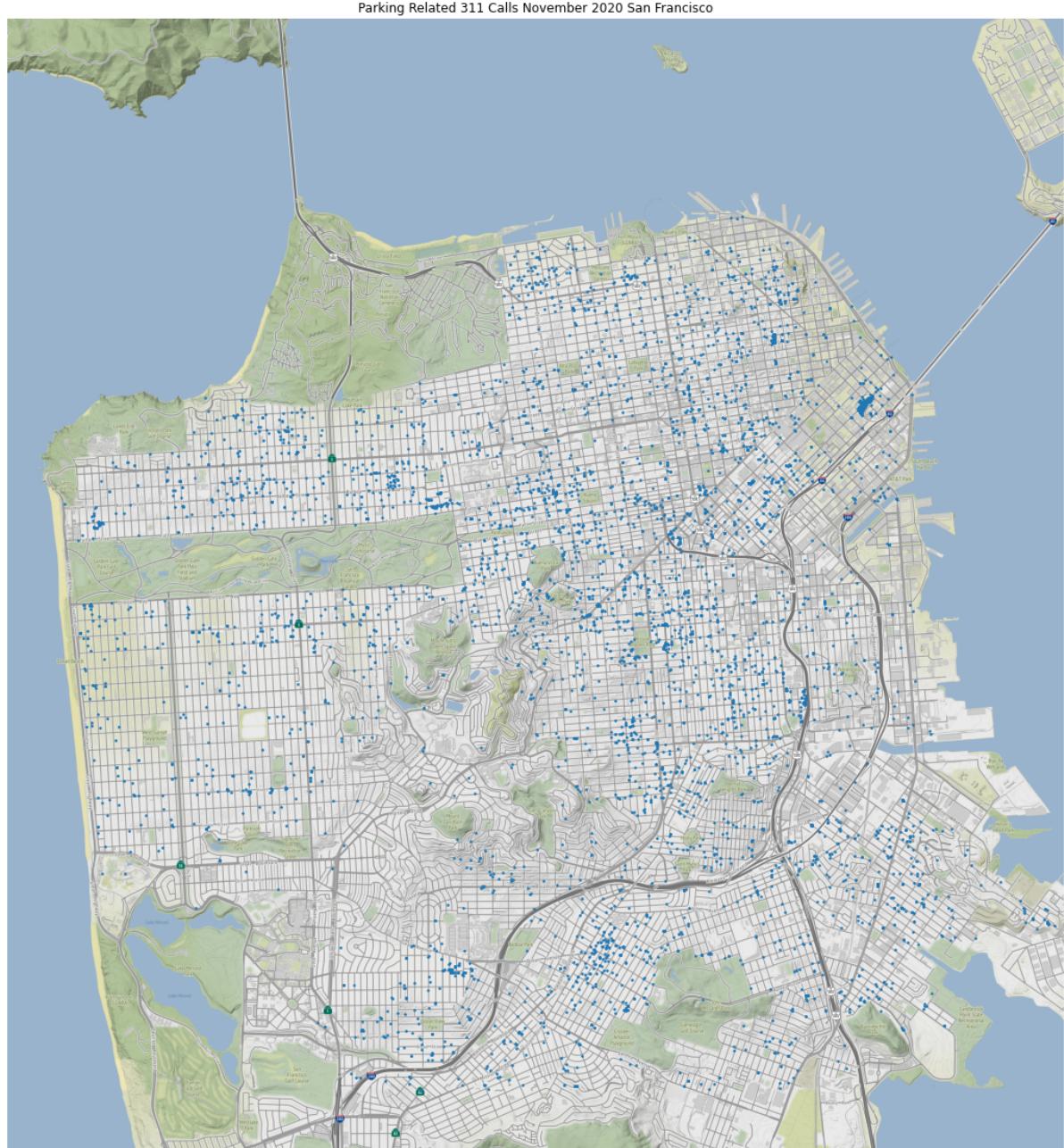
```
In [ ]: #
```

```
Out[ ]: Text(0.5, 1.0, 'Parking Related 311 Calls November 2020 San Francisco')
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x170591698c8>
```

```
Out[ ]: <AxesSubplot:title={'center':'Parking Related 311 Calls November 2020 San Francisco'}>
```

```
Out[ ]: (-13638811.83098057, -13621689.93664469, 4537301.999008061, 4555646.885796504)
```



Next, we'll do a spatial join between our 311 parking issues and San Francisco neighborhoods. But before we can run that operation, let's set the projection of our `sf_neighborhoods` to match our 311 data.

```
sf_neighborhoods = sf_neighborhoods.to_crs('EPSG:3857')
```

In [ ]: #

Here we can calculate the count of parking issues by neighborhood and then add a field to our neighborhoods data and call it "parking\_incidents". Then examine the sf\_neighborhoods geodataframe.

```
hood_counts = gpd.sjoin(sf_neighborhoods, parking_issues, how='left', op='contains')\
    .groupby('name').index_right.count()
sf_neighborhoods['parking_incidents'] = hood_counts.values
sf_neighborhoods
```

In [ ]: #

Out[ ]:

		link	name	geometry	parking_incidents
0	http://en.wikipedia.org/wiki/Sea_Cliff,_San_Fr...		Seacliff	POLYGON ((-13635909.066 4548889.167, -13635939...))	20
1		None	Lake Street	POLYGON ((-13635207.246 4548926.811, -13635222...))	14
2	http://www.nps.gov/prsf/index.htm		Presidio National Park	POLYGON ((-13634141.858 4552759.779, -13634091...))	15
3		None	Presidio Terrace	POLYGON ((-13633566.376 4549428.413, -13633409...))	1
4	http://www.sfgate.com/neighborhoods/sf/innerri...		Inner Richmond	POLYGON ((-13633590.339 4549283.086, -13632945...))	13
...	...	...	...	...	...
112	http://en.wikipedia.org/wiki/Corona_Heights,_S...		Corona Heights	POLYGON ((-13629422.779 4545953.181, -13629437...))	5
113	http://en.wikipedia.org/wiki/Haight-Ashbury		Ashbury Heights	POLYGON ((-13631289.585 4545785.263, -13631305...))	91
114	http://en.wikipedia.org/wiki/Eureka_Valley,_Sa...		Eureka Valley	POLYGON ((-13629662.606 4545908.690, -13629628...))	0
115	http://en.wikipedia.org/wiki/St._Francis_Wood,...		St. Francis Wood	POLYGON ((-13633472.736 4542016.274, -13633109...))	9
116	http://en.wikipedia.org/wiki/Neighborhoods_in_...		Sherwood Forest	POLYGON ((-13632062.910 4542836.706, -13632047...))	0

117 rows × 4 columns

Here we can plot all our 311 parking issues by neighborhood.

```
f, ax1 = plt.subplots(1, figsize=(15,15))
ax1.set_title("Parking Related 311 Calls By Neighborhood")
plt.imshow(basemap, extent=basemap_extent)
sf_neighborhoods.plot('parking_incidents', ax=plt.gca(),
                      cmap='Wistia', alpha=.5, legend=True, legend_kwds=
{'orientation':'horizontal'})
ax1.set_axis_off()
plt.axis(parking_issues.total_bounds[[0,2,1,3]])
```

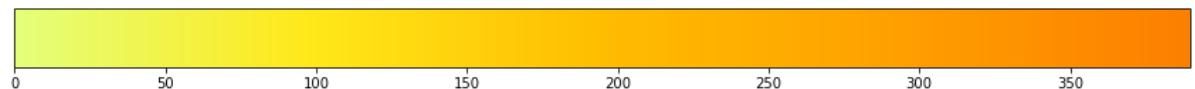
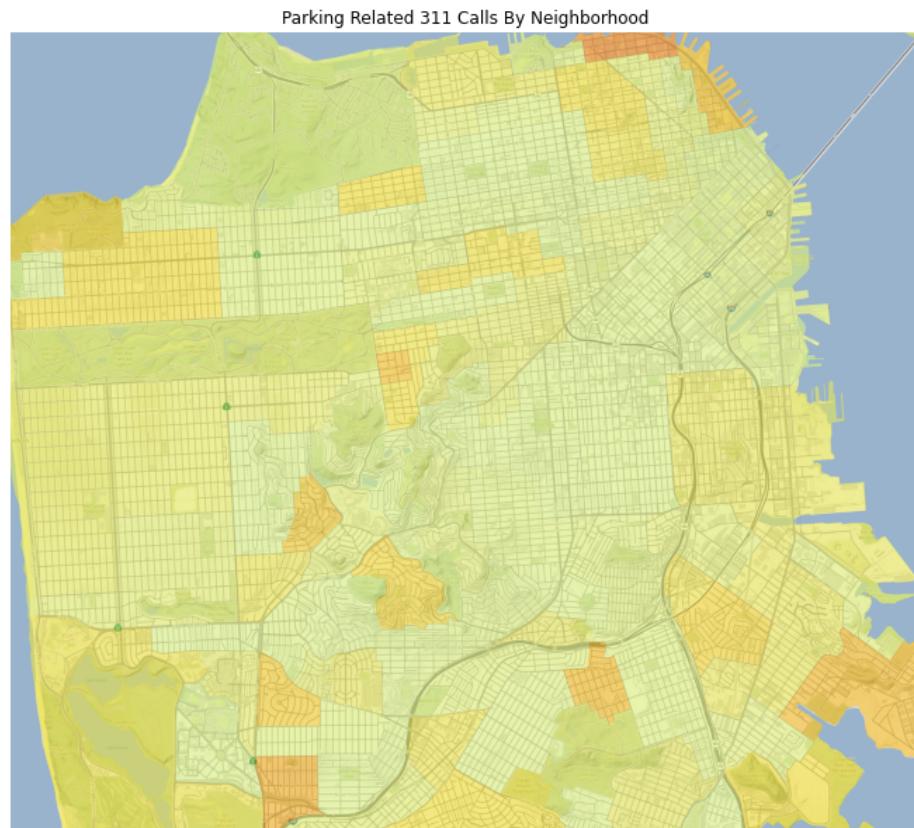
```
In [ ]: #
```

```
Out[ ]: Text(0.5, 1.0, 'Parking Related 311 Calls By Neighborhood')
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x17060258188>
```

```
Out[ ]: <AxesSubplot:title={'center':'Parking Related 311 Calls By Neighborhood'}>
```

```
Out[ ]: (-13637895.109407913,  
         -13621842.872231368,  
         4538284.832547026,  
         4552426.083463726)
```



**Turn in your iPython notebook once you have completed this exercise.**

- ➔ This directs you to do something specific, maybe in the operating system or answer something conceptual.
- ☒ Code to just run, typically boilerplate.
- ⌨ Coding you need to write, in the subsequent code cell.
- ❓ Questions to answer in the same markdown cell.
- Prompt for an interpretation or answering the question.

# Introduction to arcpy

In this notebook, we'll start to explore using ArcGIS via the `arcpy` module, which will provide us access to all of the geoprocessing capabilities of ArcGIS. We'll look at:

- Getting access to arcpy and some key environments
  - *Where's my data?* Using `env` and `os` to help
  - Using the `os` module to work with multiple workspaces
- Creating lists of arcpy objects
  - List of FeatureClasses or a list of Rasters
  - Multiple lists within a workspace
  - List of fields
  - List of workspaces
- Using geoprocessing tools
  - Toolbox shortcuts
  - Getting help directly from the code cell
  - Spatial Analyst tools & map algebra
  - Processing as a numpy array
- Using Describe & Exists for geoprocessing
  - Describe
    - Using Describe to get extent
    - Using the extents to trim a raster
    - Using Describe with imagery (multiple bands)
    - Other Describe objects
  - Exists
    - Detect if a dataset exists
    - Delete before creating a new dataset
    - Detect if a field exists
  - Create a smaller workspace using Exists and Describe in a loop
  - Convert geodatabase feature classes to shapefiles
  - Feature to raster

The next notebook will then get into

- Debugging and messaging
- Data management
- Cursors
- Using geopandas together with arcpy
- Using pandas with arcpy

This exercise can either be run within ArcGIS in a notebook window (where outputs will go to maps) or from an IDE running Jupyter notebooks outside (I recommend VS-code), as long as you're continuing to use Python kernel cloned from the ArcGIS Pro installation.

```
In [ ]: from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

## Getting access to arcpy and setting some key environments

The `arcpy` module gives Python access to ArcGIS and geoprocessing. Some of these are geoprocessor-specific things, others are the 400+ GIS tools (like buffer, clip, etc.) that you normally use in the ArcToolbox.

- Importing `arcpy` is always going to be required, and is such an essential requirement that this step has already been done for you in the notebook environment provided by ArcGIS Pro. If you're working in ArcGIS Pro, it doesn't hurt to include it anyway just so your notebook will work in either place.

```
import arcpy
```

In [ ]:

## Where's my data?: Using env and os to help

We'll start right at the beginning to take control of where our data is located on my computer/network. This is one of the *big three* sources of problems in GIS (the other two being *Where's my data on the planet (coordinate systems)* and *typos*). We'll use environment settings and the `os` module to help.

- First we'll create a shortcut to the environment setting object `arcpy.env` by simply shortening that to `env`, by either:

```
from arcpy import env
```

or

```
env = arcpy.env
```

In [ ]: #

- As you know by now with working with ArcGIS, there are many environment settings, and we'll explore others along the way. But one essential environment setting provides access to your data, which needs the location of the workspace.

```
print(env.workspace)
```

In [ ]: #

```
C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy\arcpy.gdb
```

If you're running this from the Notebook interface in ArcGIS Pro, you're going to see the workspace associated with the ArcGIS Pro project, probably a geodatabase. However, if you are running this in an IDE like VS-code, you probably see `None` displayed. We will want to be able to work from either location, so we're going to need to pay attention to how we store our data, *which should also use relative paths*, and work with the `env.workspace` setting. The `os` module will help for that...

## Using the `os` module to work with multiple workspaces

To access some data while avoiding absolute paths, we'll need to use the very useful `os` module we looked at earlier. We have some geodatabases residing in the project folder that can be accessed by using a relative path to get there by going *up* one level to the project folder and then *down* into the geodatabase or other workspace folder.

project folder

- `hmb.gdb`
  - `city`
  - `elev`
  - `geology`
  - `...`
- `pen.gdb`
  - `cities`
  - `faultcov_arc`
  - `geol`
  - `landusePen`
  - `water`
- *other geodatabases / other data folders*
- *various notebook .ipynb files, like this one*
- `...`

If you've set things up right, the `.ipynb` files should be in the main project folder for an ArcGIS project. In my case, this folder is `Ex07_08_arcpy` and the default workspace is `arcpy.gdb`, but yours may differ. It doesn't really matter what the project folder is named or what its path is if we work with relative paths. But you should have copied various geodatabases such as `hmb.gdb`, `pen.gdb` and `marbles.gdb`, and since they should be at the same level, we can navigate to them by going *up* to the project folder and then down into a parallel geodatabase.

We're going to need to have the path to the project folder to get our code working without a lot of manual editing. One approach that would work if you are working in ArcGIS is to use `os.path.dirname` with the current workspace to provide the folder that holds the geodatabase, and so that would be the project folder:

```
import os
proj = os.path.dirname(env.workspace)
```

*However this isn't going to work for code run from Notebooks outside ArcGIS, so we need a method that works in either location.*

▀ So instead we'll use another `os` method: `os.getcwd()` which returns the path to the folder we started in, either in the IDE or to open the `.aprx` (*but*).

