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.

铋 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.
**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:

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

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

Enter the square of a number, follow it with **2**.

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

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".
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, e.g. “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

```python
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`

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

```python
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`.

```python
In [ ]: False * 2
Out[ ]: 0
In [ ]: True * 2
Out[ ]: 2
```

```python
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[ ]: 0.4999999999999999
Out[ ]: 0.4999999999999999
```

- **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

\[
x = 2
\]
\[
x.
\]
\[
x . denominator
\]

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.

```python
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

```python
In [ ]: #
Out[ ]: [5, 7.83, 'Fred']
```
You can also populate a list with previously assigned variables or other objects like lists.

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

```python
define variables and combine with constants:
```python

```python
In [ ]: 
```python
```
```python
```python
```python
```python
```
```python
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:

```python
lyr = ["geology", "landuse", "publands", "streams", "cities"]
```

%tabular
<table>
<thead>
<tr>
<th>Position</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>geology</td>
</tr>
<tr>
<td>1</td>
<td>landuse</td>
</tr>
<tr>
<td>2</td>
<td>publands</td>
</tr>
<tr>
<td>3</td>
<td>streams</td>
</tr>
<tr>
<td>4</td>
<td>cities</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

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

```python
lyr[1] starts at beginning of "landuse", so it refers to "landuse"
```
In essence, this is shorthand for what could be given as:

```python
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]`

```python
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?

```python
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:

```python
In [ ]: #
Out[ ]: ['streams', 'cities']
```

⌨ How would you get only the last item?

```python
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 [ ]:
```python
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:

```python
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]

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").

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']
```

```
States = Pacific
States.append(Desert)
States
```

```
In [ ]: States = Pacific
States.append(Desert)
States
Out[ ]: ['AK', 'CA', 'OR', 'WA', 'AZ', 'NV', 'UT', 'ID', 'MT', 'WY', 'CO', 'NM']
```

Sorting lists

`.sort` sorts a list. If we have a random list...
In [ ]:
```python
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:

```python
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.

```python
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:

```python
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:

```python
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)
```

```
[(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:

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

```
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.DataFrames, 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:

```python
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:

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

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

PRECIPITATION["PLACERVILLE"]
```

In [ ]: #

Out [ ]: 170.69

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

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

In [ ]: #

Out [ ]: {'GROVELAND': 176.02, 'LEE VINING': 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.

```python
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:**

```python
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'>

 résultation (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.

```python
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 modulos in a cycle.
```

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.

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

```
In [ ]: #
Out[ ]: '5.278'
```

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

```
In [ ]: #
Out[ ]: 4
```

```python
int("4.17")
```

❓ Why did you get an error?

```
In [ ]: #
```

```python
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:

```python
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.

```python
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
```

### 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

```python
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:

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

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`.

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

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.

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

:: **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.

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

:: **Interpret the above, and provide other examples:**

```python
In [ ]: #
"elev" > 1000
```

:: **Interpret the above:**
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.

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

Interpret the above:

```
In [1]: 
10
.shp
['d:', 'work', 'lu.shp']
```

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:

```python
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.*

```python
In [2]: 
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 ≥, ≠, or ≤).

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)

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

<table>
<thead>
<tr>
<th>In [ ]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
</tr>
</tbody>
</table>

```
True
False
False
```

<table>
<thead>
<tr>
<th>In [ ]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>True &amp; False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Out[ ]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
</tr>
</tbody>
</table>

Interpret what the above is showing us.
This is a little weird, but what does it show us?

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

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:

```python
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)
```

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 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:

```python
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:

```python
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:

```python
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:

```python
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)
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.

\[
\begin{align*}
\text{asin} &= \text{math.asin}; \ \text{acos} = \text{math.acos}; \ \text{atan2} = \text{math.atan2} \\
\sin &= \text{math.sin(math.radians(45))} \\
\sin &= \text{math.sin(45/180*pi)} \\
\math.degrees &= \text{math.degrees(asin(0.5))} \\
\text{asin(0.5)} &= \text{math.degrees(asin(0.5))} \\
\end{align*}
\]

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.
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:

```python
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)
```

Interpret what the above is showing us. :

The next graph using these data may help.
```python
import matplotlib.pyplot
pyplot.plot([14,17,17,14],[8,8,12,8])
```

```
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?

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

Interpret what the above is showing us.

How about a normal distribution?

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

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

`Interpret what the above is showing us. `
Sorting and frequency counting some random numbers

```python
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, 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.

�建 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:

<table>
<thead>
<tr>
<th>operation</th>
<th>when to operate</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>Only execute the statements if a particular condition is true</td>
</tr>
<tr>
<td>while</td>
<td>Keep executing (loop through) the set multiple times while a condition is true</td>
</tr>
<tr>
<td>for</td>
<td>Loop through the set for each value in a range of values</td>
</tr>
</tbody>
</table>

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.
**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:

<table>
<thead>
<tr>
<th>significant date</th>
<th>solar declination</th>
</tr>
</thead>
<tbody>
<tr>
<td>June solstice (June 21)</td>
<td>23.44</td>
</tr>
<tr>
<td>Equinox (Mar 21, Sept 21)</td>
<td>0</td>
</tr>
<tr>
<td>December solstice (Dec. 21)</td>
<td>-23.44</td>
</tr>
</tbody>
</table>

Solar Declination Angle
(Northern Hemisphere)

```python
# 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.)
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:

```python
print('

Start 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.

```python
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 of the file explorer window until it changes to display the path, looking something like mine: `C:\py\ex01\data`.

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

```python
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 \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.

```python
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:

```python
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 [ ]:

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

Try the following code, which illustrates a for loop:

```python
for x in [1, 2, 3, 4]:
    msg = "Hello World"
    print(str(x) + " " + msg)
```
Replace the list \([1, 2, 3, 4]\) with \(\text{range}(4)\).

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

Note the use of lists and the \(\text{len}\) function in the following.

```python
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?

```python
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.

 CFR 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.

```python
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.

```python
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 ...

```python
decls
sunangles
azimuths
```

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.

❓ 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.

```python
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`.

```python
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.

```python
In [ ]: #
```

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

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

```python
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...)

```python
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')

Out[ ]: 500.0

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° 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° to the north and going around clockwise, in contrast to the way you would have learned it in math classes with 0° 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 \(dYlab\))

\[
\begin{align*}
\text{plt.text(*a,"a",size=24)} \\
\text{plt.text(*b,"b",size=24)}
\end{align*}
\]

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 \(\text{distangle()}\) 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.

```python
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.

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

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

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

```
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:

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

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

```python
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)))
```
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.

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

```bash
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.
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:

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

or

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

---

**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:

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

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

```python
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:

```python
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.

```python
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:

```python
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:

```python
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:

```python
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': '460001']
```
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.

```python
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.

```python
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 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 where you can also see the code that generates them.