```
proj = os.getcwd()  
proj
```

This mostly doesn't work if you're using ArcGIS Pro from the start menu, so I've learned to always start Pro by opening an existing project by opening it from the .aprx in a Windows folder. The exception is if you're creating a new project in Pro; this will generally provide the path you'd expect.

In [ ]: #

Out[ ]: 'C:\\\\Users\\\\900008452\\\\Box\\\\course\\\\625\\\\exer\\\\Ex07\_08\_arcpy'

Before continuing, *make sure that this shows the folder where this notebook is stored*, which should be where your geodatabases and other data are located, as an absolute path. Also, *make sure you understand where your files and data are stored (including their absolute paths) and clearly understand how relative paths help you with this*.

You'll also want to understand the various path delimiters like /, \ and \\ : remember that \ is an escape character, thus the need for using \\ to provide the \ that Windows tends to use.

We're going to want to deal with multiple workspaces, so we will need to learn some methods. Basically, the two options are:

1. Set the workspace with env.workspace ; for instance env.workspace = proj + "\\pen.gdb"
2. Use with to access the workspace just within a code block

I'm increasingly using the with method, since once you get used to it, it can simplify your code and help you avoid making mistakes when it's set wrong. In general, environment settings can create problems (it's almost one of the "Big Three" but that would make it the "Big Four"), so we should look at the with method right away.

We'll start with a common need: to access lists of data in folders, one of the most important methods in GIS programming, since GIS is so data-centric.

## Creating lists of arcpy objects

There is a set of `arcpy` methods that create lists of `arcpy` objects, like feature classes. Each of the following returns a list that you can process.

- `ListFeatureClasses`
- `ListRasters`
- `ListFiles`
- `ListWorkspaces`
- `ListFields`
- `ListTables`
- `ListDatasets`

*Navigating through lists like these is the first types of operations where we can see Python coding helping us do our work, in this case for managing our data, letting a script perform tedious repeated operations.*

### List of feature classes or a list of rasters

▀ If we changed the workspace to one with feature classes, we can get a list displayed. Now that we're in the `pen.gdb`, we can see what feature classes are in it with:

```
arcpy.ListFeatureClasses()
```

You should know by now to *pay attention to capitalization*. Everything but Windows filenames are case-sensitive. Typically `arcpy` methods including geoprocessing functions will use "Camel case" where words in the middle of a method name (like `Feature` and `Classes`) are capitalized so the entire method name has humps like a camel: `ListFeatureClasses`. (Sometimes the camel-case analogy is taken even further so you might have a name looking like `myCamelCase` with no hump at the start, but that's not the case with `arcpy` methods.)

We could change the workspace, we'll use the `with` structure:

```
with arcpy.EnvManager(workspace='pen.gdb'):  
    arcpy.ListFeatureClasses()
```

In [ ]: #

Assuming you've set up your folders right, and the "pen.gdb" folder is in the folder where your notebook .ipynb file is located, the above should display the list of Now that we have a list, we can loop through it to perform some operation on it, probably with the same with structure, but that depends on what you're doing.

But we haven't looked at geoprocessing tools yet, so we'll just list the name and length (of the name), in a for loop structure; later we'll use a loop structure run geoprocessing tools on each feature class (or a selection) in a given workspace:

```
with arcpy.EnvManager(workspace='pen.gdb'):
    for f in arcpy.ListFeatureClasses():
        print(f"name: {f} length: {len(f)}")
```

In [ ]: #

```
name: cities length: 6
name: faultcov_arc length: 12
name: geol length: 4
name: water length: 5
name: landusePen length: 10
name: urban length: 5
```

But we can use ListFeatureClasses to select those that are a particular type of geometry, like point, line or polygon:

```
with arcpy.EnvManager(workspace='pen.gdb'):
    for dtype in ["point", "line", "polygon"]:
        print(dtype)
        for f in arcpy.ListFeatureClasses(feature_type=dtype):
            print(f"name: {f}")
```

In [ ]: #

```
point
line
name: faultcov_arc
polygon
name: cities
name: geol
name: water
name: landusePen
name: urban
```

Now let's look at the other method -- actually changing the workspace -- to "hmb.gdb" with `env.workspace = "hmb.gdb"` and then try the same code as above to see those feature classes. We'll reset the workspace back to the original at the end.

```
env.workspace = "hmb.gdb"
for dtype in ["point", "line", "polygon"]:
    print(dtype)
    for f in arcpy.ListFeatureClasses(feature_type=dtype):
        print(f"name: {f}")
env.workspace = proj
```

In [ ]: #

```
point
name: pourpoint
name: landing
line
name: streams
name: faults
name: roads
polygon
name: areaclip
name: publands
name: geolstr200
name: geol
name: stbuff200
```

Sometimes you'll find that you want to actually change the workspace, for instance when you're doing a lot of steps in that workspace. It all depends on code readability, which is generally helped by having less code to look at. You'll need to decide which works best for your situation, but we'll mostly make use of the `with` structure.

Now use `ListRasters` to see the list of rasters. In this case, we'll do this all on one line, but we'll start by inventing a method `ENV` to access the `arcpy.EnvManager` method; we can keep using that `ENV` shortcut later to shorten our code.

```
ENV = arcpy.EnvManager
with ENV(workspace="hmb.gdb"): arcpy.ListRasters()
```

In [ ]: #

From here on, you'll need to remember how to handle the workspace setting, either by changing and resetting it, or using `with` structures, which I recommend. I'll just provide you with any new code you'll need.

## Selecting a subset

▀ We can select a subset based on the name, to display all of those starting with a `e`.

```
arcpy.ListRasters("e*")
```

Later we'll look at other properties of data sets with the `.Describe` method that might allow us to just look at features of a given property.

In [ ]: #

## Multiple lists within a workspace

We'll do several things within the `with` structure to demonstrate how several operations can happen, and also see the effect of the `with` structure. I'll give you all of the code here, though it assumes you've previously defined `proj`, `env`, and `ENV`.

```
with ENV(workspace = proj + "\hmb.gdb"):
    print("Workspace:")
    print(env.workspace)
    print("\nRasters:")
    print(arcpy.ListRasters())
    print("\nElevation rasters:")
    print(arcpy.ListRasters("elev*"))
    print("\nFeatureClasses:")
    print(arcpy.ListFeatureClasses())
print("\nWorkspace outside the with structure:")
print(env.workspace)
```

In [ ]:

#

```

Workspace:
C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy\hmb.gdb

Rasters:
['watergrd', 'geology', 'landuse', 'pub', 'newdev', 'city', 'elev', 'elev30',
 'numpyarraytoraster_f258ed9f_eec4_429b_8182_dbd92283d1c0_284017208', 'numpyarraytoraster_dbbe317f_ddb6_440b_8a5e_36af55bdad7b_284017208', 'numpyarraytoraster_13b48e1f_9a2f_4694_b3b6_e22866798da1_2453623504', 'numpyarraytoraster_95fa8d4c_0930_4ab2_a851_e38fb3359cf_2453623504', 'numpyarraytoraster_445f664f_38d8_4a15_be0b_f770a0be3df1_401611712', 'numpyarraytoraster_fc5bc198_377a_4502_ae67_77a6c4662b91_401611712', 'numpyarraytoraster_249bae79_4b16_4f0b_b510_93f6befada75_2888410860', 'numpyarraytoraster_2d8b9707_cc96_47d1_92ad_c7a458a74fee_2888410860', 'numpyarraytoraster_83003b86_b865_4c3b_a796_98bd8e4c52b3_2888410860', 'numpyarraytoraster_0270248f_2469_4a08_b20a_0d192898984f_2470410924', 'numpyarraytoraster_8cb26adc_892d_437c_a498_b9f3ef151698_2470410924', 'numpyarraytoraster_d5ff6afb_d750_428f_bb6e_a6ad6c302c54_2470410924', 'steepurban', 'numpyarraytoraster_29fe2ea7_0b00_4e3f_a528_bcae4b549928_2385214608', 'highElev_np', 'numpyarraytoraster_3513d5c7_61af_488f_ba7d_c431c1bf698b_2385214608', 'trimmed_elev']

Elevation rasters:
['elev', 'elev30']

FeatureClasses:
['streams', 'areaclip', 'faults', 'pourpoint', 'publands', 'roads', 'landing', 'geolstr200', 'geol', 'stbuff200']

Workspace outside the with structure:
C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy

```

## List of fields

▀ The following code creates a list of fields from a data table. Look up `ListFields` in the help system so you can understand why, while it *creates no error*, printing the field object might not be that useful. Consider what the list is composed of, in contrast to what we found for feature classes and rasters.

```

with ENV(workspace = proj + "\pen.gdb"):
    flds = arcpy.ListFields("geol")
    for fld in flds:
        print(fld)

```

In [ ]: #

```
<geoprocessing describe field object object at 0x0000019A2E587EB0>
<geoprocessing describe field object object at 0x0000019A2E587E30>
<geoprocessing describe field object object at 0x0000019A2E587F50>
<geoprocessing describe field object object at 0x0000019A2E587E50>
<geoprocessing describe field object object at 0x0000019A2E587F70>
<geoprocessing describe field object object at 0x0000019A2E587F10>
<geoprocessing describe field object object at 0x0000019A2E587EF0>
<geoprocessing describe field object object at 0x0000019A2E587ED0>
<geoprocessing describe field object object at 0x0000019A2E587E90>
```

Now we'll try to fix it:

Fields are complex objects, not simple strings. Since `ListFields` creates a list of objects that are not easy to print, like the text strings returned by all of the other lists, this doesn't create an error, and similarly can't be displayed interactively like the ones above. Also note that this complex object is assigned to the variable `fld`.

Fix the last line by adding the property `.name` to the `fld` object and try it again. Then replace the print statement with: `print("{} is type {} with length {}".format(fld.name, fld.type, fld.length))`

In [ ]: #

```
OBJECTID is type OID with length 4
Shape is type Geometry with length 0
AREA is type Double with length 8
PERIMETER is type Double with length 8
GEOLOGY_ is type Double with length 8
GEOLOGY_ID is type Double with length 8
TYPE_ID is type Integer with length 4
Shape_Length is type Double with length 8
Shape_Area is type Double with length 8
```

Alternative method to get a list of fields, using *list comprehension*, the syntax:

```
newlist = [ expression for item in iterable if condition == True]
```

So for our example, we don't need an if condition but we can do it this way:

```
with ENV(workspace = proj + "\pen.gdb"):
    fldnames = [f.name for f in arcpy.ListFields("geol")]
    fldnames
```

In [ ]: #

## List of workspaces

▀ Our project folder has multiple workspaces. Let's use .ListWorkspaces to display a list, but let's do more than that and navigate through all of them and list each feature class of a given data type and each raster. We'll limit the type of workspace to "FileGDB".

```
for ws in arcpy.ListWorkspaces("*","FileGDB"):
    print(ws)
    with ENV(workspace = ws):
        if arcpy.ListFeatureClasses():
            print("There are {} FeatureClasses:".format(len(arcpy.ListFeatureClasse
s())))
            for dtype in ["point", "line", "polygon"]:
                fs = arcpy.ListFeatureClasses(feature_type=dtype)
                if fs:
                    print("\t{}".format(dtype))
                    for f in fs:
                        print("\t\t{}".format(f))
    rass = arcpy.ListRasters()
    if rass:
        print("There are {} Rasters:".format(len(rass)))
        for ras in rass:
            print("\t{}".format(ras))
```

In [ ]: #

```
C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy\arcpy.gdb
There are 1 FeatureClasses:
    point
        samplePts_MeanCenter
There are 5 Rasters:
    Extract_newd1
    Extract_pub1
    numpyarraytoraster_c6525966_f1c5_431b_b716_12ff456bdb6_1806419640
    numpyarraytoraster_254a82f0_c9f2_4f87_ab4b_2f25d21acad4_1787618684
    numpyarraytoraster_e0f844ec_c5ee_44b1_af76_37c43839e3a8_1787618684
C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy\bozo.gdb
C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy\hmb.gdb
There are 10 FeatureClasses:
    point
        pourpoint
        landing
    line
        streams
        faults
        roads
    polygon
        areaclip
        publands
        geolstr200
        geol
        stbuff200
There are 25 Rasters:
    watergrd
    geology
    landuse
    pub
    newdev
    city
    elev
    elev30
    numpyarraytoraster_f258ed9f_eec4_429b_8182_db92283d1c0_284017208
    numpyarraytoraster_dbbe317f_ddb6_440b_8a5e_36af55bdad7b_284017208
    numpyarraytoraster_13b48e1f_9a2f_4694_b3b6_e22866798da1_2453623504
    numpyarraytoraster_95fa8d4c_0930_4ab2_a851_e38fdb3359cf_2453623504
    numpyarraytoraster_445f664f_38d8_4a15_be0b_f770a0be3df1_401611712
    numpyarraytoraster_fc5bc198_377a_4502_ae67_77a6c4662b91_401611712
    numpyarraytoraster_249bae79_4b16_4f0b_b510_93f6befada75_2888410860
    numpyarraytoraster_2d8b9707_cc96_47d1_92ad_c7a458a74fee_2888410860
    numpyarraytoraster_83003b86_b865_4c3b_a796_98bd8e4c52b3_2888410860
    numpyarraytoraster_0270248f_2469_4a08_b20a_0d192898984f_2470410924
    numpyarraytoraster_8cb26adc_892d_437c_a498_b9f3ef151698_2470410924
    numpyarraytoraster_d5ff6afb_d750_428f_bb6e_a6ad6c302c54_2470410924
    steepurban
    numpyarraytoraster_29fe2ea7_0b00_4e3f_a528_bcae4b549928_2385214608
    highElev_np
    numpyarraytoraster_3513d5c7_61af_488f_ba7d_c431c1bf698b_2385214608
    trimmed_elev
C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy\HMBcity.gdb
There are 2 Rasters:
    elev
    elev30
C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy\marbles.gdb
```

There are 11 FeatureClasses:

- point
  - co2july95
  - samples
  - marblePts
- line
  - streams
  - trails
  - contours10m
  - cont
- polygon
  - geology
  - veg
  - water
  - watrshed

There are 3 Rasters:

- elev
- elev10
- geolgrd

C:\Users\900008452\Box\course\625\exer\Ex07\_08\_arcpy\pen.gdb

There are 6 FeatureClasses:

- line
  - faultcov\_arc
- polygon
  - cities
  - geol
  - water
  - landusePen
  - urban

There are 1 Rasters:

- landusePenras

C:\Users\900008452\Box\course\625\exer\Ex07\_08\_arcpy\SF.gdb

There are 3 FeatureClasses:

- point
  - SF\_Schools
  - BA\_TransitStops
- line
  - BA\_BikeRoutes

C:\Users\900008452\Box\course\625\exer\Ex07\_08\_arcpy\testPath.gdb

# Using geoprocessing tools

As we've seen, Geoprocessing tools used in a script are the same tools as are used in ArcToolbox, but we access them by providing the tool name. But remember that there may be more than one tool by a given name -- for instance, there are two Clip tools, one vector-based in the Analysis toolbox, one raster-based in Data Management Tools. Each is distinct, working with different types of data, but how do we distinguish them in a script? In ArcToolbox, we select which clip we wish to use, but in a script we have to distinguish them in some way. To do this we use aliases, which become part of the tool name.

- Go to the help for the Buffer and Clip tools of the Analysis Toolbox, and then look at their scripting syntax. Note that the syntax gives each an alias that identifies which toolbox it comes from, in this case analysis.

```
Clip_analysis (in_features, clip_features, out_feature_class, cluster_tolerance)
```

```
Buffer_analysis (in_features, out_feature_class, buffer_distance.....)
```

- 🌐 Make sure you have a map open and available to display results along the way.

Before we use these tools, let's learn about some shortcuts and other ways to get help.

## Toolbox shortcuts

Each of the geoprocessing toolboxes has an *alias*, which is useful since they become part of the name of the tool. This has been needed since tools in different toolboxes may have the same name (though ArcGIS is moving away from that). So the full name of a tool includes its alias, such as the `Clip` tool in the Analysis toolbox is called with `Clip_analysis` or more fully `arcpy.Clip_analysis` and `Slope` in the Spatial Analysis toolbox (alias `sa`) is called as `arcpy.Slope_sa`. We can shorten the alias even further by creating variable shortcuts with unrequired but standard codes by defining them as for instance `from arcpy import analysis as AN`. Here are some of the toolbox aliases and codes:

toolbox	alias	code
Analysis	analysis	AN
Conversion	conversion	CV
Data Management	management	DM

So to set up these shortcuts, we could include the following code:

```
from arcpy import analysis as AN
from arcpy import conversion as CV
from arcpy import management as DM
```

... then we can simply call Clip for instance as:

```
AN.Clip()
```

For Spatial Analyst, we can make the nice Map Algebra syntax work by importing everything and not having to use a shortcut with:

```
from arcpy.sa import *
```

This is also done with Image Analyst (`arcpy.ia`).

- Write some boilerplate that creates all of these toolbox shortcuts, and also imports `arcpy` and `os`, creates the shortcut to `arcpy.env` as `env`, and sets `env.overwriteOutput = True`. Then assign a new variable `proj` as `os.getcwd()` -- this will be useful in specifying the path to the project folder. And finally set the workspace to `hmb.gdb`.

In [ ]: #

In [ ]: env.workspace

Out[ ]: 'hmb.gdb'

Note that in the above code, we just set the workspace, since we are only working in that one place, so that makes for less indented code. It would also work by wrapping everything in a `with` block. *Your choice.*

## Getting help directly from the code cell

It makes your life easier to access help directly from a code cell. Once you have your shortcuts, you can easily get help with the parameters for various tools.

- Use `help()` to get help for `AN.Clip`:

In [ ]:

```
Help on function Clip in module arcpy.analysis:
```

```
Clip(in_features=None, clip_features=None, out_feature_class=None, cluster_tolerance=None)
    Clip_analysis(in_features, clip_features, out_feature_class, {cluster_tolerance})
```

Extracts input features that overlay the clip features.

### INPUTS:

`in_features` (Feature Layer / Scene Layer / Building Scene Layer / File):  
The features that will be clipped.

`clip_features` (Feature Layer):

The features that will be used to clip the input features.

`cluster_tolerance` {Linear Unit}:

The minimum distance separating all feature coordinates as well as the

distance a coordinate can move in x or y (or both). Set the value higher for data with less coordinate accuracy and lower for data with

h

extremely high accuracy. Changing this parameter's value may cause failure or unexpected

results. It is recommended that you do not modify this parameter. It has been removed from view on the tool dialog box. By default, the input feature class's spatial reference x,y tolerance property is used.

### OUTPUTS:

`out_feature_class` (Feature Class / File):  
The dataset that will be created.

- Note the parameter names, and call the Clip function with `in_features = proj + "/pen.gdb/geol"`, `clip_features` to `"areaclip"` , and `out_feature_class` to `"geol"` .

I recommend using parameter names *explicity* (so `clip_features="areaclip"` for instance), since even though verbose it makes your code more readable, *and* you can skip over unneeded parameters.

In [ ]: #

*What's the `msg =` part for?* It's simply to avoid the code chunk output for displaying the word "Messages". All geoprocessing tools, like `Clip` , are objects that have a value, and normally that value is a set of messages of how the tool ran. Sometimes you want to see these, to help debug a problem that doesn't cause the tool to fail, what's called a "logic error". But when everything is working fine, we can avoid seeing the "Messages" header displayed by simply assigning the tool result to a variable we'll just call `msg` . *Don't confuse these with tool outputs.*

It's fine to not use the `msg =` if you don't mind seeing "Messages" (and I won't include that trick in code suggestions later in this notebook), but later on we'll be running code that run a lot of tools, so we'll see a lot of "Messages" printed out unless we use this trick.

- Now see what you got on the map (this is where it's handy to be in ArcGIS Pro), and make sure you understand what was done, what was named and where inputs were accessed and outputs were stored.

- Now do the same thing for AN.Buffer -- use help to check the parameter names etc. -- and run it to create an output feature class "stbuff200" with "streams" (in hmb.gdb) as input features and a buffer distance of 200. *Check the naming of parameters carefully.* ● Then check the output on the map.

In [ ]: #

## Spatial Analyst tools & Map Algebra

As we've already seen, we can now use map algebra and create raster objects. Map algebra was invented by Dana Tomlin in the early 1980s.

The map algebra method involves creating *raster objects* created either from raster data or from tools that typically access other raster objects.

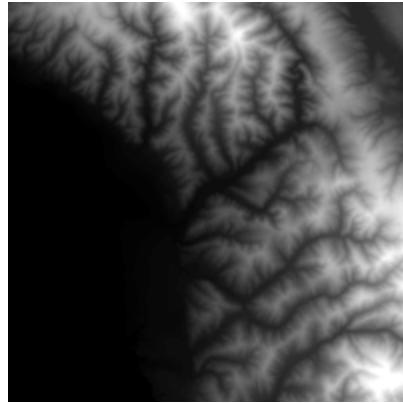
- In ArcGIS Pro, add 'elev', 'landuse', and 'geology' rasters from hmb to a new map. We'll be accessing these rasters by name and start by assigning them to new raster objects.

```
elev = Raster('elev')
landuse = Raster('landuse')
geology = Raster('elev')
elev
```

Note that only the last object provided in the code cell is displayed. This is similar to what we've seen before with dataframes. This is telling us that code cells work somewhat like functions that return one item.

In [ ]: #

Out[ ]:



- Use a series of map algebra statements to end with a `steepurban` raster object that represents slopes > 10 and landuse < 20 (urban):

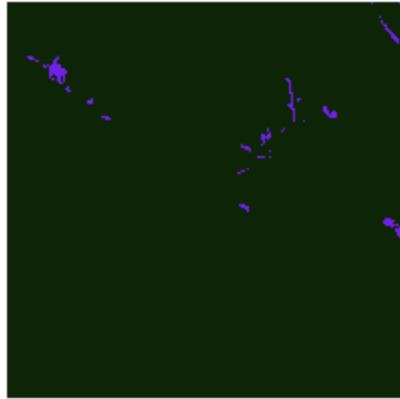
```
steep = Slope("elev") > 10
urban = Raster("landuse") < 20
steepurban = steep & urban
steepurban.save("steepurban")
steepurban
```

#### Notes:

- The `landuse` raster required assigning to a raster object before we could query it essentially with `landuse < 20`, so this required `Raster(landuse) < 20`. This may be surprising, since we didn't have to make a raster object out of `elev` before deriving the slope of it; this is because the `Slope` tool is expecting an elevation raster so it converts it for you.
- Raster objects are retained in memory, but not retained in your workspace, requiring you to save them to have them available later without having to run the code again.
- You should see a result displayed only after you've saved the result as shown above. The temporary rasters `steep` and `urban` won't similarly display, since we haven't saved them.

```
In [ ]: #
```

```
Out[ ]:
```



## Processing as a numpy array

Numpy arrays provide similar map algebra capabilities as we have just been using in ArcGIS, and add some other specialized capabilities in its numerical methods, such as 2D Fourier transforms. One limitation to ndarrays however is that they don't use a geospatial coordinate referencing system, so if we process something and bring it back into ArcGIS we'll need to establish the crs.

We'll try to a couple of operations, one not using crs but creating a graph from data, the other going both ways -- from ArcGIS raster object to ndArray and back again -- and seeing how we maintain the crs.

### Create a histogram from a converted ndArray

We could create a histogram in ArcGIS of course, but doing this in Python is of course more automated.

Import numpy as np, and then create a 2D ndarray from the elevation raster object with:

```
elev2D = arcpy.RasterToNumPyArray(elev)
```

Ok, now we have a 2D ndarray. We can make a histogram out of it, but this will require making it a 1D ndarray, which I only discovered by trying to create a histogram from the original 2D ndarray. (You might try this to see what you get.) The conversion to a 1D array uses the `.reshape` numpy method but we don't have to figure out the size, so the size parameter is set to -1 to get the total cells. (You could also use `.flatten`)... then convert it to a 1D ndarray in order to make a histogram out of it.

```
elev1D = np.reshape(elev2D, -1)
```

```
In [ ]: #
```

Now we can make the histogram in matplotlib:

```
import matplotlib.pyplot as plt
plt.hist(elev1D, bins='auto') # this applies .histogram connected to plt, or something like that...
plt.title("Histogram of elev")
plt.show()
```

In [ ]: #

## Converting there and back again, using np map algebra along the way

First we need to get the spatial reference, lower left corner and cell size from elev :

```
sr = elev.spatialReference
lowleft = elev.extent.lowerLeft
cellsize = elev.meanCellHeight
```

In [ ]: #  
#

Out[ ]: 60.0

Then we'll convert to a ndarray

```
elev2D = arcpy.RasterToNumPyArray(elev)
```

In [ ]: #

Then we'll see what we have as a 2D ndarray:

```
import matplotlib.pyplot as plt
fig = plt.figure()
plt.imshow(elev2D, interpolation='none')
plt.colorbar()
plt.title("elev")
plt.show()
```

In [ ]: #

Then we'll do some map algebra, but to the ndarray with numpy:

```
highthreshold = 300
highElev_np = elev2D > highthreshold
```

In [ ]: #

Then display that:

```
fig = plt.figure()
plt.imshow(highElev_np)
plt.colorbar()
plt.title("high elev (elev > {})".format(highthreshold))
plt.show()
```

In [ ]: #

So it looks like this worked fine. The 1s represent True, and 0s False. Interestingly, however, these are stored as True and False in the ndarray, though the legend shows it as numeric.

We could have just done this operation in ArcGIS, but the process of there and back again is the same, so it will serve as an example. To bring this back into ArcGIS, however, this format is not recognized, so I figured out a trick of converting the Trues and Falses to 1s and 0s.

```
highElev_np1 = (highElev_np * 2)/2
```

In [ ]: #

Then we can bring it back in and apply the various environment settings we grabbed earlier:

```
highElev0 = arcpy.NumPyArrayToRaster(highElev_np1, lowleft, cellsize, cellsize)
DM.DefineProjection(highElev0, sr)
```

In [ ]: #

For some reason, we need to make the result discrete, so the following works for that:

```
highElev = Con(highElev0==1,1,0)
highElev.save("highElev_np")
highElev
```

In [ ]: #

Out[ ]:

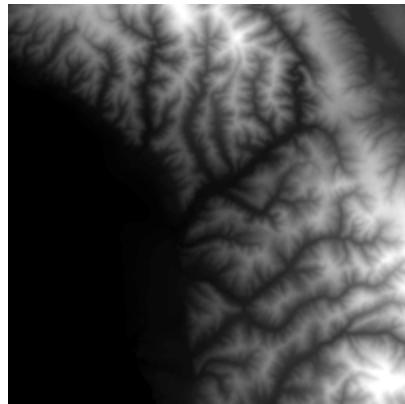


Here's a simpler example that doesn't need unique values, and stands to represent a process involving continuous numerical data:

```
elevft2D = elev2D / 0.3048
elevft2D
elevft = arcpy.NumPyArrayToRaster(elevft2D, lowleft, cellsize, cellsize)
arcpy.DefineProjection_management(elevft, sr)
elevft
```

In [ ]: #

Out[ ]:



## Using Describe & Exists for Geoprocessing

There are many situations in which we want to use characteristics of GIS datasets to process other data. For instance, in raster operations we may want to use the extent of one dataset to delimit another dataset, much like a clip operation, or to detect the type of topology of a feature class. The value of Describe and Exists for coding really makes sense when you realize that it provides your code with the eyes to see information about your data; your code is otherwise blind.

- Start by running the boilerplate we used in the last notebook with module imports and shortcuts. Also initially set the workspace to `hmb.gdb`.

In [ ]: #

## Describe

→ Start by exploring the help system to get a sense of the scope of ArcGIS objects you can get information about. Google ArcGIS Pro Describe should get you to somewhere like this, and you may want to change to the version of ArcGIS Pro we're using (though there won't be many differences in Describe).

<https://pro.arcgis.com/en/pro-app/latest/arcpy/functions/describe.htm> (<https://pro.arcgis.com/en/pro-app/latest/arcpy/functions/describe.htm>)

In contrast to geoprocessing tools, where we can use `help()` from a code cell to get what we need to run the tool, `help(arcpy.Describe)` doesn't provide us with much help because what we need to know about are the objects we want to describe and what their properties are. The link to the help system is thus the best way we have to access this information. It's similar to environment settings. For both objects and the environment there are *many* settings, though I'm usually just looking for a few of them, like the extent (which is part of the environment and associated with objects like feature classes and rasters) and its various parts, like XMin, etc.

There you should be able to find your way to explore properties of various data types that will be useful for us:

- Dataset
- FeatureClass
- File
- Folder
- Layer
- Raster Band
- Raster Dataset
- Table

and lots of others. You should use this resource as we're learning about various Describe properties. We'll start with a problem needing to get the properties of a raster dataset in order to be able to trim it.

In [ ]: #

Out[ ]: ['streams', 'areaclip', 'faults', 'pourpoint', 'publands', 'roads', 'landin  
g', 'geolstr200', 'geol', 'stbuff200']

## Using Describe to get extent

We'll write some code to use Describe to obtain the bounding rectangle of a raster, then use that to clip the edges by a percentage. We're going to need the extent of the raster, but how do we know what's that's called or what it's associated with?

- Go to the same Describe help page we were just looking at, and go to the Raster Dataset properties. (There is no set of properties for anything called simply a "raster", so Raster Dataset seems the closest.).

There we'll see several properties:

- bandCount
- compressionType
- format
- permanent
- sensorType

These all look useful, but aren't what we're looking for. But we see in the heading of the Raster Dataset properties section a message that says that Dataset properties and Raster Band properties are also supported. Datasets include rasters and feature classes, and both have an `extent` (something that is also apparent if you go to the Geoprocessing Environments dialogs in ArcGIS Pro.)

- ▀ We can access the `extent` property for any data set, including rasters, via the `Describe` object. We'll start by assigning that object to a variable:

```
dsc = arcpy.Describe("elev")
```

... then see what it holds with `dsc`

```
In [ ]: #
```

Out[ ]:

<b>catalogPath</b>	C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy\hmb.gdb\elev
<b>dataType</b>	RasterDataset
<b>bandCount</b>	1
<b>format</b>	FGDBR
<b>fields</b>	
<b>spatialReference</b>	
<b>name (Projected Coordinate System)</b>	NAD_1983_UTM_Zone_10N
<b>factoryCode (WKID)</b>	26910
<b>linearUnitName (Linear Unit)</b>	Meter
<b>spatialReference.GCS</b>	
<b>name (Geographic Coordinate System)</b>	GCS_North_American_1983
<b>factoryCode (WKID)</b>	4269
<b>angularUnitName (Angular Unit)</b>	Degree
<b>datumName (Datum)</b>	D_North_American_1983

For additional help, see [arcpy.Describe](#)

- ▀ You should see displayed nothing of clearly immediate use, simply that it's a "geoprocessing describe data object", however we can go one step further by first referring to the Describe help system and seeing that `extent` is one of them, so we'll create another variable `ext` and then display its value with:

```
ext = dsc.extent  
ext
```

In [ ]: #

Out[ ]:

<b>XMin (Left)</b>	545692.537124
<b>XMax (Right)</b>	557692.537124
<b>YMin (Bottom)</b>	4141427.326978
<b>YMax (Top)</b>	4153427.326978
spatialReference	
<b>name (Projected Coordinate System)</b>	NAD_1983_UTM_Zone_10N
<b>factoryCode (WKID)</b>	26910
<b>linearUnitName (Linear Unit)</b>	Meter
spatialReference.GCS	
<b>name (Geographic Coordinate System)</b>	GCS_North_American_1983
<b>factoryCode (WKID)</b>	4269
<b>angularUnitName (Angular Unit)</b>	Degree
<b>datumName (Datum)</b>	D_North_American_1983

... and we get yet another object, the Extent object.

▀ If we then check the help system for Describe and click on what Extent contains, we can see that it includes things like XMin , YMin , XMax , YMax , and quite a few other things. So we should be able to see these as a tuple with:

```
ext.XMin, ext.YMin, ext.XMax, ext.YMax
```

In [ ]: #

Out[ ]: (545692.5371239004, 4141427.3269781955, 557692.5371239004, 4153427.3269781955)

... and if we don't want to use the variables we could get them with:

```
 arcpy.Describe("elev").extent.XMin
```

etc., although that's slower since it has to call Describe for each of the values we're looking for.

In [ ]: #

Out[ ]: (545692.5371239004, 4141427.3269781955, 557692.5371239004, 4153427.3269781955)

There are similar ways of accessing other Describe properties, and this will be very useful for working with data in arcpy. Extent properties just happen to be one that I need the most. In the following code we need to write, we'll use these four extent variables `ext.XMin` etc. to trim a raster.

## Using the extents to trim a raster

We'll use the extent variables `ext.XMin`, `ext.YMin`, `ext.XMax`, and `ext.YMax` to create a smaller area with a third of the width trimmed off each side and the top and bottom. This is admittedly kind of arbitrary, but it demonstrates how we can access these values to manipulate our data, and it's easy to see what it's doing. The method we'll apply is:

1. Create width from `ext.XMax-ext.XMin` and height from `ext.YMax-ext.YMin`
2. Create a trim width `w_trim` as `width/3.0` and a trim height `h_trim` as `height/3.0`
3. Create a rectangle object representing a trimmed areas as a string variable `rect` of four numbers separated by spaces, created using the f string method

```
f"{ext.XMin + w_trim} {ext.YMin + h_trim} {ext.XMax - w_trim} {ext.YMax - h_trim}"
```

▀ We'll do the above first. It should be pretty easy to understand as well as code. End the code cell by displaying `rect` to see if the numbers and format looks right.

```
In [ ]: #
```

```
Out[ ]: '549692.5371239004 4145427.3269781955 553692.5371239004 4149427.3269781955'
```

```
In [ ]: #
```

```
Out[ ]:
```



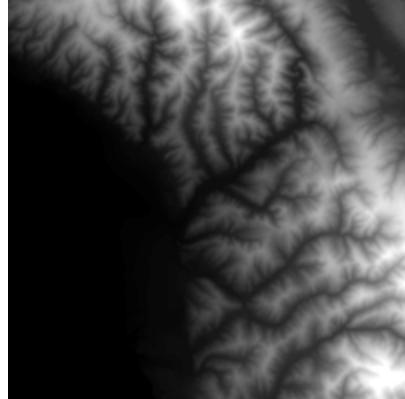
▀ Finally use the rect string to Clip the elev raster to that extent, producing `"trimmed_elev"` and use `Raster()` to create a raster object to display it (no need to save the raster object -- we're done with it so just want to display it. Use `help()` with `DM.Clip` to get the parameters. The first three parameters is all you'll need, but use the explicit method for clarity.

• Interpret what the above is doing, including comparing with what you see with `Raster("elev")`.

:

In [ ]: #

Out[ ]:



Still with the workspace set to hmb.gdb , see what you get with:

```
 arcpy.Describe("streams").DataType  
arcpy.Describe("landing").ShapeType  
arcpy.Describe("landuse").DataType  
arcpy.Describe("geology").DataType
```

In [ ]: #

```
FeatureClass  
Point  
RasterDataset  
RasterDataset
```

Interpret what the above is showing us. Can you see how it might be useful? Remember how a program needs eyes.

## Using Describe with Imagery (multiple bands)

While orthophotography and satellite imagery is in one way just another raster, most imagery ♦ anything beyond black & white ♦ comes in multiple bands. Standard color is RGB, so 3 bands, and some hyperspectral satellite imagery can have hundreds of bands.

▀ We'll look at a Landsat image in the imagery folder.

- Create a new map from the image to see what it looks like. You can change which bands display as red, green or blue using Symbology settings. You should also be able to understand what is meant by "landsatHMB201707.tif/Band\_1" below from exploring the data.
- Use Describe to see how many bands each has and what the cell size is, and fill in the blanks below. The key Describe properties are `.bandCount`, a property of the image itself, and `.meanCellWidth` or `.meanCellHeightSet`, properties of an individual band. For instance, to get the cell size of the Landsat imagery you'll use:

```
arcpy.Describe("imagery/landsatHMB201707.tif/Band_1").meanCellWidth
```

... but to get the `bandCount`, this would be:

```
arcpy.Describe("landsatHMB201707.tif").bandCount
```

⌚ What's the band count and cell width?

In [ ]: #

```
30.0  
7
```

## Other Describe objects

There are many, but one I've used a lot is the **spatial reference**, a dataset property. Probably the most common use is when creating a new dataset where we need to specify the spatial reference, we can "borrow" it from an existing dataset.

▀ We'll borrow one from `elev` by assigning `sr = arcpy.Describe(elev).spatialReference`, then we'll use it with the `out_path` and create a new feature class named "empty" (since we don't have anything to put in it yet). Check out the help system for what `DM.CreateFeatureclass` needs, and we'll provide it with the `out_path` as `env.workspace`, the `out_name` as "empty" and the `spatial_reference` as `sr`.

Then display `sr` to see what spatial reference we assigned, and use `arcpy.ListFeatureClasses()` to confirm it got created.

In [ ]: #

Out[ ]:

<b>name (Projected Coordinate System)</b>	NAD_1983_UTM_Zone_10N
<b>factoryCode (WKID)</b>	26910
<b>linearUnitName (Linear Unit)</b>	Meter
spatialReference.GCS	
<b>name (Geographic Coordinate System)</b>	GCS_North_American_1983
<b>factoryCode (WKID)</b>	4269
<b>angularUnitName (Angular Unit)</b>	Degree
<b>datumName (Datum)</b>	D_North_American_1983

In [ ]: #

Out[ ]: ['streams', 'areaclip', 'faults', 'pourpoint', 'publands', 'roads', 'landing', 'geolstr200', 'geol', 'stbuff200', 'empty']

## Exists

An even more basic piece of information about a dataset is whether it exists or not. Testing for the existence of a dataset can also help us avoid errors, since setting overwriteOutput to True (1) doesn't work for some tools (or at least hasn't always worked in the past). A very useful technique is to detect the existence of a particular dataset using  `arcpy.Exists`, which can be used to detect any type of data.

### Detect if a dataset exists

For instance if you wanted to detect whether an input dataset existed and then delete it if so, you could do something like this. We'll use this to delete the "empty" feature class we just created.

```
if arcpy.Exists("empty"): DM.Delete("empty")
```

In [ ]: #

['streams', 'areaclip', 'faults', 'pourpoint', 'publands', 'roads', 'landing', 'geolstr200', 'geol', 'stbuff200', 'empty']

## Delete before creating a new dataset

It's sometimes useful to delete something before creating a new dataset of the same name. This used to be essential before `env.overwriteOutput = True` became more reliable, but there continue to be situations where you need to do this.

```
if arcpy.Exists("stbuff200"): DM.Delete("stbuff200")
AN.Buffer("streams", "stbuff200", 200)
```

Write code that does the above (in the hmb.gdb workspace), and confirm that it works by running it a couple of times. Include some statements that display the feature classes (`print(arcpy.ListFeatureClasses())`) after deleting and then after creating it anew.

In [ ]: #

```
stbuff200 exists
```

In [ ]: #

```
['streams', 'areaclip', 'faults', 'pourpoint', 'publands', 'roads', 'landin
g', 'geolstr200', 'geol', 'stbuff200', 'empty']
```

In [ ]: #

```
['streams', 'areaclip', 'faults', 'pourpoint', 'publands', 'roads', 'landin
g', 'geolstr200', 'geol', 'empty', 'stbuff200']
```

## Detect if a field exists

You can't use the above method to test whether a field exists, but the following use of `ListFields` does the trick.

- █ Use the following code to add a field as long as it doesn't already exist. Then as before, try it a second time
- ♦ without the test, it would create an error. Note: you may need to close and reopen your project if refresh doesn't work.

```
if not arcpy.ListFields("streams", "stream_class"):
    DM.AddField('streams', "stream_class", 'LONG')
```

- █ Check to see if this worked with

```
for fld in arcpy.ListFields("streams"):
    print(fld.name)
```

We'll use it later in the Data Management and Cursors section to first check whether a field exists before we try to create it.

In [ ]:

```
#  
OBJECTID  
Shape  
FNODE_  
TNODE_  
LPOLY_  
RPOLY_  
LENGTH  
STREAMS_  
STREAMS_ID  
ST_CODE  
Shape_Length  
stream_class
```

The approach we just used was to only create the field if it doesn't exist. What if we wanted to delete it if it does exist? To avoid errors, we'll similarly need to confirm it exists before we delete it.

- Modify the above code to create the `stream_class` field after first deleting it if exists. The basic usage with explicit parameter names is `DM.DeleteField(in_table, drop_field)`. To confirm that it's working, use the for loop twice: after deleting it and then after creating it anew.

In [ ]:

```
#  
OBJECTID  
Shape  
FNODE_  
TNODE_  
LPOLY_  
RPOLY_  
LENGTH  
STREAMS_  
STREAMS_ID  
ST_CODE  
Shape_Length  
OBJECTID  
Shape  
FNODE_  
TNODE_  
LPOLY_  
RPOLY_  
LENGTH  
STREAMS_  
STREAMS_ID  
ST_CODE  
Shape_Length  
stream_class
```

## Create a Smaller Workspace Using Exists and Describe in a Loop

The challenge is to mask (using "city" as the mask) a whole series of rasters and put them into a new workspace. In the process, use a method for checking to see if an output exists before we create a new one. For this script, we'll use `Describe` and `Exists` as part of processing a large set of datasets (however to save time we'll just process a small set).

- We'll start by creating a new geodatabase `HMBcity.gdb` in the project folder, so parallel to our current `hmb.gdb`. For this we can use the path to the project folder stored as the variable `proj` and also define a `newPath` variable which holds the path to the new geodatabase, and delete it first if it already exists before creating a new one.

```
newWS = "HMBcity.gdb"
proj = os.getcwd()
newPath = proj + "/" + newWS
if arcpy.Exists(newPath):
    DM.Delete(newPath)
DM.CreateFileGDB(proj, newWS)
```

Note that in this process, we'll keep `hmb.gdb` as our workspace, and we'll then reference `newPath` for where we want to send the outputs.

In [ ]: #

- Now we'll loop through a list of rasters created with the `ListRasters` method assigning each to `ras`, (hint: `for ras in ...`) and use the `ExtractByMask` tool to clip, storing the clipped city area raster in the `hmbcity` folder.

- Inside the for loop, use an if structure to only process the single band rasters `if arcpy.Describe(ras).bandCount == 1:` and in the if structure:
  - `ExtractByMask` using input `ras` and mask raster "city" and assigning to `newras`
  - Save `newras` using the output name derived as `outputname = newPath + "/" + ras`. *To save time just loop through `arcpy.ListRasters("e")`\**
  - In the loop print the raster name that is being processed so you can see its progress.

In [ ]: #

```
elev
elev30
```

- Wrap up by listing the rasters. (Use a `with` structure.)

```
In [ ]: #
```

## Convert geodatabase feature classes to shapefiles

- Use feature classes from pen.gdb to create a smaller workspace called penshapes (parallel to pen.gdb, so as a folder workspace inside pen), including all of the feature classes in pen.gdb converted to shapefiles in penshapes:

```
cities geol landuse water faultcov_arc
```

Hints:

- You can borrow a lot of the logic used in the previous script to create the parallel workspace. It's simpler because there aren't bands.
- You'll want to have: `from arcpy import conversion as CV`
- Since you'll be using it 3 times and it has to be the same, set a string variable to hold the folder name: `shapefolder = "penshapes"` and you can build its path as `shapesPath = proj + "/" + shapefolder`, and start by deleting it if it already exists.

```
In [ ]: #
```

```
In [ ]: #
```

```
C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy/penshapes
```

- Continuing...

- Use `CreateFolder` to create the penshapes folder. (We won't use a geodatabase because our goal is to create shapefiles.)

Look up the usage for `CreateFolder` -- it needs both a folder to put the folder in and the name of the folder you want to create.

```
In [ ]: #
```

## Continuing...

- In your loop through the feature classes, you'll need a tool from the conversion toolbox: `FeatureClassToShapefile` and the output folder would be the `shapesPath` created above.
- If you store a feature class into a folder, it will be created as a shapefile.

Note that since the output is specified as the folder name, you don't need to specify the shapefile name as the output, and it will simply be named the same as the original.

- As before, check to see what you get by setting the workspace to `shapesPath` and printing the list of feature classes (shapefiles are also feature classes).

In [ ]:

Out[ ]: 'C:\\\\Users\\\\900008452\\\\Box\\\\course\\\\625\\\\exer\\\\Ex07\_08\_arcpy\\\\hmb.gdb'

In [ ]: #

```
[ 'areaclip.shp', 'empty.shp', 'faults.shp', 'geol.shp', 'geolstr200.shp', 'landing.shp', 'pourpoint.shp', 'publands.shp', 'roads.shp', 'stbuff200.shp', 'streams.shp' ]
```

## Feature to raster

To create rasters from feature classes, you need to specify a field to get the Value to assign to the raster.

To do this, create a two Python lists, to hold the dataset pairs `geol`, `TYPE_ID`, `landusePen`, `LU_CODE` in `pen.gdb`.

Then convert each feature class in `pen.gdb` to a raster (to store back in the same gdb), with the corresponding field, and use "60" as the cell size. \*Hint: to connect the indices for the feature class and the field, you'll want to loop through the indices like `for i in range(len(feat))`

```
feat = ["geol", "landusePen"]
fld = ["TYPE-ID", "LU-CODE"]
```

In [ ]: #

# Debugging and Messaging

Debugging in Jupyter is not quite the same as you may find in an IDE like Spyder, and for writing script tools we'll probably want to explore those methods. However, the cell structure of Jupyter does provide some decent debugging options.

## Splitting cells

A useful way of debugging in Jupyter is to break up your code into multiple code cells. This is also a good idea for providing more documentation to your code so you will remember what it's doing and for anyone who reads your code to understand it. You can either write your code this way from the beginning or split it by inserting your cursor where you want it split and using `Edit/Split Cell`.

- Copy one of your fully working code cells here, and then split it into code cells at likely locations.

In [ ]:

## Print statements

The tried and true method used by programmers since the dawn of time: simply printing the current value of variables, or printing information about the progress of the script. Simply add a print statement to tell the user where the program has gotten to, and once you're done debugging, you can leave it as a comment statement so you'll remember later on what you're doing. Some good places to put print statements include:

- at the end of the code cell, where you've arrived at a product of that cell. This is a lot like a function, when you have one final result.
- during each step of a loop, where a print statement can show progress. If an operation takes a lot of time, this can show you that it's still running (and isn't hung up) and give you a sense of how much time it'll take to complete (you could probably also include some code to derive an estimated time remaining to complete the task).

You could keep print statements in your code but just turn them off with a #, as shown here:

```
AN.Buffer("streams", "stbuff200", 200)
#print("Finished Buffer. Now clipping...")
AN.Clip("geol", "stbuff200", "geolstr200")
```

- For one of the programs you've already run, add print statements to tell the user what the program is doing, especially during a loop.

In [ ]:

## try...except blocks

One way of handling errors is to provide some code to run in case an "exception" (error) is raised. This allows you to put things back in order and also display any messages that the offending code produced. A try...except block can go anywhere in your code, and code within the try block is run until an exception is raised where it shifts execution to the except block, which only runs if an exception is raised.

For your feature to raster code created earlier, copy it into the next code cell. Then insert a `try:` statement shortly before the code where you're creating the feature and field codes, and then at the end of the section to evaluate (where the exception might be raised), insert an `except:` statement. Indent all of the code in between, as well as the code to run if the exception is raised. Make sure to create an exception with a typo -- a misspelling of the field name perhaps.

Heres what that part of the code might look like, where you note that `LU-CODE` is typed instead of `LU_CODE`, the correct field name in the attribute table:

```
try:  
    feat = ["geol", "landusePen"]  
    fld = ["TYPE_ID", "LU-CODE"]  
    env.workspace = projdir + "\pen.gdb"  
    for i in range(len(feat)):  
        CV.FeatureToRaster(feat[i], fld[i], feat[i] + "ras", 60)  
    except:  
        print(arcpy.GetMessages())
```

Note that what's in the exception section is printing `arcpy.GetMessages()` which is what is produced from the most recently executed geoprocessing function. You'll find that the message displayed is very clear about what the problem is.

In [ ]: #

→ Explore the `GetMessages` section of the help system. You'll find for instance that different types of messages can be selected to display, like warnings, general messages and errors; by default, all messages are displayed. You may also see `.AddMessage` which doesn't really do anything in Jupyter, but is useful when running script tools and provides information to the user when they run the script tool (which `print()` does not).

# Data management

Processing data in tables, as well as creating fields to store those data, and other data operations are important in GIS work. Using a script is a good choice when you need to perform a sequence of data management and analysis steps involving data tables. In some cases we need to process data fields, and there is an array of tools we can use to, for example, add fields, calculate values for fields, delete fields, and join tables to bring in additional data fields via a relate field. Some of these tools also create new summary tables, where input data fields are summarized using various statistics. In other cases, we need to work with rows of data, which might be individual features with vector data or values for raster data; we'll learn about using cursors to process rows, one at a time.

## Data Management (including related Analysis) Tools

First, we'll look at some of the tools we can use to process data tables, primarily involving fields. A few tools also select records (by attributes), or copy or delete selected records.

Toolbox	Tool	What it does	Output
Analysis	Frequency	Calculates frequency statistics for field(s) in the input table	table
	Statistics	Calculates summary statistics for field(s) in the input table.	table
	Select	Uses a where clause to selects features from an input to create an output	new feature class
	TableSelect	Extracts selected attributes from an input table, using an SQL Where clause	table
Data Management	AddField	Adds a field to an data table	field in existing table
	CalculateField	Calculates a value for a field using an expression	values in an existing field
	DeleteField	Removes a field from a table	
	AddXY	Adds an x & y (and maybe z) fields to a point feature class	x, y, etc. fields with values
	CopyFeatures	Copies selected features to a new feature class	new feature class
	DeleteFeatures	Deletes selected features	
	AddJoin	Joins a table to a layer (or a table to a table) based on a common field	join relationship
	RemoveJoin	Removes an existing join	
	SelectLayerByAttribute	Creates, updates, or removes a selection on the layer or table view using an attribute query	
	CreateFeatureclass	Create a new feature class (including shapefiles)	feature class

## Create and calculate a new field

Using the `marbles.gdb` geodatabase, create code that adds the field "ContourFeet" to a 10m elevation contour created from the elevation raster (you'll need to create the contours first), and calculates its values as `!elev! / 0.3048`. (Fields are denoted with exclamation point brackets in expressions.) Here are some hints:

- Start with your boilerplate including working with Spatial Analyst and the DM shortcut to the data management toolbox. As before, save the script in the project folder and use the relative path method.
- To avoid creating a problem due to inconsistent names, before running the statements that need them, assign names of fields and datasets to string variables. This is good practice (assign hard-coded values only once in a script) and also prepares you for getting these as inputs in a script tool later on. Here are some you'll want to use:
  - `contourfcl = "contours10m"` (for the contour feature class)
  - `elevras = Raster("elev")` (for the elevation raster)
  - `elevfeet = "ContourFeet"` (for the field name)
- Use `Exists` and `Delete` to delete `contourfcl` if it already exists.
- Create the contour feature class using the Contour tool in Spatial Analyst. Note that since this creates a feature class, not a raster object, so doesn't work in map algebra. However, since you've imported everything (\*) from `arcpy.sa`, you can run the tool with some shorthand as: `Contour(elevras, contourfcl, 10)` to create 10m contours
- To display the resulting field names with one line of code, use:

```
for fld in arcpy.ListFields(contourfcl): print(fld.name)
```

- Add the `elevfeet` field as type "DOUBLE" after first making sure it doesn't already exist, with `if not arcpy.ListFields(contourfcl, elevfeet):`
- Use `CalculateField` from the management toolbox to assign "`!Contour! / 0.3048`" to the `elevfeet` field.
- Finally check out the results in ArcGIS.

In [ ]: #

In [ ]: #

Note the use of `!Contour!` to represent the `Contour` field. This is required in this type of expression.

## AddXY

Create code that uses the AddXY tool in the management toolbox (features toolset) to add x & y values to "samples" in the marbles workspace.

In [ ]: #

## Using codeblocks to loop through data when calculating values

In the next section, we'll be looking at cursors, which has even more capabilities for dealing with individual records, but `CalculateFields` can also use a function to operate differently depending on a given value, by using a `codeblock` function. If you run the `Calculate Field` geoprocessing tool using the dialog, you'll see the `Code Block` textbox. Here's an example. We have whale count data in a CSV from 2019, and we'd like to adjust whale counts from two observers:

- Allison has poor distance vision but we've found from independent observers that there are actually twice as many whales visible as she reports.
- Sydney exaggerates counts, and from independent observers we've found that it's best to reduce his counts by half.

First we'll give you a taste of how we might use cursors to print an attribute table as a pandas dataframe:

```
In [ ]: import arcpy, os
proj = os.getcwd()
import pandas as pd
arcpy.env.workspace = proj
def table(tbl):
    fld_list = arcpy.ListFields(tbl)
    flds = []
    for fld in fld_list: flds.append(fld.name)
    table = []
    cur = arcpy.SearchCursor(tbl, flds)
    for row in cur:
        rowList = []
        for fld in flds:
            rowList.append(row.getValue(fld))
        table.append(rowList)
    del cur
    df = pd.DataFrame(table, columns = flds)
    return df
```

So we'd like our code to multiply Allison's counts by 2, Sydney's by 0.5, and leave the rest the same -- so multiply by 1. Our function is able to detect these conditions and come up with the adjusted count, stored in a new field in the attribute table. See if you can correctly interpret it.

```
whalesWS = "whalesGGate"
if not arcpy.Exists(whalesWS):
    arcpy.CreateFolder_management(proj,whalesWS)
else: print(whalesWS + " exists.")

codeblock = """def multiplier(observer):
    if observer == "Allison":
        return 2
    elif observer == "Sydney":
        return 0.5
    else: return 1"""

whaleData = proj+"\ex07_08_arcpy_data\Humpback_2019.csv"
env.workspace = proj+"\\"+whalesWS
if arcpy.Exists("whales.shp"): DM.Delete("whales.shp")
DM.XYTableToPoint(whaleData, "whales.shp", x_field="Longitude", y_field="Latitude")
expression = "!n_whales! * multiplier(!Observer!)"
adjustedCountFld = "adjCount"
if not arcpy.ListFields("whales.shp",adjustedCountFld):
    DM.AddField("whales.shp",adjustedCountFld,"DOUBLE")
arcpy.CalculateField_management ("whales.shp", adjustedCountFld, expression, "PYTHON",
N", codeblock)
table("whales.dbf")
```

Learn more about this at: <https://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/00170000004m000000>  
[\(https://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/00170000004m000000\)](https://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/00170000004m000000)

In [ ]:

#

whalesGGate exists.

Out[ ]:

	FID	Shape	Observer	Year	Month	Day	Hour	Minute	Latitude	Longitude
0	0	(-122.493592 37.794274 NaN NaN)	Allison	2019	4	2	12	45	37.794274	-122.493592
1	1	(-122.459953 37.830697 NaN NaN)	Allison	2019	5	1	11	30	37.830697	-122.459953
2	2	(-122.599558 37.81561 NaN NaN)	Allison	2019	5	4	12	30	37.815610	-122.599558
3	3	(-122.604379 37.797698 NaN NaN)	Allison	2019	5	4	15	30	37.797698	-122.604379
4	4	(-122.527239 37.7292360000001 NaN NaN)	Allison	2019	5	14	11	30	37.729236	-122.527239
...	...	...	...	...	...	...	...	...	...	...
172	172	(-122.515109 37.8055350000001 NaN NaN)	Allison	2019	10	13	8	30	37.805535	-122.515109
173	173	(-122.809715 37.6611310000001 NaN NaN)	Allison	2019	10	13	10	0	37.661131	-122.809715
174	174	(-122.947331 37.730912 NaN NaN)	Allison	2019	10	13	2	0	37.730912	-122.947331
175	175	(-122.620435 37.782788 NaN NaN)	Allison	2019	11	9	12	30	37.782788	-122.620435
176	176	(-122.569333 37.768944 NaN NaN)	Allison	2019	11	9	1	0	37.768944	-122.569333

177 rows × 13 columns



## Select

Selecting by attributes is a common need.

▀ Use `help(AN.Select)` to learn about its parameters.

▀ We'll want to set the workspace to "pen.gdb", and use  `arcpy.ListFeatureClasses()` to see what it contains.

```
In [ ]: #
```

```
Out[ ]: ['cities', 'faultcov_arc', 'geol', 'water', 'landusePen', 'urban']
```

- For the land use feature class in pen.gdb, find out what the fields are. You'll want to find the land-use code.

```
In [ ]: #
```

```
OBJECTID  
Shape  
AREA  
PERIMETER  
LANDUSE_  
LANDUSE_ID  
LU_CODE  
Shape_Length  
Shape_Area
```

- Use the `Select` tool from the analysis toolbox with the input features being the land use feature class, create "urban" as the output feature class, and use a where clause that gets all land-codes under 20.

```
In [ ]: #
```

- Check the map to confirm that you got what you were hoping for.

## Layers

Often you'll want or need to work with layers similar to how you often work in a map. Instead of creating a new feature class with a selection, you might apply a selection query to a layer.

- Let's do the same thing we just did but create the output as a layer named "Urban" by using the `MakeFeatureLayer` tool in the data management toolbox. Use the same input feature class and where clause.

```
In [ ]: #
```

- Then check out the results on the map. In the contents, go to the properties of both this new layer "Urban" and the previous feature class "urban" to check their source.

- What do you find?

..

# Cursors

Cursors give you access to values in your data fields, and allow you to loop through records in your data tables. Since each record (row) might be a vector feature or a raster value, this gives you considerable power to process your data, manipulate and create geometries, and develop tools that can do what isn't possible in the stock ArcGIS GUI.

There are three types of cursors:

- SearchCursor : read values in a row
- InsertCursor : insert new rows
- UpdateCursor : to change values in rows and delete rows

▀ First we'll use a simple cursor that goes through the X & Y values of a feature class by *getting the first part* of the value in the shape field, after determining that field's name. Note that all geometries have parts, which make sense for polylines and polygons, but even points have parts, even if there's only one part, as there is here. Note that you need to delete the cursor at the end.

It's useful to remember that each feature is a row in the database. The following code loops through the database as a "cursor" scrolling through the rows. The `for ptf in cur:` assigns each row feature to an object we'll call `ptf` (for "point feature"), then `getValue(shapefld)` gets the shape field value from that feature. The `.getPart()` gets the only part of the point geometry, which is then assigned to `pnt`, from which we can get X and Y values with `pnt.X` and `pnt.Y`.

```
import arcpy, os
projdir = os.getcwd()
from arcpy import env
env.workspace = "marbles.gdb"
desc = arcpy.Describe("co2july95")
shapefld = desc.ShapeFieldName
cur = arcpy.SearchCursor("co2july95")
for ptf in cur:
    pnt = ptf.getValue(shapefld).getPart()
    print(f"{pnt.X}, {pnt.Y}")
del cur
```

In [ ]:

```
#  
484700.700000002, 4601827.0  
484164.299999998, 4601425.0  
484164.299999998, 4601425.0  
484164.299999998, 4601425.0  
484903.299999998, 4599948.0  
485286.0, 4600092.0  
485126.0, 4600703.0  
485126.0, 4600703.0  
483986.0, 4600852.0  
483454.0, 4601142.0  
483454.0, 4601142.0  
483479.0, 4601325.0  
483614.0, 4601283.0  
482971.700000002, 4602427.0  
482816.700000002, 4602599.0  
483486.0, 4601072.0  
482983.299999998, 4602562.0  
483349.0, 4602502.0  
483534.0, 4601546.0  
483509.400000004, 4601475.0  
485077.0, 4599968.0  
483662.299999998, 4601293.0
```

⌚ What shape field was detected?

∴

⌨ Print out a list of all of the field names, so we can build the next code:

In [ ]:

```
#  
OBJECTID  
Shape  
DATE_  
PDT  
CO2_  
SOIL°C  
AIR°C  
D_CM  
ID  
LOC  
ELEV  
PM  
DESCRIPTION  
X  
Y
```

⌨ Modify the above code that display pnt.X and pnt.Y to also display the date, elevation, PM (parent material, such as rock type), and CO2 value, using the names we found for those fields.

In [ ]:

```
#  
7/14/95, 4820, till?, 1.0800000429153442, 484700.7000000002, 4601827.0  
7/14/95, 5320, schist, 0.5600000023841858, 484164.2999999998, 4601425.0  
7/14/95, 5320, schist, 1.0499999523162842, 484164.2999999998, 4601425.0  
7/14/95, 5320, schist, 0.47999998927116394, 484164.2999999998, 4601425.0  
7/15/95, 5840, neoglacial till, 0.620000047683716, 484903.2999999998, 459994  
8.0  
7/15/95, 5800, neoglacial till, 0.8199999928474426, 485286.0, 4600092.0  
7/15/95, 5600, mmv, 0.8299999833106995, 485126.0, 4600703.0  
7/15/95, 5600, mmv, 0.5199999809265137, 485126.0, 4600703.0  
7/15/95, 5520, marble, 0.800000011920929, 483986.0, 4600852.0  
7/15/95, 5800, marble, 0.5099999904632568, 483454.0, 4601142.0  
7/15/95, 5800, marble, 0.5299999713897705, 483454.0, 4601142.0  
7/15/95, 5750, schist, 1.0800000429153442, 483479.0, 4601325.0  
7/15/95, 5660, marble, 0.6399999856948853, 483614.0, 4601283.0  
7/15/95, 6800, marble, 0.23000000417232513, 482971.7000000002, 4602427.0  
7/15/95, 7000, marble, 0.18000000715255737, 482816.7000000002, 4602599.0  
7/15/95, 5740, marble, 0.5600000023841858, 483486.0, 4601072.0  
7/15/95, 6940, neoglacial till, 0.3100000023841858, 482983.2999999998, 460256  
2.0  
7/15/95, 6600, marble, 0.8100000023841858, 483349.0, 4602502.0  
7/15/95, 5820, schist, 0.4300000071525574, 483534.0, 4601546.0  
7/15/95, 5820, schist, 0.6800000071525574, 483509.4000000004, 4601475.0  
7/15/95, 5880, neoglacial till, 0.6800000071525574, 485077.0, 4599968.0  
7/15/95, 5660, marble, 0.6000000238418579, 483662.2999999998, 4601293.0
```

Note in the above output for CO2 and X and Y values have excessive decimal places (resulting from the way digital numbers are stored, which at one level deep in the decimal places is an approximation).

- Using either the `round()` function or (better) the f-strong formatting method, round these to 2 decimal places for CO2 (how they were originally recorded) and 0 decimal places for X & Y (the approximate accuracy of our location method in the field).

In [ ]:

```
#  
7/14/95, 4820, till?, 1.08, 484701, 4601827  
7/14/95, 5320, schist, 0.56, 484164, 4601425  
7/14/95, 5320, schist, 1.05, 484164, 4601425  
7/14/95, 5320, schist, 0.48, 484164, 4601425  
7/15/95, 5840, neoglacial till, 0.62, 484903, 4599948  
7/15/95, 5800, neoglacial till, 0.82, 485286, 4600092  
7/15/95, 5600, mmv, 0.83, 485126, 4600703  
7/15/95, 5600, mmv, 0.52, 485126, 4600703  
7/15/95, 5520, marble, 0.80, 483986, 4600852  
7/15/95, 5800, marble, 0.51, 483454, 4601142  
7/15/95, 5800, marble, 0.53, 483454, 4601142  
7/15/95, 5750, schist, 1.08, 483479, 4601325  
7/15/95, 5660, marble, 0.64, 483614, 4601283  
7/15/95, 6800, marble, 0.23, 482972, 4602427  
7/15/95, 7000, marble, 0.18, 482817, 4602599  
7/15/95, 5740, marble, 0.56, 483486, 4601072  
7/15/95, 6940, neoglacial till, 0.31, 482983, 4602562  
7/15/95, 6600, marble, 0.81, 483349, 4602502  
7/15/95, 5820, schist, 0.43, 483534, 4601546  
7/15/95, 5820, schist, 0.68, 483509, 4601475  
7/15/95, 5880, neoglacial till, 0.68, 485077, 4599968  
7/15/95, 5660, marble, 0.60, 483662, 4601293
```

## SearchCursor with the Data Access module

While the above cursor method worked perfectly well for what we were asking it, the Data Access module ( `arcpy.da` ) provides a lot of advantages. When used with cursors, it will provide an easier way of referencing coordinates. Making use of it simply requires using  `arcpy.da`  instead of  `arcpy`  when calling the  `SearchCursor` .

Try the following code which demonstrates the use of the  `SearchCursor`  to simply display the values of two fields from a shapefile. Note the use of field names as properties of the row.

```
import arcpy, os  
projdir = os.getcwd()  
from arcpy import env  
ws = env.workspace = projdir + "/curdata"  
cur = arcpy.da.SearchCursor("contour.shp", ("ID", "ELEV"))  
for row in cur:  
    print(f"{row[0]}, {row[1]}")
```

In [ ]:

```
#  
('ID', 'ELEV')  
1, 100  
2, 110  
3, 120  
4, 130  
5, 140  
6, 90
```

Change the code to display all of the fields. We'll detect the number of fields, and with a bit of extra coding manage to create a comma-delimited string from it that contains the row of data.

```
flds = arcpy.ListFields(contours)  
for i in range(len(flds)):  
    flds[i] = flds[i].name  
print(flds)  
cur = arcpy.da.SearchCursor(contours, flds)  
for row in cur:  
    datastring = ""  
    for i in range(len(flds)-1): datastring = datastring + str(row[i]) + ","  
    print(datastring + str(row[len(flds)-1]))
```

In [ ]:

```
#  
['FID', 'Shape', 'ID', 'ELEV']  
0,(122.18374783759187, 247.1517621378525),1,100  
1,(242.11510684480945, 204.621200940252),2,110  
2,(280.20973034864556, 193.513895989157),3,120  
3,(333.11804451342624, 241.15101055645692),4,130  
4,(324.04447355095357, 254.64798628809612),5,140  
5,(123.96583398725548, 293.28553785297885),6,90
```

## Using a SearchCursor to write out point features

*Geometries* are going to be one of the things we'll work with a lot with cursors. We'll briefly explore them, but there's a lot more to them, so learn more about how these are structured in <https://pro.arcgis.com/en/pro-app/latest/arcpy/classes/geometry.htmexplore> (<https://pro.arcgis.com/en/pro-app/latest/arcpy/classes/geometry.htmexplore>). We'll be using the Data Access module to facilitate geometry use.

■ Use a SearchCursor to read and print out values for all of the point features, including its id, Calcium Carbonate concentrations and XY coordinates from samples.shp from the Marble Mountains, in `curdata`.

- Start with the typical boilerplate, including the workspace setting to `curdata` we just used.
- Set a variable `ptfeats` to have the path to `samples.shp`
- Create a list of field names composed of `"sample_id"`, `"CATOT"`, and `"SHAPE@XY"` and assign it to a variable `fields`
- Use a `with` structure to enclose your cursor loop, which limits the cursor to the structure: `with arcpy.da.SearchCursor(ptfeats, fields) as cur:`
- Within that `with` structure, use a `for` loop structure to go through all of the point features in the list and print out the three fields (including shape) with

```
for ptf in cur:  
    print(f"{ptf[0]}, {ptf[1]}, {ptf[2]}")
```

- After the loop, delete the cursor with `del cur` though that's not really necessary...

In [ ]: #

48, 0.8, (485701.96875, 4602950.0)  
49, 0.83, (485956.28125, 4602944.5)  
47, 0.83, (485589.78125, 4602934.0)  
4, 0.63, (485857.5, 4602919.0)  
46, 0.67, (485403.4375, 4602791.5)  
1, 0.7, (485711.375, 4602716.0)  
45, 0.77, (485291.28125, 4602685.0)  
44, 0.83, (485141.84375, 4602566.5)  
43, 0.73, (485023.1875, 4602377.5)  
42, 0.8, (484917.71875, 4602118.5)  
41, 0.77, (484847.40625, 4601885.5)  
33, 0.92, (483146.5625, 4601819.5)  
18, 0.56, (484807.8125, 4601771.0)  
24, 0.56, (484785.84375, 4601753.5)  
32, 0.77, (483366.28125, 4601709.5)  
31, 0.8, (483678.3125, 4601657.0)  
30, 0.67, (483933.25, 4601643.5)  
29, 0.61, (484051.90625, 4601599.5)  
55, 0.77, (485145.34375, 4601557.0)  
54, 0.88, (485014.6875, 4601545.5)  
28, 0.36, (484161.78125, 4601538.0)  
25, 0.58, (484627.625, 4601534.0)  
53, 0.88, (484866.9375, 4601486.0)  
23, 1.02, (484804.4375, 4601457.5)  
27, 0.63, (484298.0, 4601441.5)  
67, 0.09, (482853.5, 4601431.0)  
65, 0.38, (482839.375, 4601398.0)  
68, 0.17, (482815.8125, 4601384.0)  
69, 0.04, (482835.84375, 4601383.5)  
66, 0.4, (482858.5625, 4601378.0)  
26, 0.61, (484324.375, 4601353.5)  
22, 0.58, (484258.46875, 4601287.5)  
17, 0.83, (484959.4375, 4601175.5)  
40, 0.73, (485075.9375, 4601175.5)  
21, 0.8, (483902.4375, 4601160.0)  
16, 0.8, (484818.8125, 4601085.5)  
39, 0.19, (485262.71875, 4601072.5)  
15, 0.4, (484776.0, 4601020.0)  
64, 0.7, (482739.25, 4600977.5)  
14, 0.38, (484759.46875, 4600958.0)  
63, 0.83, (482852.875, 4600915.0)  
38, 0.17, (485260.5, 4600903.0)  
5, 0.73, (484772.65625, 4600892.0)  
62, 0.8, (482943.78125, 4600824.0)  
61, 0.58, (483003.4375, 4600812.5)  
56, 0.48, (483756.21875, 4600733.0)  
58, 0.67, (483699.40625, 4600716.0)  
57, 0.63, (483750.53125, 4600710.5)  
59, 0.8, (483412.5, 4600705.0)  
13, 0.38, (484884.71875, 4600704.0)  
11, 0.92, (483136.96875, 4600702.0)  
10, 0.97, (483193.78125, 4600648.0)  
60, 0.88, (483216.5, 4600636.5)  
9, 0.7, (484922.0625, 4600555.0)  
37, 0.05, (485317.625, 4600545.0)  
12, 0.44, (484904.5, 4600479.0)  
20, 0.92, (483506.90625, 4600437.0)

```
36, 0.05, (485408.84375, 4600386.5)
35, 0.07, (485482.4375, 4600217.5)
8, 0.05, (484966.03125, 4600185.5)
6, 0.02, (485371.46875, 4599995.5)
7, 0.02, (484911.09375, 4599726.5)
```

Q What does `ptf[2]` represent and how is it structured? :

## Write points out to a text file

■ Now modify the code to write this out to a text file -- save the script as a new name.

- Add some code just after assigning the `ws` variable to create a new textfile (open it for writing), and first delete it if it already exists:

```
txtpath = ws + "/samples.csv"
if arcpy.Exists(txtpath): arcpy.Delete_management(txtpath)
txtout = open(txtpath, "w")
```

- Change the last item in the fields list from "SHAPE@XY" to "SHAPE@X" , "SHAPE@Y" . Why? We're going to write the data to a text file, and to read it in again, it's going to be easier to parse out x and y as separate items. What we're seeing is a lot of flexibility with the geometry.
- Before the loop, write out the field names to the text file, to make it more useful:

```
for fld in fields[:-1]:
    txtout.write(f"{fld}, ")
txtout.write(f"{fields[-1]}\n")
```

- Within the loop, instead of printing to the screen, use the following to write out all of the data to the text file. Note the addition of "\n" since (unlike print) the write method doesn't otherwise insert a new line:

```
for fi in range(len(fields)-1): txtout.write(f"{ptf[fi]}, ")
txtout.write(f"{ptf[-1]}\n")
```

- Then at the end, out of the for loop (so unindent), close the text file.

	A	B	C	D
1	sample_id	CATOT	SHAPE@X	SHAPE@Y
2	48	0.8	485702	4602950
3	49	0.83	485956.3	4602945
4	47	0.83	485589.8	4602934
5	4	0.63	485857.5	4602919
6	46	0.67	485403.4	4602792
7	1	0.7	485711.4	4602716
8	45	0.77	485291.3	4602685

In [ ]: #

Later we may modify this script to run as a script tool with the feature layer as an input and writing out a text file.

## InsertCursor

We'll write an InsertCursor to populate a feature class we'll create.

▀ Use an InsertCursor to insert some points that we'll hard-code into the script, after first creating the feature class. We'll create a shapefile, but similar code could be used to create a feature class in a geodatabase. We'll also create a data folder to put the new shapefile in.

1. Set things up.

- Use the same boilerplate as the last two scripts to import everything you need and set the workspace.
- Set up the shortcut to data management tools as DM (from arcpy import management as DM)
- Assign "marblePts.shp" to ptfeatname .
- *Because of a bug, insert print(arcpy.ListFeatureClasses()) to "wake it up" TEST*

```
In [ ]: #  
['co2july95.shp', 'contour.shp', 'geology.shp', 'marblePts.shp', 'mvalley_pt  
s.shp', 'randomPolygons.shp', 'randomPolylines.shp', 'samplePts.shp', 'sample  
s.shp', 'streams.shp', 'trails.shp', 'veg.shp', 'water.shp']
```

⌚ Do you see the shapefile that you're going to create?

Step 2. Create the shapefile, borrowing an existing spatial reference, and add the fields we'll need.

- If that shapefile (use the variable ptfeatname ) already exists in the workspace, DM.Delete it .
- Use Describe to get the spatialReference from existing samples.shp data, and assign it to sr . Hint on coding this: assign arcpy.Describe("samples.shp").spatialReference to sr .
- Create a shapefile using the CreateFeatureclass tool from DM , with the following parameters: (ws, ptfeatname, "POINT", "", "", "", sr)
- For testing, see if it exists with print(arcpy.ListFeatureClasses())
- Use the AddField tool to add the fields id of type LONG and then name of type TEXT to ptfeatname .

```
In [ ]: #  
['co2july95.shp', 'contour.shp', 'geology.shp', 'marblePts.shp', 'mvalley_pt  
s.shp', 'randomPolygons.shp', 'randomPolylines.shp', 'samplePts.shp', 'sample  
s.shp', 'streams.shp', 'trails.shp', 'veg.shp', 'water.shp']  
FID  
Shape  
Id  
name
```

⌚ Did it get created, with the desired fields?

### Step 3.

- Create a list `ptdata` of points defined as tuples {hint: `[(...),(...)]`}, with each point definition as an id, name, and x&y coordinates in UTM, using the following set of values (note that in the shapefile, a numeric Id field is always created by default, so we'll just populate it):

```
(12, "Upper Meadow", (483473,4601523))  
(42, "Sky High Camp", (485339,4600001))
```

```
In [ ]: #
```

### Step 4. Cursor processing

- Create an insert cursor named `cur` using your newly created `ptfeatures` feature class.