For our first Matplotlib plot, we'll create a simple line plot from thee 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.

```python
import matplotlib
from matplotlib import pyplot
pyplot.plot([1,2,3],[4,5,1])
```
A slightly more efficient way to start the plot is to import pyplot as `plt`

```python
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:

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

Note that the axes expanded a bit to include the new feature.
Change `.plot` to `.scatter` to create points instead of lines.
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.
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:

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

```python
In [ ]: #
Out[ ]: [<matplotlib.lines.Line2D at 0x1ecdbf1d070>]
Out[ ]: [<matplotlib.lines.Line2D at 0x1ecdbb66040>]
```
In [ ]: #

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

Change the above to create green triangles with 'g^'
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), 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 \( c = a \times b \)
- 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:

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

That worked, but what did we get?

```
myArray
```

```
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.

```python
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:

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

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

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.

```python
b = np.array([1, 2, 3])
b
b.dtype
```
One float creates a float...

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

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

Tuple:

```python
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:

```python
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.

```python
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.

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

```python
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.

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

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

```python
a**2
```

Statistical summaries of ndarrays

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

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

```python
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]])
```

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

```python
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 for more NumPy methods for this.

Create an 2x3 ndarray of uniform random numbers with

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

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 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:

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

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.)

```python
r = np.random.default_rng(42)
r.random(5)
```

```python
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 [ ]:

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

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

Then extract the X and Y values as arrays.

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

```
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 \( r \cdot (\text{max} - \text{min}) + \text{min} \) if \( r \) is a random number (or an array of random numbers) between 0 and 1, the minimum value is \( \text{min} \), the maximum value is \( \text{max} \).

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 [ ]:

```
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))
```

```
array([[-0.52949271,  0.23267621,  0.02185215,  1.60177889, -0.23935633],
       [-1.02349749,  0.17927563,  0.2199668,  1.35918758,  0.83511125],
       [ 0.35857106,  1.46330289, -1.18876305, -0.63975153, -0.92657594],
       [-0.3898098 , -1.37668615,  0.63515095, -0.2222227 , -1.47080629]])
```
In [ ]: #

The mean (μ 'mu') is 100 and the standard deviation (σ '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.

```python
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.

```python
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:

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

```python
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
```
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:


Meshgrids for mathematical functions

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

```python
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()
```
... then use this to plot a function.

```python
[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="\(\frac{\sin(X^2 + Y^2)}{X^2 + Y^2}\)"
```

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

```python
def read_field_names(file):
    f = open(file, "r")
    fields = f.readline().replace(" ", "").split(",")
    return fields
```

```python
f = open("npdata/exported_ptdata.csv", "r")
fields = f.readline().replace(" ", "").split(",")
f.close()
```
In [ ]:

Out[ ]:

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.

```python
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,  4.60185500e+06,  5.90000000e+01, ..., 
                 -6.51900000e+01,  6.00000000e-03, -4.21000000e+00],
               [ 4.85371469e+05,  4.59999500e+06,  6.00000000e+01, ..., 
                 8.07600000e+00,  4.30000000e-02, -3.37000000e+00],
               [ 4.84911094e+05,  4.59972650e+06,  7.74600000e+01, ..., 
                 2.60000000e+00,  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>),
1)

... 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".

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

To make the scatterplot plot 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).

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

### Matplotlib elements: put it all together

Take all of the code starting from:

```python
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:

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

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

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.

Ascii key: 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:

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

Run the boilerplate that provides multiple outputs per cell:

```python
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.

```python
import pandas as pd
elevation = pd.Series([52, 394, 510, 564, 725, 848, 1042, 1225, 1486, 1775, 1899, 2551])
latitude = pd.Series([39.52, 38.91, 37.97, 38.70, 39.09, 39.25, 39.94, 37.75, 40.35, 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
```
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.

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

```
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
```

... or a constant ...

```python
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.)

```python
:```
... or some random numbers. The following creates \texttt{r} as a random number generator (rng) with a seed of 42.

\begin{verbatim}
r = np.random.default_rng(seed=42)
pd.Series(r.random(10))
\end{verbatim}

```
In []: 

Out[]:  
```

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

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 \texttt{new} series of elevation, but name it \texttt{elev} to retain both versions since we'll use each type below when we're building dataframes. We'll do the same for latitude (\texttt{Lat}) and temperature (\texttt{temp}).

\begin{verbatim}
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
\end{verbatim}
A series is like a NumPy ndarray, so we can do similar operations, such as: `elev[0]`

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

But if you want an actual ndarray, you can use `.to_numpy`.

Using the `.to_numpy` method on `elev`, derive `elevarray`:

**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.
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...

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

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

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

... and then display each to confirm that they have these named indices.
In [ ]: #

Out[ ]:  
<table>
<thead>
<tr>
<th>City</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oroville</td>
<td>39.52</td>
</tr>
<tr>
<td>Auburn</td>
<td>38.91</td>
</tr>
<tr>
<td>Sonora</td>
<td>37.97</td>
</tr>
<tr>
<td>Placerville</td>
<td>38.70</td>
</tr>
<tr>
<td>Colfax</td>
<td>39.09</td>
</tr>
<tr>
<td>Nevada City</td>
<td>39.25</td>
</tr>
<tr>
<td>Quincy</td>
<td>39.94</td>
</tr>
<tr>
<td>Yosemite</td>
<td>37.75</td>
</tr>
<tr>
<td>Sierraville</td>
<td>40.35</td>
</tr>
<tr>
<td>Truckee</td>
<td>39.33</td>
</tr>
<tr>
<td>Tahoe City</td>
<td>39.17</td>
</tr>
<tr>
<td>Bodie</td>
<td>38.21</td>
</tr>
<tr>
<td></td>
<td>dtype: float64</td>
</tr>
</tbody>
</table>

Out[ ]:  
<table>
<thead>
<tr>
<th>City</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oroville</td>
<td>10.7</td>
</tr>
<tr>
<td>Auburn</td>
<td>9.7</td>
</tr>
<tr>
<td>Sonora</td>
<td>7.7</td>
</tr>
<tr>
<td>Placerville</td>
<td>9.2</td>
</tr>
<tr>
<td>Colfax</td>
<td>7.3</td>
</tr>
<tr>
<td>Nevada City</td>
<td>6.7</td>
</tr>
<tr>
<td>Quincy</td>
<td>4.0</td>
</tr>
<tr>
<td>Yosemite</td>
<td>5.0</td>
</tr>
<tr>
<td>Sierraville</td>
<td>0.9</td>
</tr>
<tr>
<td>Truckee</td>
<td>-1.1</td>
</tr>
<tr>
<td>Tahoe City</td>
<td>-0.8</td>
</tr>
<tr>
<td>Bodie</td>
<td>-4.4</td>
</tr>
<tr>
<td></td>
<td>dtype: float64</td>
</tr>
</tbody>
</table>

### 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.**

```python
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.*
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.

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

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

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.

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

```
<table>
<thead>
<tr>
<th></th>
<th>temperature</th>
<th>elevation</th>
<th>latitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oroville</td>
<td>10.7</td>
<td>52</td>
<td>39.52</td>
</tr>
<tr>
<td>Auburn</td>
<td>9.7</td>
<td>394</td>
<td>38.91</td>
</tr>
<tr>
<td>Sonora</td>
<td>7.7</td>
<td>510</td>
<td>37.97</td>
</tr>
<tr>
<td>Placerville</td>
<td>9.2</td>
<td>564</td>
<td>38.70</td>
</tr>
<tr>
<td>Colfax</td>
<td>7.3</td>
<td>725</td>
<td>39.09</td>
</tr>
<tr>
<td>Nevada City</td>
<td>6.7</td>
<td>848</td>
<td>39.25</td>
</tr>
<tr>
<td>Quincy</td>
<td>4.0</td>
<td>1042</td>
<td>39.94</td>
</tr>
<tr>
<td>Yosemite</td>
<td>5.0</td>
<td>1225</td>
<td>37.75</td>
</tr>
<tr>
<td>Sierraville</td>
<td>0.9</td>
<td>1516</td>
<td>40.35</td>
</tr>
<tr>
<td>Truckee</td>
<td>-1.1</td>
<td>1775</td>
<td>39.33</td>
</tr>
<tr>
<td>Tahoe City</td>
<td>-0.8</td>
<td>1899</td>
<td>39.17</td>
</tr>
<tr>
<td>Bodie</td>
<td>-4.4</td>
<td>2551</td>
<td>38.21</td>
</tr>
</tbody>
</table>
```
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 [link](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
```
It's usually a good idea to set a useful named index:

```python
sierraFeb.set_index("STATION_NAME")
```
... and from that we can reference a single column as a series `sierraFeb.PRECIPITATION`
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:

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

Or add a series to an existing dataframe

```python
sierraFeb["ELEVATION_FT"] = sierraFeb.ELEVATION / 0.3048
sierraFeb
```
<table>
<thead>
<tr>
<th>In [ ]:</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[ ]:</td>
<td></td>
</tr>
<tr>
<td><strong>STATION_NAME</strong></td>
<td><strong>COUNTY</strong></td>
</tr>
<tr>
<td>GROVELAND 2, CA US</td>
<td>Tuolumne</td>
</tr>
<tr>
<td>CANYON DAM, CA US</td>
<td>Plumas</td>
</tr>
<tr>
<td>KERN RIVER PH 3, CA US</td>
<td>Kern</td>
</tr>
<tr>
<td>DONNER MEMORIAL ST PARK, CA US</td>
<td>Nevada</td>
</tr>
<tr>
<td>BOWMAN DAM, CA US</td>
<td>Nevada</td>
</tr>
<tr>
<td>PACIFIC HOUSE, CA US</td>
<td>El Dorado</td>
</tr>
<tr>
<td>MAMMOTH LAKES RANGER STATION, CA US</td>
<td>Mono</td>
</tr>
<tr>
<td>COLFAX, CA US</td>
<td>Placer</td>
</tr>
<tr>
<td>COLGATE POWERHOUSE, CA US</td>
<td>Yuba</td>
</tr>
<tr>
<td>BODIE CALIFORNIA STATE HISTORIC PARK, CA US</td>
<td>Mono</td>
</tr>
</tbody>
</table>

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)...

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

```text
In [ ]: #
```

... and then reading it in.

```python
sierraFebNew = pd.read_json("pdData/sierraFeb.json")
sierraFebNew
```
In [ ]:

```
#
```

Out[ ]:

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

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:

```python
infile = "pdData/sierraFeb.json"
f = open(infile, "r")
f.readline()
f.close()
```
In [ ]: #
1/27/24, 2:49 PM

Ex04_Pandas_intro_results

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 U
S","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 U
S","10":"CALAVERAS BIG TREES, CA US","11":"BUCKS CREEK, CA US","12":"GRASS VA
LLEY 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 AIR
PORT, 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":"FORESTH
ILL RANGER STATION, CA US","24":"MANZANITA LAKE, CA US","25":"OROVILLE, CA U
S","26":"QUINCY, CA US","27":"VISALIA, CA US","28":"PORTOLA, CA US","29":"BAL
CH POWER HOUSE, CA US","30":"DOWNIEVILLE, CA US","31":"HAIWEE, CA US","32":"V
OLTA POWER HOUSE, CA US","33":"CAMP PARDEE, CA US","34":"ELLERY LAKE, CA U
S","35":"TRUCKEE RANGER STATION, CA US","36":"BISHOP AIRPORT, CA US","37":"DE
SABLA, CA US","38":"LAKE SABRINA, CA US","39":"ASH MOUNTAIN, CA US","40":"SOU
TH 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":"SIERR
AVILLE R S, CA US","49":"CHESTER, CA US","50":"NEW MELONES DAM HQ, CA US","5
1":"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","6
1":"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","6
6":"YOSEMITE 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 U
S","72":"YOSEMITE PARK HEADQUARTERS, CA US","73":"BISHOP CRK INTAKE 2, CA U
S","74":"BOCA, CA US","75":"HUNTINGTON LAKE, CA US","76":"LODGEPOLE, CA U
S","77":"PACIFIC HOUSE, CA US","78":"MAMMOTH LAKES RANGER STATION, CA US","7
9":"COLFAX, CA US","80":"COLGATE POWERHOUSE, CA US","81":"BODIE CALIFORNIA ST
ATE HISTORIC PARK, CA US"},"COUNTY":{"0":"Tuolumne","1":"Plumas","2":"Ker
n","3":"Nevada","4":"Nevada","5":"Butte","6":"Tulare","7":"Mono","8":"Butt
e","9":"Tulare","10":"Calaveras","11":"Plumas","12":"Nevada","13":"El Dorad
o","14":"Tulare","15":"Kern","16":"Tuolumne","17":"Placer","18":"Mono","1
9":"Tehama","20":"Lassen","21":"Mono","22":"Placer","23":"Placer","24":"Shast
a","25":"Butte","26":"Plumas","27":"Tulare","28":"Plumas","29":"Fresno","3
0":"Sierra","31":"Inyo","32":"Shasta","33":"Calaveras","34":"Mono","35":"Neva
da","36":"Inyo","37":"Butte","38":"Inyo","39":"Tulare","40":"El Dorado","4
1":"Tulare","42":"Tuolumne","43":"Fresno","44":"Placer","45":"Madera","46":"N
evada","47":"Sacramento","48":"Sierra","49":"Plumas","50":"Tuolumne","51":"Bu
tte","52":"Mariposa","53":"Fresno","54":"Tuolumne","55":"Calaveras","56":"Mar
iposa","57":"Nevada","58":"Tuolumne","59":"Amador","60":"Fresno","61":"El Dor
ado","62":"Fresno","63":"Inyo","64":"Amador","65":"Yuba","66":"Mariposa","6
7":"El Dorado","68":"Butte","69":"Placer","70":"Kern","71":"Inyo","72":"Marip
osa","73":"Inyo","74":"Nevada","75":"Fresno","76":"Tulare","77":"El Dorad
o","78":"Mono","79":"Placer","80":"Yuba","81":"Mono"},"ELEVATION":{"0":853.
7,"28":1478.3,"29":528.8,"30":888.5,"31":1165.9,"32":676.7,"33":200.6,"34":29
39.8,"35":1774.9,"36":1250.3,"37":826.0,"38":2763.0,"39":520.6,"40":1924.5,"4
1":132.6,"42":1452.4,"43":637.0,"44":393.8,"45":806.2,"46":847.6,"47":89.9,"4
8":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":91
file:///C:/Users/900008452/Box/course/625/exer/exercise_html/results/Ex04_Pandas_intro_results.html

17/52


Set the index

Set the index to the Station Name:

sierraFebNew = sierraFebNew.set_index("STATION_NAME")
In [ ]: #

Out[ ]:

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

82 rows × 7 columns

**Transpose**

See what happens when you transpose `sierraFebNew` with `.transpose()`
In []: #

Out[]:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNTY</td>
<td>Tuolumne</td>
<td>Plumas</td>
<td>Kern</td>
<td>Nevada</td>
<td>Nevada</td>
<td>Butte</td>
<td></td>
</tr>
<tr>
<td>ELEVATION</td>
<td>853.4</td>
<td>1389.9</td>
<td>823.9</td>
<td>1809.6</td>
<td>1641.3</td>
<td>1085.1</td>
<td></td>
</tr>
<tr>
<td>LATITUDE</td>
<td>37.8444</td>
<td>40.1705</td>
<td>35.7831</td>
<td>39.3239</td>
<td>39.4539</td>
<td>39.6949</td>
<td></td>
</tr>
<tr>
<td>LONGITUDE</td>
<td>-120.2258</td>
<td>-121.0886</td>
<td>-118.4389</td>
<td>-120.2331</td>
<td>-120.6556</td>
<td>-121.3452</td>
<td></td>
</tr>
<tr>
<td>PRECIPITATION</td>
<td>176.02</td>
<td>164.08</td>
<td>67.06</td>
<td>167.39</td>
<td>276.61</td>
<td>296.16</td>
<td></td>
</tr>
<tr>
<td>TEMPERATURE</td>
<td>6.1</td>
<td>1.4</td>
<td>8.9</td>
<td>-0.9</td>
<td>2.9</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>ELEVATION_FT</td>
<td>2799.868766</td>
<td>4560.03937</td>
<td>2703.08399</td>
<td>5937.007874</td>
<td>5384.84252</td>
<td>3560.03937</td>
<td>660</td>
</tr>
</tbody>
</table>

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.

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

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

Some other options you can set:

```python
pd.set_option('display.max_columns', 100)
```

```python
pd.set_option('display.width', 1000)
```

Make some of these changes and call up your sf_weather dataframe to see the difference.
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.

```python
sf_weather = sf_weather.set_index('date_time')
sf_weather
```
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()
How would we display the first 10 lines of sf_weather? Figure this out by typing

```python
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.*

```python
In [ ]: #
```

```python
Out[ ]:
```

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

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[ ]:

```
   date_time  maxtempC  mintempC  maxtempF  mintempF  tempC  tempF  moonrise  moonset  sun
0  12/22/2019  12:00     14      10     57.2     50.0      12    53.6      3:33 AM  2:33 PM
1  12/23/2019  12:00     13       5     55.4     41.0      11    51.8      4:41 AM  3:11 PM
2  12/24/2019  12:00     11       8     51.8     46.4      11    51.8      5:48 AM  3:54 PM
3  12/25/2019  12:00     12       6     53.6     42.8      11    51.8      6:52 AM  4:43 PM
4  12/26/2019  12:00     14       8     57.2     46.4      12    53.6      7:52 AM  5:37 PM
5  12/27/2019  12:00     15       8     59.0     46.4      12    53.6      8:44 AM  6:34 PM
6  12/28/2019  12:00     12       8     53.6     46.4      11    51.8      9:30 AM  7:33 PM
7  12/29/2019  12:00     11      10     51.8     50.0      11    51.8     10:09 AM  8:32 PM
8  12/30/2019  12:00     17       8     62.6     46.4      13    55.4     10:42 AM  9:31 PM
9  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
0  1/1/2019  12:00    11
1  1/2/2019  12:00    10
2  1/3/2019  12:00    11
3  1/4/2019  12:00    12
4  1/5/2019  12:00    13
..  12/27/2019  12:00    15
5  12/28/2019  12:00    12
6  12/29/2019  12:00    11
7  12/30/2019  12:00    17
8  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 excepts 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[ ]: maxtempC            int64
       mintempC            int64
       maxtempF            float64
       mintempF            float64
       tempC                int64
       tempF                float64
       moonrise            object
       moonset             object
       sunrise             object
       sunset              object
      FeelsLikeC int64
       WindGustKmph         int64
       winddirDegree         int64
       windspeedKmph         int64
       location            object
       dtype: object
```

**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.

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

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.
# 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
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[ ]:

<table>
<thead>
<tr>
<th>date_time</th>
<th>maxtempC</th>
<th>mintempC</th>
<th>maxtempF</th>
<th>mintempF</th>
<th>tempC</th>
<th>tempF</th>
<th>moonrise</th>
<th>moonset</th>
<th>sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/1/2019 12:00</td>
<td>13</td>
<td>10</td>
<td>55.4</td>
<td>50.0</td>
<td>13</td>
<td>55.4</td>
<td>5:02 AM</td>
<td>2:58 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/2/2019 12:00</td>
<td>12</td>
<td>10</td>
<td>53.6</td>
<td>50.0</td>
<td>12</td>
<td>53.6</td>
<td>5:51 AM</td>
<td>3:49 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/3/2019 12:00</td>
<td>12</td>
<td>10</td>
<td>53.6</td>
<td>50.0</td>
<td>12</td>
<td>53.6</td>
<td>6:35 AM</td>
<td>4:42 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/4/2019 12:00</td>
<td>9</td>
<td>8</td>
<td>48.2</td>
<td>46.4</td>
<td>9</td>
<td>48.2</td>
<td>7:15 AM</td>
<td>5:38 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/5/2019 12:00</td>
<td>10</td>
<td>6</td>
<td>50.0</td>
<td>42.8</td>
<td>10</td>
<td>50.0</td>
<td>7:51 AM</td>
<td>6:33 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/6/2019 12:00</td>
<td>10</td>
<td>5</td>
<td>50.0</td>
<td>41.0</td>
<td>9</td>
<td>48.2</td>
<td>8:22 AM</td>
<td>7:30 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/7/2019 12:00</td>
<td>10</td>
<td>5</td>
<td>50.0</td>
<td>41.0</td>
<td>10</td>
<td>50.0</td>
<td>8:52 AM</td>
<td>8:26 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/8/2019 12:00</td>
<td>11</td>
<td>7</td>
<td>51.8</td>
<td>44.6</td>
<td>10</td>
<td>50.0</td>
<td>9:21 AM</td>
<td>9:23 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/9/2019 12:00</td>
<td>11</td>
<td>8</td>
<td>51.8</td>
<td>46.4</td>
<td>11</td>
<td>51.8</td>
<td>9:49 AM</td>
<td>10:20 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/10/2019 12:00</td>
<td>10</td>
<td>7</td>
<td>50.0</td>
<td>44.6</td>
<td>9</td>
<td>48.2</td>
<td>10:17 AM</td>
<td>11:19 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/11/2019 12:00</td>
<td>11</td>
<td>6</td>
<td>51.8</td>
<td>42.8</td>
<td>10</td>
<td>50.0</td>
<td>10:47 AM</td>
<td>No moonset</td>
<td>:</td>
</tr>
<tr>
<td>2/12/2019 12:00</td>
<td>11</td>
<td>7</td>
<td>51.8</td>
<td>44.6</td>
<td>11</td>
<td>51.8</td>
<td>11:21 AM</td>
<td>12:19 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/13/2019 12:00</td>
<td>14</td>
<td>11</td>
<td>57.2</td>
<td>51.8</td>
<td>14</td>
<td>57.2</td>
<td>11:59 AM</td>
<td>1:22 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/14/2019 12:00</td>
<td>14</td>
<td>11</td>
<td>57.2</td>
<td>51.8</td>
<td>11</td>
<td>51.8</td>
<td>12:43 PM</td>
<td>2:26 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/15/2019 12:00</td>
<td>11</td>
<td>9</td>
<td>51.8</td>
<td>48.2</td>
<td>11</td>
<td>51.8</td>
<td>1:37 PM</td>
<td>3:31 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/16/2019 12:00</td>
<td>11</td>
<td>9</td>
<td>51.8</td>
<td>48.2</td>
<td>10</td>
<td>50.0</td>
<td>2:38 PM</td>
<td>4:33 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/17/2019 12:00</td>
<td>9</td>
<td>7</td>
<td>48.2</td>
<td>44.6</td>
<td>9</td>
<td>48.2</td>
<td>3:47 PM</td>
<td>5:32 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/18/2019 12:00</td>
<td>12</td>
<td>5</td>
<td>53.6</td>
<td>41.0</td>
<td>10</td>
<td>50.0</td>
<td>5:00 PM</td>
<td>6:24 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/19/2019 12:00</td>
<td>11</td>
<td>5</td>
<td>51.8</td>
<td>41.0</td>
<td>10</td>
<td>50.0</td>
<td>6:15 PM</td>
<td>7:11 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/20/2019 12:00</td>
<td>11</td>
<td>6</td>
<td>51.8</td>
<td>42.8</td>
<td>11</td>
<td>51.8</td>
<td>7:29 PM</td>
<td>7:51 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/21/2019 12:00</td>
<td>11</td>
<td>6</td>
<td>51.8</td>
<td>42.8</td>
<td>10</td>
<td>50.0</td>
<td>8:40 PM</td>
<td>8:28 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/22/2019 12:00</td>
<td>10</td>
<td>5</td>
<td>50.0</td>
<td>41.0</td>
<td>9</td>
<td>48.2</td>
<td>9:50 PM</td>
<td>9:03 AM</td>
<td>:</td>
</tr>
</tbody>
</table>
We'll do some of this with the simpler sierra data

sierra

In [ ]: #

Out[ ]:

<table>
<thead>
<tr>
<th>temperature</th>
<th>elevation</th>
<th>latitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oroville</td>
<td>10.7</td>
<td>52</td>
</tr>
<tr>
<td>Auburn</td>
<td>9.7</td>
<td>394</td>
</tr>
<tr>
<td>Sonora</td>
<td>7.7</td>
<td>510</td>
</tr>
<tr>
<td>Placerville</td>
<td>9.2</td>
<td>564</td>
</tr>
<tr>
<td>Colfax</td>
<td>7.3</td>
<td>725</td>
</tr>
<tr>
<td>Nevada City</td>
<td>6.7</td>
<td>848</td>
</tr>
<tr>
<td>Quincy</td>
<td>4.0</td>
<td>1042</td>
</tr>
<tr>
<td>Yosemite</td>
<td>5.0</td>
<td>1225</td>
</tr>
<tr>
<td>Sierraville</td>
<td>0.9</td>
<td>1516</td>
</tr>
<tr>
<td>Truckee</td>
<td>-1.1</td>
<td>1775</td>
</tr>
<tr>
<td>Tahoe City</td>
<td>-0.8</td>
<td>1899</td>
</tr>
<tr>
<td>Bodie</td>
<td>-4.4</td>
<td>2551</td>
</tr>
</tbody>
</table>

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

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

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

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, 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[ ]:

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

```python
In [ ]:

Out[ ]:
```

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:

```python
sf_weather.iloc[0]
```
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[ ]:

<table>
<thead>
<tr>
<th>date_time</th>
<th>maxtempC</th>
<th>mintempC</th>
<th>maxtempF</th>
<th>mintempF</th>
<th>tempC</th>
<th>tempF</th>
<th>moonrise</th>
<th>moonset</th>
<th>sunrise</th>
<th>sunset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1/2019 12:00</td>
<td>11</td>
<td>5</td>
<td>51.8</td>
<td>41.0</td>
<td>11</td>
<td>51.8</td>
<td>3:17 AM</td>
<td>2:10 PM</td>
<td>7:25 AM</td>
<td>5:02 PM</td>
</tr>
<tr>
<td>1/2/2019 12:00</td>
<td>10</td>
<td>5</td>
<td>50.0</td>
<td>41.0</td>
<td>9</td>
<td>48.2</td>
<td>4:17 AM</td>
<td>2:47 PM</td>
<td>7:25 AM</td>
<td>5:02 PM</td>
</tr>
<tr>
<td>1/3/2019 12:00</td>
<td>11</td>
<td>5</td>
<td>51.8</td>
<td>41.0</td>
<td>10</td>
<td>50.0</td>
<td>5:17 AM</td>
<td>3:28 PM</td>
<td>7:25 AM</td>
<td>5:02 PM</td>
</tr>
<tr>
<td>1/4/2019 12:00</td>
<td>12</td>
<td>6</td>
<td>53.6</td>
<td>42.8</td>
<td>11</td>
<td>51.8</td>
<td>6:13 AM</td>
<td>4:13 PM</td>
<td>7:25 AM</td>
<td>5:02 PM</td>
</tr>
<tr>
<td>1/5/2019 12:00</td>
<td>13</td>
<td>8</td>
<td>55.4</td>
<td>46.4</td>
<td>13</td>
<td>55.4</td>
<td>7:06 AM</td>
<td>5:02 PM</td>
<td>7:25 AM</td>
<td>5:02 PM</td>
</tr>
</tbody>
</table>
```

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[ ]:

```
<table>
<thead>
<tr>
<th>date_time</th>
<th>maxtempC</th>
<th>mintempC</th>
<th>maxtempF</th>
<th>mintempF</th>
<th>tempC</th>
<th>tempF</th>
<th>moonrise</th>
<th>moonset</th>
<th>sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1/2019 12:00</td>
<td>11</td>
<td>5</td>
<td>51.8</td>
<td>41.0</td>
<td>11</td>
<td>51.8</td>
<td>3:17 AM</td>
<td>2:10 PM</td>
<td></td>
</tr>
<tr>
<td>1/2/2019 12:00</td>
<td>10</td>
<td>5</td>
<td>50.0</td>
<td>41.0</td>
<td>9</td>
<td>48.2</td>
<td>4:17 AM</td>
<td>2:47 PM</td>
<td></td>
</tr>
<tr>
<td>1/3/2019 12:00</td>
<td>11</td>
<td>5</td>
<td>51.8</td>
<td>41.0</td>
<td>10</td>
<td>50.0</td>
<td>5:17 AM</td>
<td>3:28 PM</td>
<td></td>
</tr>
</tbody>
</table>
```

Now create the last 3 rows and the last 10 columns

In [ ]: #

Out[ ]:

```
<table>
<thead>
<tr>
<th>date_time</th>
<th>tempF</th>
<th>moonrise</th>
<th>moonset</th>
<th>sunrise</th>
<th>sunset</th>
<th>FeelsLikeC</th>
<th>WindGustKmph</th>
<th>winddirDegr</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/29/2019 12:00</td>
<td>51.8</td>
<td>10:09 AM</td>
<td>8:32 PM</td>
<td>7:25 AM</td>
<td>4:59 PM</td>
<td>10</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>12/30/2019 12:00</td>
<td>55.4</td>
<td>10:42 AM</td>
<td>9:31 PM</td>
<td>7:25 AM</td>
<td>5:00 PM</td>
<td>13</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>12/31/2019 12:00</td>
<td>55.4</td>
<td>11:12 AM</td>
<td>10:28 PM</td>
<td>7:25 AM</td>
<td>5:01 PM</td>
<td>13</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
```

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` ...

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

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

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.

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:

```python
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})})
```

```python
In [ ]: #
```

```python
Out[ ]:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
```

Now we'll select two columns with `df[['x', 'y']]`
Then select two rows with `df.loc[['b','c']]`

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']]`

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:

```python
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:

```python
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

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

```python
sf_weather.std()
```

Do the same for standard deviation (std):
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:

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

Then to see the earliest sunset all year we’d use `min` instead

```python
sf_weather['sunset'].min()
```

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

```python
sf_weather['mintempF'].describe()
```

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()`.)
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:

```python
jan_sf_weather = sf_weather.head(n=31)
jan_sf_weather
```

```
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
```
<table>
<thead>
<tr>
<th>date_time</th>
<th>maxtempC</th>
<th>mintempC</th>
<th>maxtempF</th>
<th>mintempF</th>
<th>tempC</th>
<th>tempF</th>
<th>moonrise</th>
<th>moonset</th>
<th>sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1/2019 12:00</td>
<td>11</td>
<td>5</td>
<td>51.8</td>
<td>41.0</td>
<td>11</td>
<td>51.8</td>
<td>3:17 AM</td>
<td>2:10 PM</td>
<td></td>
</tr>
<tr>
<td>1/2/2019 12:00</td>
<td>10</td>
<td>5</td>
<td>50.0</td>
<td>41.0</td>
<td>9</td>
<td>48.2</td>
<td>4:17 AM</td>
<td>2:47 PM</td>
<td></td>
</tr>
<tr>
<td>1/3/2019 12:00</td>
<td>11</td>
<td>5</td>
<td>51.8</td>
<td>41.0</td>
<td>10</td>
<td>50.0</td>
<td>5:17 AM</td>
<td>3:28 PM</td>
<td></td>
</tr>
<tr>
<td>1/4/2019 12:00</td>
<td>12</td>
<td>6</td>
<td>53.6</td>
<td>42.8</td>
<td>11</td>
<td>51.8</td>
<td>6:13 AM</td>
<td>4:13 PM</td>
<td></td>
</tr>
<tr>
<td>1/5/2019 12:00</td>
<td>13</td>
<td>8</td>
<td>55.4</td>
<td>46.4</td>
<td>13</td>
<td>55.4</td>
<td>7:06 AM</td>
<td>5:02 PM</td>
<td></td>
</tr>
<tr>
<td>1/6/2019 12:00</td>
<td>13</td>
<td>8</td>
<td>55.4</td>
<td>46.4</td>
<td>12</td>
<td>53.6</td>
<td>7:54 AM</td>
<td>5:54 PM</td>
<td></td>
</tr>
<tr>
<td>1/7/2019 12:00</td>
<td>13</td>
<td>7</td>
<td>55.4</td>
<td>44.6</td>
<td>13</td>
<td>55.4</td>
<td>8:37 AM</td>
<td>6:48 PM</td>
<td></td>
</tr>
<tr>
<td>1/8/2019 12:00</td>
<td>13</td>
<td>9</td>
<td>55.4</td>
<td>48.2</td>
<td>13</td>
<td>55.4</td>
<td>9:15 AM</td>
<td>7:44 PM</td>
<td></td>
</tr>
<tr>
<td>1/9/2019 12:00</td>
<td>13</td>
<td>10</td>
<td>55.4</td>
<td>50.0</td>
<td>13</td>
<td>55.4</td>
<td>9:50 AM</td>
<td>8:40 PM</td>
<td></td>
</tr>
<tr>
<td>1/10/2019 12:00</td>
<td>12</td>
<td>9</td>
<td>53.6</td>
<td>48.2</td>
<td>12</td>
<td>53.6</td>
<td>10:20 AM</td>
<td>9:37 PM</td>
<td></td>
</tr>
<tr>
<td>1/11/2019 12:00</td>
<td>12</td>
<td>8</td>
<td>53.6</td>
<td>46.4</td>
<td>12</td>
<td>53.6</td>
<td>10:50 AM</td>
<td>10:33 PM</td>
<td></td>
</tr>
<tr>
<td>1/12/2019 12:00</td>
<td>13</td>
<td>8</td>
<td>55.4</td>
<td>46.4</td>
<td>12</td>
<td>53.6</td>
<td>11:18 AM</td>
<td>11:30 PM</td>
<td></td>
</tr>
<tr>
<td>1/13/2019 12:00</td>
<td>13</td>
<td>8</td>
<td>55.4</td>
<td>46.4</td>
<td>12</td>
<td>53.6</td>
<td>11:46 AM</td>
<td>No moonset</td>
<td></td>
</tr>
<tr>
<td>1/14/2019 12:00</td>
<td>12</td>
<td>8</td>
<td>53.6</td>
<td>46.4</td>
<td>12</td>
<td>53.6</td>
<td>12:15 PM</td>
<td>12:28 AM</td>
<td></td>
</tr>
<tr>
<td>1/15/2019 12:00</td>
<td>11</td>
<td>11</td>
<td>51.8</td>
<td>51.8</td>
<td>11</td>
<td>51.8</td>
<td>12:48 PM</td>
<td>1:29 AM</td>
<td></td>
</tr>
<tr>
<td>1/16/2019 12:00</td>
<td>13</td>
<td>9</td>
<td>55.4</td>
<td>48.2</td>
<td>13</td>
<td>55.4</td>
<td>1:24 PM</td>
<td>2:32 AM</td>
<td></td>
</tr>
<tr>
<td>1/17/2019 12:00</td>
<td>13</td>
<td>10</td>
<td>55.4</td>
<td>50.0</td>
<td>13</td>
<td>55.4</td>
<td>2:07 PM</td>
<td>3:38 AM</td>
<td></td>
</tr>
<tr>
<td>1/18/2019 12:00</td>
<td>12</td>
<td>8</td>
<td>53.6</td>
<td>46.4</td>
<td>12</td>
<td>53.6</td>
<td>2:57 PM</td>
<td>4:45 AM</td>
<td></td>
</tr>
<tr>
<td>1/19/2019 12:00</td>
<td>15</td>
<td>9</td>
<td>59.0</td>
<td>48.2</td>
<td>15</td>
<td>59.0</td>
<td>3:56 PM</td>
<td>5:52 AM</td>
<td></td>
</tr>
<tr>
<td>1/20/2019 12:00</td>
<td>13</td>
<td>10</td>
<td>55.4</td>
<td>50.0</td>
<td>12</td>
<td>53.6</td>
<td>5:03 PM</td>
<td>6:54 AM</td>
<td></td>
</tr>
<tr>
<td>1/21/2019 12:00</td>
<td>12</td>
<td>8</td>
<td>53.6</td>
<td>46.4</td>
<td>12</td>
<td>53.6</td>
<td>6:15 PM</td>
<td>7:50 AM</td>
<td></td>
</tr>
<tr>
<td>1/22/2019 12:00</td>
<td>13</td>
<td>8</td>
<td>55.4</td>
<td>46.4</td>
<td>12</td>
<td>53.6</td>
<td>7:30 PM</td>
<td>8:39 AM</td>
<td></td>
</tr>
</tbody>
</table>
Plot the data setting the y axis to the `maxtempF` field from our data frame:

```python
jan_sf_weather.plot(y=['maxtempF'], figsize=(15,15))
```
Plot the mintempF for the month of February.
In [ ]: 

```python
#
```
Out[ ]:

<table>
<thead>
<tr>
<th>date_time</th>
<th>maxtempC</th>
<th>mintempC</th>
<th>maxtempF</th>
<th>mintempF</th>
<th>tempC</th>
<th>tempF</th>
<th>moonrise</th>
<th>moonset</th>
<th>sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/1/2019 12:00</td>
<td>13</td>
<td>10</td>
<td>55.4</td>
<td>50.0</td>
<td>13</td>
<td>55.4</td>
<td>5:02 AM</td>
<td>2:58 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/2/2019 12:00</td>
<td>12</td>
<td>10</td>
<td>53.6</td>
<td>50.0</td>
<td>12</td>
<td>53.6</td>
<td>5:51 AM</td>
<td>3:49 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/3/2019 12:00</td>
<td>12</td>
<td>10</td>
<td>53.6</td>
<td>50.0</td>
<td>12</td>
<td>53.6</td>
<td>6:35 AM</td>
<td>4:42 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/4/2019 12:00</td>
<td>9</td>
<td>8</td>
<td>48.2</td>
<td>46.4</td>
<td>9</td>
<td>48.2</td>
<td>7:15 AM</td>
<td>5:38 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/5/2019 12:00</td>
<td>10</td>
<td>6</td>
<td>50.0</td>
<td>42.8</td>
<td>10</td>
<td>50.0</td>
<td>7:51 AM</td>
<td>6:33 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/6/2019 12:00</td>
<td>10</td>
<td>5</td>
<td>50.0</td>
<td>41.0</td>
<td>9</td>
<td>48.2</td>
<td>8:22 AM</td>
<td>7:30 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/7/2019 12:00</td>
<td>10</td>
<td>5</td>
<td>50.0</td>
<td>41.0</td>
<td>10</td>
<td>50.0</td>
<td>8:52 AM</td>
<td>8:26 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/8/2019 12:00</td>
<td>11</td>
<td>7</td>
<td>51.8</td>
<td>44.6</td>
<td>10</td>
<td>50.0</td>
<td>9:21 AM</td>
<td>9:23 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/9/2019 12:00</td>
<td>11</td>
<td>8</td>
<td>51.8</td>
<td>46.4</td>
<td>11</td>
<td>51.8</td>
<td>9:49 AM</td>
<td>10:20 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/10/2019 12:00</td>
<td>10</td>
<td>7</td>
<td>50.0</td>
<td>44.6</td>
<td>9</td>
<td>48.2</td>
<td>10:17 AM</td>
<td>11:19 PM</td>
<td>:</td>
</tr>
<tr>
<td>2/11/2019 12:00</td>
<td>11</td>
<td>6</td>
<td>51.8</td>
<td>42.8</td>
<td>10</td>
<td>50.0</td>
<td>10:47 AM</td>
<td>No</td>
<td>moonset</td>
</tr>
<tr>
<td>2/12/2019 12:00</td>
<td>11</td>
<td>7</td>
<td>51.8</td>
<td>44.6</td>
<td>11</td>
<td>51.8</td>
<td>11:21 AM</td>
<td>12:19 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/13/2019 12:00</td>
<td>14</td>
<td>11</td>
<td>57.2</td>
<td>51.8</td>
<td>14</td>
<td>57.2</td>
<td>11:59 AM</td>
<td>1:22 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/14/2019 12:00</td>
<td>14</td>
<td>11</td>
<td>57.2</td>
<td>51.8</td>
<td>11</td>
<td>51.8</td>
<td>12:43 PM</td>
<td>2:26 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/15/2019 12:00</td>
<td>11</td>
<td>9</td>
<td>51.8</td>
<td>48.2</td>
<td>11</td>
<td>51.8</td>
<td>1:37 PM</td>
<td>3:31 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/16/2019 12:00</td>
<td>11</td>
<td>9</td>
<td>51.8</td>
<td>48.2</td>
<td>10</td>
<td>50.0</td>
<td>2:38 PM</td>
<td>4:33 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/17/2019 12:00</td>
<td>9</td>
<td>7</td>
<td>48.2</td>
<td>44.6</td>
<td>9</td>
<td>48.2</td>
<td>3:47 PM</td>
<td>5:32 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/18/2019 12:00</td>
<td>12</td>
<td>5</td>
<td>53.6</td>
<td>41.0</td>
<td>10</td>
<td>50.0</td>
<td>5:00 PM</td>
<td>6:24 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/19/2019 12:00</td>
<td>11</td>
<td>5</td>
<td>51.8</td>
<td>41.0</td>
<td>10</td>
<td>50.0</td>
<td>6:15 PM</td>
<td>7:11 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/20/2019 12:00</td>
<td>11</td>
<td>6</td>
<td>51.8</td>
<td>42.8</td>
<td>11</td>
<td>51.8</td>
<td>7:29 PM</td>
<td>7:51 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/21/2019 12:00</td>
<td>11</td>
<td>6</td>
<td>51.8</td>
<td>42.8</td>
<td>10</td>
<td>50.0</td>
<td>8:40 PM</td>
<td>8:28 AM</td>
<td>:</td>
</tr>
<tr>
<td>2/22/2019 12:00</td>
<td>10</td>
<td>5</td>
<td>50.0</td>
<td>41.0</td>
<td>9</td>
<td>48.2</td>
<td>9:50 PM</td>
<td>9:03 AM</td>
<td>:</td>
</tr>
<tr>
<td>date_time</td>
<td>maxtempC</td>
<td>mintempC</td>
<td>maxtempF</td>
<td>mintempF</td>
<td>tempC</td>
<td>tempF</td>
<td>moonrise</td>
<td>moonset</td>
<td>sun</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>-------</td>
<td>-------</td>
<td>------------</td>
<td>---------</td>
<td>-----</td>
</tr>
<tr>
<td>2/23/2019 12:00</td>
<td>12</td>
<td>5</td>
<td>53.6</td>
<td>41.0</td>
<td>11</td>
<td>51.8</td>
<td>10:57 PM</td>
<td>9:37 AM</td>
<td></td>
</tr>
<tr>
<td>2/24/2019 12:00</td>
<td>11</td>
<td>8</td>
<td>51.8</td>
<td>46.4</td>
<td>11</td>
<td>51.8</td>
<td>No moonrise</td>
<td>10:11 AM</td>
<td></td>
</tr>
<tr>
<td>2/25/2019 12:00</td>
<td>11</td>
<td>4</td>
<td>51.8</td>
<td>39.2</td>
<td>10</td>
<td>50.0</td>
<td>12:01 AM</td>
<td>10:47 AM</td>
<td></td>
</tr>
<tr>
<td>2/26/2019 12:00</td>
<td>12</td>
<td>3</td>
<td>53.6</td>
<td>37.4</td>
<td>12</td>
<td>53.6</td>
<td>1:04 AM</td>
<td>11:26 AM</td>
<td></td>
</tr>
<tr>
<td>2/27/2019 12:00</td>
<td>14</td>
<td>9</td>
<td>57.2</td>
<td>48.2</td>
<td>14</td>
<td>57.2</td>
<td>2:03 AM</td>
<td>12:08 PM</td>
<td></td>
</tr>
<tr>
<td>2/28/2019 12:00</td>
<td>14</td>
<td>10</td>
<td>57.2</td>
<td>50.0</td>
<td>13</td>
<td>55.4</td>
<td>2:58 AM</td>
<td>12:55 PM</td>
<td></td>
</tr>
</tbody>
</table>

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:

```python
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 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."

```python
bayArea_unemployment_noparse = pd.read_csv('pdData/Unemployment_BA_Counties_2020_202105.csv')
bayArea_unemployment_noparse
```

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alameda</td>
<td>1487</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Marin</td>
<td>144</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>San Francisco</td>
<td>945</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>652</td>
<td>San Francisco</td>
<td>3878</td>
<td>233</td>
</tr>
<tr>
<td>653</td>
<td>San Mateo</td>
<td>2346</td>
<td>140</td>
</tr>
<tr>
<td>654</td>
<td>Santa Clara</td>
<td>5307</td>
<td>419</td>
</tr>
<tr>
<td>655</td>
<td>Solano</td>
<td>1580</td>
<td>140</td>
</tr>
<tr>
<td>656</td>
<td>Sonoma</td>
<td>1410</td>
<td>126</td>
</tr>
</tbody>
</table>

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.

```python
bayArea_unemployment = pd.read_csv('pdData/Unemployment_BA_Counties_2020_202105.csv', parse_dates= ['Week_Ending'])
bayArea_unemployment
```

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-01-11</td>
<td>Alameda</td>
<td>1487</td>
<td>0</td>
</tr>
<tr>
<td>2020-01-11</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
</tr>
<tr>
<td>2020-01-11</td>
<td>Marin</td>
<td>144</td>
<td>0</td>
</tr>
<tr>
<td>2020-01-11</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
</tr>
<tr>
<td>2020-01-11</td>
<td>San Francisco</td>
<td>945</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2021-05-29</td>
<td>San Francisco</td>
<td>3878</td>
<td>233</td>
</tr>
<tr>
<td>2021-05-29</td>
<td>San Mateo</td>
<td>2346</td>
<td>140</td>
</tr>
<tr>
<td>2021-05-29</td>
<td>Santa Clara</td>
<td>5307</td>
<td>419</td>
</tr>
<tr>
<td>2021-05-29</td>
<td>Solano</td>
<td>1580</td>
<td>140</td>
</tr>
<tr>
<td>2021-05-29</td>
<td>Sonoma</td>
<td>1410</td>
<td>126</td>
</tr>
</tbody>
</table>

657 rows x 4 columns

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:

```python
bayArea_unemployment_noPUA = bayArea_unemployment.drop(columns = ['PUA_Claims'])
bayArea_unemployment_noPUA
```
In [ ]:
#

Out[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alameda</td>
<td>1487</td>
</tr>
<tr>
<td>1</td>
<td>Contra Costa</td>
<td>1073</td>
</tr>
<tr>
<td>2</td>
<td>Marin</td>
<td>144</td>
</tr>
<tr>
<td>3</td>
<td>Napa</td>
<td>162</td>
</tr>
<tr>
<td>4</td>
<td>San Francisco</td>
<td>945</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>652</td>
<td>San Francisco</td>
<td>3878</td>
</tr>
<tr>
<td>653</td>
<td>San Mateo</td>
<td>2346</td>
</tr>
<tr>
<td>654</td>
<td>Santa Clara</td>
<td>5307</td>
</tr>
<tr>
<td>655</td>
<td>Solano</td>
<td>1580</td>
</tr>
<tr>
<td>656</td>
<td>Sonoma</td>
<td>1410</td>
</tr>
</tbody>
</table>

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 dataframe to a new dataframe:

```
bayArea_unemployment
```

In [ ]:
#

Out[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alameda</td>
<td>1487</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Marin</td>
<td>144</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>San Francisco</td>
<td>945</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>652</td>
<td>San Francisco</td>
<td>3878</td>
<td>233</td>
</tr>
<tr>
<td>653</td>
<td>San Mateo</td>
<td>2346</td>
<td>140</td>
</tr>
<tr>
<td>654</td>
<td>Santa Clara</td>
<td>5307</td>
<td>419</td>
</tr>
<tr>
<td>655</td>
<td>Solano</td>
<td>1580</td>
<td>140</td>
</tr>
<tr>
<td>656</td>
<td>Sonoma</td>
<td>1410</td>
<td>126</td>
</tr>
</tbody>
</table>

657 rows × 4 columns
Drop a row using an index

*First let’s check the syntax usage for .drop by typing...*

```python
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 .)

```python
In [ ]: #
Out[ ]:
```

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Marin</td>
<td>144</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>San Francisco</td>
<td>945</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>San Mateo</td>
<td>467</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>652</td>
<td>San Francisco</td>
<td>3878</td>
<td>233</td>
</tr>
<tr>
<td>653</td>
<td>San Mateo</td>
<td>2346</td>
<td>140</td>
</tr>
<tr>
<td>654</td>
<td>Santa Clara</td>
<td>5307</td>
<td>419</td>
</tr>
<tr>
<td>655</td>
<td>Solano</td>
<td>1580</td>
<td>140</td>
</tr>
<tr>
<td>656</td>
<td>Sonoma</td>
<td>1410</td>
<td>126</td>
</tr>
</tbody>
</table>

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]:

```python
bayArea_unemployment_drop4 = bayArea_unemployment.drop(index = [0,1,2,3])
bayArea_unemployment_drop4
```
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[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2020-01-11</td>
<td>San Francisco</td>
<td>945</td>
</tr>
<tr>
<td>5</td>
<td>2020-01-11</td>
<td>San Mateo</td>
<td>467</td>
</tr>
<tr>
<td>6</td>
<td>2020-01-11</td>
<td>Santa Clara</td>
<td>1443</td>
</tr>
<tr>
<td>7</td>
<td>2020-01-11</td>
<td>Solano</td>
<td>678</td>
</tr>
<tr>
<td