```
cur = arcpy.da.InsertCursor(ptfeatname, ("id", "name", "SHAPE@XY"))
```

- Loop through your `ptdata` list of points, assigning each as `pt` and inside the list:

```
cur.insertRow(pt)
```

```
In [ ]: #
```

### Step 5. Wrap things up.

- Outside of the loop, delete the cursor and all objects

```
del cur
```

- As a finishing touch, add xy coordinates with the `AddXY` tool:

```
DM.AddXY(ptfeatname)
```

```
In [ ]: #
```

## Mapping XY points

If you are running this in ArcGIS Pro and had a map open, you will have been able to see the points displayed on the map.

### Geopandas alternative

But if (and only if) you're running this from Jupyter Notebooks, you'll want to use Geopandas to see the result:

```
import geopandas as gpd
import matplotlib.pyplot as plt
marblepts = gpd.read_file('curdata/' + ptfeatname)
marblepts
```

Then to plot the two points just requires:

```
marblepts.plot()
```

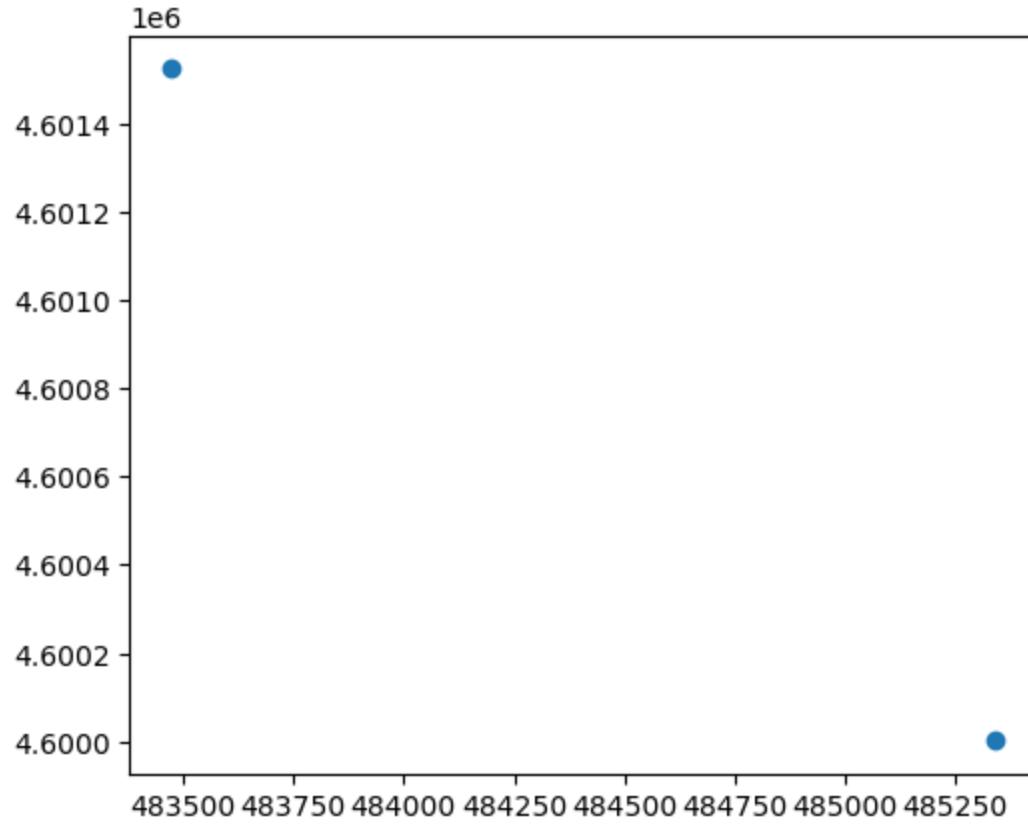
In [ ]: #

Out[ ]:

	<b>Id</b>	<b>name</b>	<b>POINT_X</b>	<b>POINT_Y</b>	<b>geometry</b>
0	12	Upper Meadow	483473.0	4601523.0	POINT (483473.000 4601523.000)
1	42	Sky High Camp	485339.0	4600001.0	POINT (485339.000 4600001.000)

In [ ]:

Out[ ]: &lt;AxesSubplot: &gt;



## Same as above, but in a file geodatabase.

The above code is designed to work with shapefiles in the curdata folder. We're using shapefiles in most of these cursor scripts to make it easy to work with text files stored in the same folder, but we should also briefly look at how a cursor-based script would work with a geodatabase.

- The above script doesn't work with text files, and is pretty easy to change to working with a geodatabase: simply change the workspace to "marbles.gdb" and change "marblePts.shp" and "samples.shp" to "marblePts" and "samples".
- However, we won't be able to look at it with geopandas, since that can only work with OpenGIS data like shapefiles.

In [ ]: #

```
['co2july95', 'geology', 'samples', 'streams', 'trails', 'veg', 'water', 'watershed', 'contours10m', 'marblePts', 'cont']  
[['co2july95', 'geology', 'samples', 'streams', 'trails', 'veg', 'water', 'watershed', 'contours10m', 'cont', 'marblePts']]  
OBJECTID  
Shape  
id  
name
```

## Read a text file into a feature class

In an earlier script, we used a search cursor to read in a shapefile and write out a text CSV file of data. What if we wanted to do the reverse? We could do this by combining a method for reading text files (see section 1) with what we just wrote to create a feature class and insert point features. Note that we should end up with just a copy of the feature class, but in the process we'll see how we can create a feature class. Feel free to borrow parts of code, but you do need to be careful to set it up to work the way it needs to.

- We'll work with the samples.csv file we wrote out earlier, and create a samplepts.shp output in the data folder we just used for shapefiles. After you do this, you might want to try modifying it to work with the file geodatabase.
- Start with the code from Mb3\_InsertCreatePts.py, but change the output feature class to "samplePts" and make other changes after that ↴

In [ ]: #

```
['co2july95.shp', 'contour.shp', 'geology.shp', 'marblePts.shp', 'mvalley_pts.shp', 'randomPolygons.shp', 'randomPolylines.shp', 'samplePts.shp', 'sample.shp', 'streams.shp', 'trails.shp', 'veg.shp', 'water.shp']
```

Assign to inputFile a path to the samples.csv file you created earlier, and then create a textfile read object with  
textin = open(inputFile, "r")

- Have a careful look at the samples.csv file you created earlier by opening it in a text editor like Notepad (not Excel). -- Note that it's comma-delimited, and the first line of text is the list of field names. Get the field names as a list with flds[0]

In [ ]: #

Out[ ]: ['sample\_id', 'CATOT', 'SHAPE@X', 'SHAPE@Y']

Continue borrowing code from Mb3\_InsertCreatePts.py, but change the AddField statements to instead add "sample\_id" of type "LONG" and "CATOT" of type "DOUBLE". This should make sense from your review of the samples.csv file.

For now, make sure to include the various print(arcpy.ListFeatureClasses()) statements to wake things up, due to some kind of bug related to Pro.

In [ ]: #

```
['co2july95.shp', 'contour.shp', 'geology.shp', 'marblePts.shp', 'mvalley_pts.shp', 'randomPolygons.shp', 'randomPolylines.shp', 'samplePts.shp', 'sample.shp', 'streams.shp', 'trails.shp', 'veg.shp', 'water.shp']
```

Cursor processing while reading in data:

- You might be tempted to use the field names detected above, but we'll want to create `SHAPE@XY` instead of a separate `SHAPE@X` and `SHAPE@Y`, so set up the cursor with

```
cur = arcpy.da.InsertCursor(ptfeatname, ("sample_id", "CATOT", "SHAPE@XY"))
```

- After the insert cursor creation line, create a Boolean flag to allow you to ignore the first row of text from `textin`: ````

```
firstrow = True
```

- The loop will be different: use

```
for pt in textin:
```

then within the loop, use

```
if not firstrow:
```

to only create new point features when not the first row of text.

- Read in the values from the row of text, splitting at the commas:

```
dta = txtrow.split(",")
```

- Build the point geometric object with the X & Y values from the text file (notice how this is different from what we did before, and that we have to float the text), then insert it.

```
outdta = (int(dta[0]), float(dta[1]), (float(dta[2]), float(dta[3])))  
cur.insertRow(outdta)
```

After the `if` structure (so unindented), set `firstrow` to `False`.

In [ ]: #

- Close the text file, delete the cursor and use `AddXY` the same as before.

In [ ]: #

Out[ ]:

## Messages

In an ArcGIS map you should see the results (refresh the map if necessary).

### ... or outside ArcGIS, use Geopandas:

Then using similar code to what you used before with geopandas and matplotlib, create a map of the samplepts.

```
import geopandas as gpd
import pandas as pd
import matplotlib.pyplot as plt
samplepts = gpd.read_file('curdata/' + ptfeatname)
samplepts

samplepts.plot(column="CATOT", cmap="YlOrBr")
```

In [ ]: #

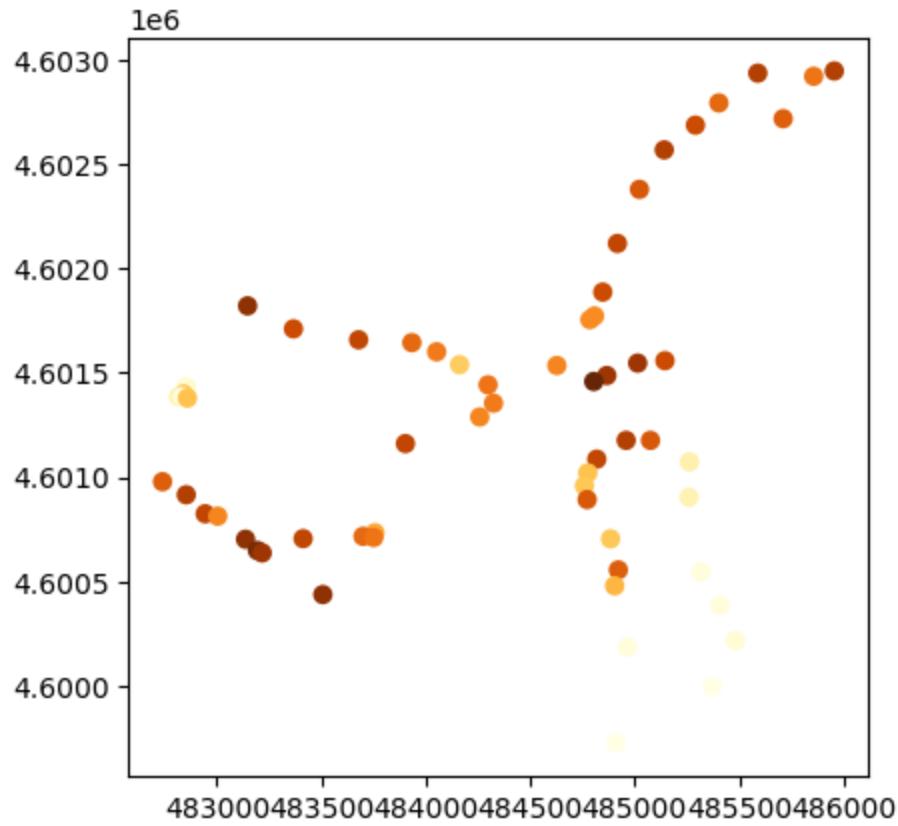
Out[ ]:

	<b>Id</b>	<b>sample_id</b>	<b>CATOT</b>	<b>POINT_X</b>	<b>POINT_Y</b>	<b>geometry</b>
<b>0</b>	0	49	0.83	485956.28125	4602944.5	POINT (485956.281 4602944.500)
<b>1</b>	0	47	0.83	485589.78125	4602934.0	POINT (485589.781 4602934.000)
<b>2</b>	0	4	0.63	485857.50000	4602919.0	POINT (485857.500 4602919.000)
<b>3</b>	0	46	0.67	485403.43750	4602791.5	POINT (485403.438 4602791.500)
<b>4</b>	0	1	0.70	485711.37500	4602716.0	POINT (485711.375 4602716.000)
...	...	...	...	...	...	...
<b>56</b>	0	36	0.05	485408.84375	4600386.5	POINT (485408.844 4600386.500)
<b>57</b>	0	35	0.07	485482.43750	4600217.5	POINT (485482.438 4600217.500)
<b>58</b>	0	8	0.05	484966.03125	4600185.5	POINT (484966.031 4600185.500)
<b>59</b>	0	6	0.02	485371.46875	4599995.5	POINT (485371.469 4599995.500)
<b>60</b>	0	7	0.02	484911.09375	4599726.5	POINT (484911.094 4599726.500)

61 rows × 6 columns

In [ ]: #

Out[ ]: &lt;AxesSubplot: &gt;



## Create random polylines

Let's create some polylines. There can be many situations where you might want to create polylines or polygons from data you provide to the script. These could be from real data but random elements can also be useful. We'll keep it simple and generate random lines, using a "random-walk" algorithm similar to what might be used in an agent-based model, though greatly simplified here. We'll generate these random lines to fit within the extent of our marbles study area.

1. Start with the same boilerplate you've been using to reference the curdata workspace, importing DM, etc.  
Add random to the list of imported modules.

In [ ]: #

1. Also as we've done above, use Describe and its spatialReference property to get the spatial reference of samples.shp, and assign it to sr.

In [ ]: #

1. Then also from the Describe object, get XMin , XMax , YMin , and YMax from the Extent property and assign these to simple variables xmin , xmax , ymin and ymax . We'll use these to make sure our lines end up in the study area.

In [ ]: #

1. Assign xrg ("x range") as XMax-XMin, and yrg as YMax-YMin, then xstp as xrg/20 and ystp as yrg/20. We're going to use these to create visible spacing of vertices on our polylines.

In [ ]: #

1. Assign the filename "randomPolylines.shp" to a variable featfile, then delete it if it exists.

In [ ]: #

1. Create a feature class in ws with that filename in featfile, using the parameters: (ws, featfile, "POLYLINE", "", "", "", sr)

In [ ]: #

1. Create an InsertCursor called cur using id and SHAPE@ as fields. So assign to cur the following:  
arcpy.da.InsertCursor(ws + "\\" + featfile, ("id", "SHAPE@"))

In [ ]: #

1. Create a method variable rnd from the random method of the random module. This method returns a random floating-point number between 0 and 1: rnd = random.random

In [ ]: #

1. In order to create 12 polylines, create a for loop that repeats 12 times using `i` as index. Within that loop:

- Create an empty list called `pointlist`. Polylines and polygons are built from series of vertices, and we'll need to create each vertex as a point object.
- Create an initial point that will fit within the extent (study the method used to see how it fits within the extent), using the `rnd` method assigned above: `p = arcpy.Point(rnd() * xng + xmin, rnd() * yrg + ymin)`
- Append the point to `pointlist`

Then create an interior `for j` loop that goes 30 times to create plenty of vertices for each polyline. Note that vertices may extend a bit beyond the extent. We could test for that and avoid it, but we'll just go with it for simplicity. Note also that we'll reuse the `p` point feature, which doesn't cause any problems.

- Create a new `p` point that diverges from the previous point by a random distance from zero to `xstp` and `ystp` distances: `p = arcpy.Point(p.X + xstp*rnd() - xstp*rnd(), p.Y + ystp*rnd() - ystp*rnd())`
- Append that point to `pointlist` Then create an array from `pointlist`, then make a polyline from it:

```
array = arcpy.Array(pointlist)
polyline = arcpy.Polyline(array)
```

Insert that polyline as an inserted row in the cursor

```
cur.insertRow([i, polyline])
```

In [ ]: #

1. Finally, delete the cursor with `del cur`.

In [ ]: #

You should see the result displayed in ArcGIS... (but you may need to refresh the map)

## Or in Geopandas, similar code to what we used earlier.

Again only if you're not in ArcGIS Pro, you can see results displayed with:

```
import geopandas as gpd
import matplotlib.pyplot as plt
randomlines = gpd.read_file('curdata/' + featfile)
randomlines

randomlines.plot(column="Id", cmap=("Spectral"))
```

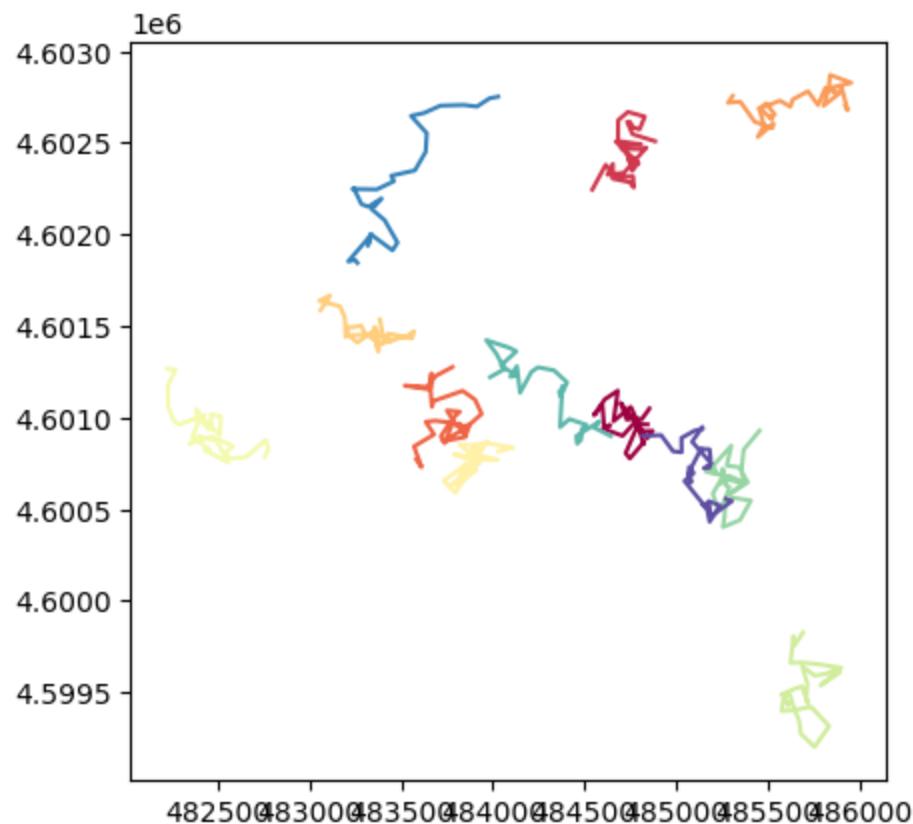
In [ ]: #

Out[ ]:

	<b>Id</b>	<b>geometry</b>
0	0	LINESTRING (484855.538 4600963.158, 484775.111...
1	1	LINESTRING (484536.985 4602234.953, 484615.485...
2	2	LINESTRING (483574.810 4600782.501, 483610.263...
3	3	LINESTRING (485918.059 4602715.329, 485935.273...
4	4	LINESTRING (483380.354 4601547.181, 483406.847...
5	5	LINESTRING (484018.669 4600813.960, 484086.832...
6	6	LINESTRING (482749.179 4600774.847, 482775.303...
7	7	LINESTRING (485698.447 4599836.956, 485646.521...
8	8	LINESTRING (485237.285 4600636.199, 485286.713...
9	9	LINESTRING (483974.995 4601215.661, 484062.971...
10	10	LINESTRING (484039.708 4602754.249, 483981.857...
11	11	LINESTRING (484808.663 4600900.560, 484918.152...

In [ ]: #

Out[ ]: <AxesSubplot: >



## Random polygons

Creating polygons is almost identical to creating polylines, just that they close from the last point to the first point again. Take the code you just wrote and convert it in various places to create polygons. These are the changes:

- "randomPolylines.shp" becomes "randomPolygons.shp"
- When creating the feature class, use "POLYGON" instead of "POLYLINE"
- The array of points is used to create polygons with `polygon = arcpy.Polygon(array)`, then that polygon feature is inserted with `cur.insertRow([i, polygon])`
- In ArcGIS Pro, you may need to refresh the map, but you should see the results on an open map.

In [ ]: #

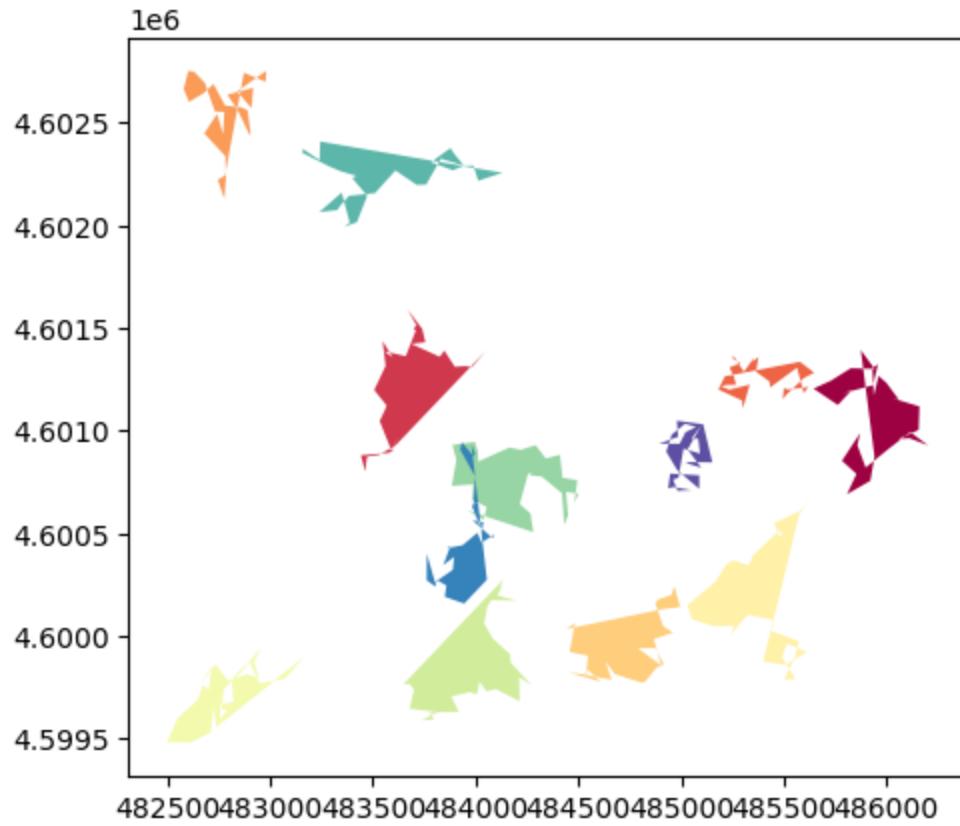
## Geopandas display

Again, only if you're running this outside ArcGIS Pro, you can display your results in Geopandas with:

```
import geopandas as gpd
import matplotlib.pyplot as plt
randompolys = gpd.read_file('curdata/' + featfile)
randompolys
```

In [ ]: #

Out[ ]: &lt;AxesSubplot: &gt;



## Using pandas with arcpy

Earlier we looked at the data analysis python library called pandas, and there we used csv files as input. We can also use it together with arcpy and its cursors and geodatabases.

We'll explore creating various data types using search cursors on a feature class and then loading that data into a pandas dataframe. Then we'll explore various methods and properties available to us to access data within our dataframes. We'll conclude with converting a table to a numpy array.

Enter the code below in your first cell:

```
import pandas as pd
import arcpy, os
ws = os.getcwd()
```

In [ ]: #

## Loading various data types in a pandas dataframe

- Next, we'll want to assign the schools variable to our "SF\_Schools" feature class:

```
schools = ws + r"\SF.gdb"
```

In [ ]: #

- Then let's see what the field names are:

```
fields = []
for fld in arcpy.ListFields(schools):
    fields.append(fld.name)
fields
```

In [ ]: #

Out[ ]: ['OBJECTID',
 'Shape',
 'X',
 'Y',
 'Match\_addr',
 'Name',
 'District',
 'County',
 'Street',
 'City',
 'State',
 'ZIP9',
 'DistType',
 'Type',
 'Latitude',
 'Longitude',
 'Grades',
 'Status\_1']

- Once we've defined the schools feature class we're going to go through multiple examples of loading different data types into a pandas dataframe. Below are examples where we can generate a list, dictionary, numpy array, etc. all with using arcpy search cursors. First, let's examine how we can build a **list** of lists and load that into a dataframe.

We'll just list the first few list members, and continue that practice in later code chunks...

```
#create a list of lists
school_list = []
with arcpy.da.SearchCursor(schools, ["OBJECTID", "X", "Y", "Name", "District", "City", "Type" ]) as cur:
    for row in cur:
        school_list.append(list(row[0:])) # we append a list of row values to larger list [row[0], row[1], row[2], row[3]]
school_list[:3]
```

In [ ]: #

Out[ ]: [[1,  
-122.419992341,  
37.7766494543,  
'Alternative/Opportunity',  
'San Francisco County Office of Education',  
'San Francisco',  
'ALTERNATIVE'],  
[2,  
-122.463582,  
37.763352,  
'Cross Cultural Environmental Leadership (xcel) Acad',  
'San Francisco Unified',  
'San Francisco',  
'HIGH SCHOOL'],  
[3,  
-122.395977,  
37.7192,  
'KIPP Bayview Academy',  
'San Francisco Unified',  
'San Francisco',  
'MIDDLE']]

⌚ What data type is returned from the above block of code? What is the data type of "School\_list"?

..

**Answer:** A list of lists

💻 We can create a dataframe from the object "school\_list" above. In this situation we'll need to define the corresponding column names when we load the data into the dataframe.

```
#load our school list into a pandas Dataframe
df_schools_list = pd.DataFrame(school_list, columns = ["OBJECTID", "Name", "District", "City", "Type"])
df_schools_list[:3]
```

In [ ]: #

Out[ ]:

	OBJECTID	X	Y	Name	District	City	Type
0	1	-122.419992	37.776649	Alternative/Opportunity	San Francisco County Office of Education	San Francisco	ALTERNATIVE
1	2	-122.463582	37.763352	Cross Cultural Enviromental Leadership (xcel ...)	San Francisco Unified	San Francisco	HIGH SCHOOL
2	3	-122.395977	37.719200	KIPP Bayview Academy	San Francisco Unified	San Francisco	MIDDLE

Next, can also create a list of **tuples** data type and load that into a dataframe. When we create a search cursor the row object that is returned, is returned as a **tuple** data type.

```
#create a list of tuples
school_list_tuples = []
with arcpy.da.SearchCursor(schools, ["OBJECTID", "Name", "District", "City", "Type"]) as cur:
    for row in cur:
        school_list_tuples.append(row[0:]) # the row object in search cursor returns the row as a tuple
school_list_tuples[:3]
```

In [ ]: #

```
Out[ ]: [(1,
-122.419992341,
37.7766494543,
'Alternative/Opportunity',
'San Francisco County Office of Education',
'San Francisco',
'ALTERNATIVE'),
(2,
-122.463582,
37.763352,
'Cross Cultural Enviromental Leadership (xcel) Acad',
'San Francisco Unified',
'San Francisco',
'HIGH SCHOOL'),
(3,
-122.395977,
37.7192,
'KIPP Bayview Academy',
'San Francisco Unified',
'San Francisco',
'MIDDLE')]
```

Then we'll load this list of **tuples** object we've built into a pandas dataframe.

```
df_schools_tuples = pd.DataFrame(school_list_tuples, columns = ["OBJECTID","Name",
"District", "City", "Type"])
df_schools_tuples[:3]
```

In [ ]: #

Out[ ]:

	OBJECTID	X	Y	Name	District	City	Type
0	1	-122.419992	37.776649	Alternative/Opportunity	San Francisco County Office of Education	San Francisco	ALTERNATIVE
1	2	-122.463582	37.763352	Cross Cultural Enviromental Leadership (xcel) ...	San Francisco Unified	San Francisco	HIGH SCHOOL
2	3	-122.395977	37.719200	KIPP Bayview Academy	San Francisco Unified	San Francisco	MIDDLE

Now, let's build a **dictionary** using a search cursor and then load that dictionary into a dataframe. What's important is that you understand the structure of the dictionary we're loading into the dataframe. We need to correspond the key to the field name and all the values of that column in the attribute table to that key.

```
school_dict = {}
fields = ["OBJECTID", "Name", "District", "City", "Type"]
with arcpy.da.SearchCursor(schools, fields) as cur:
    for row in cur:
        for field_name, data_row in zip(fields, row):
            school_dict.setdefault(field_name, []).append(data_row)
school_dict.keys()
```

In [ ]: #

Out[ ]: dict\_keys(['OBJECTID', 'X', 'Y', 'Name', 'District', 'City', 'Type'])

With this dictionary, we can create another pandas dataframe.

```
df_school_dict = pd.DataFrame(school_dict)
df_school_dict.keys()
```

In [ ]: #

Out[ ]: Index(['OBJECTID', 'X', 'Y', 'Name', 'District', 'City', 'Type'], dtype='object')

Print out each of the data types we created in the above steps:

```
print(type(school_list))
```

```
print(type(school_dict))
```

```
print(type(school_list_tuples[0]))
```

In [ ]: #

```
<class 'list'>
<class 'dict'>
<class 'tuple'>
```

# Convert a featureclass to a numpy array then a dataframe then a geodataframe

We've just been working with the `SF_Schools` feature class from `SF.gdb`. Let's look at a method for converting this to a numpy array, and then a dataframe, finally a geodataframe.

- We'll start by creating the path to the schools featureclass again.

```
schools = ws + r"\SF.gdb\SF_Schools"
```

In [ ]: #

## Create a numpy array from the feature class

- Next we'll use that featureclass path to create a numpy array:

```
arr = arcpy.da.TableToNumPyArray(schools, ("Name", "District", "City", "Type"))
arr[:3]
```

In [ ]: #

Out[ ]: array([(-122.41999234, 37.77664945, 'Alternative/Opportunity', 'San Francisco County Office of Education', 'San Francisco', 'ALTERNATIVE'),
 (-122.463582, 37.763352, 'Cross Cultural Environmental Leadership (xcel) Acad', 'San Francisco Unified', 'San Francisco', 'HIGH SCHOOL'),
 (-122.395977, 37.7192, 'KIPP Bayview Academy', 'San Francisco Unified', 'San Francisco', 'MIDDLE')],  
 dtype=[('X', '<f8'), ('Y', '<f8'), ('Name', '<U254'), ('District', '<U254'), ('City', '<U254'), ('Type', '<U254')])

## Create a dataframe from the numpy array

- After you've created your NumPy array you can load it into a dataframe using `pd.DataFrame()`.

```
schools_df = pd.DataFrame(arr)
schools_df
```

In [ ]: #

Out[ ]:

	X	Y	Name	District	City	Type
0	-122.419992	37.776649	Alternative/Opportunity	San Francisco County Office of Education	San Francisco	ALTERNATIVE
1	-122.463582	37.763352	Cross Cultural Enviromental Leadership (xcel) ...	San Francisco Unified	San Francisco	HIGH SCHOOL
2	-122.395977	37.719200	KIPP Bayview Academy	San Francisco Unified	San Francisco	MIDDLE
3	-122.437025	37.783220	KIPP San Francisco Bay Academy	San Francisco Unified	San Francisco	MIDDLE
4	-122.402234	37.777109	Five Keys Charter (SF Sheriff's)	San Francisco Unified	San Francisco	HIGH SCHOOL
...	...	...	...	...	...	...
186	-122.499586	37.750777	Sunset Elementary	San Francisco Unified	San Francisco	ELEMENTARY
187	-122.419744	37.781813	Tenderloin Community	San Francisco Unified	San Francisco	ELEMENTARY
188	-122.434098	37.783621	Western Addition Academy	San Francisco Unified	San Francisco	ELEMENTARY
189	-122.434247	37.784805	The Jump Academy	San Francisco Unified	San Francisco	ELEMENTARY
190	-122.426158	37.754743	Edison Charter Academy	SBE - Edison Charter Academy	San Francisco	ELEMENTARY

191 rows × 6 columns

## Convert to geodataframe and mapping in geopandas

- Now that we have a dataframe with X & Y coordinates in a known crs ("EPSG:4326"), we can make a geodataframe from it.

```
import geopandas as gpd
sf_schools = gpd.GeoDataFrame(schools_df, geometry=gpd.points_from_xy(schools_df.X, schools_df.Y), crs="EPSG:4326")
```

In [ ]: #

- We can also reproject it to web mercator to use with a contextily basemap.

```
sf_schools_webm = sf_schools.to_crs("EPSG:3857")
import contextily as cx
basemap, basemap_extent = cx.bounds2img(*sf_schools_webm.total_bounds, zoom=12, ll
=False)
```

In [ ]: #

- Then use similar code to what we've used before to plot the school locations

```
from matplotlib import pyplot as plt
f, ax1 = plt.subplots(1, figsize=(20,20))
ax1.set_title("San Francisco Schools")
plt.imshow(basemap, extent=basemap_extent)
sf_schools_webm.plot(ax=plt.gca(), marker='o', markersize=18, color="purple")
ax1.set_axis_off()
plt.axis()
```

In [ ]:

#

Out[ ]: (-13638811.83098057,  
-13619243.951739563,  
4529964.044292687,  
4559315.8631541915)

## San Francisco Schools



**key**

- ➡ This directs you to do something specific, maybe in the operating system or answer something conceptual.
- 💻 Coding you need to do, in the subsequent code cell.
- ❓ Questions to answer in the same markdown cell.
- Response to question or need for interpretation.

In [ ]: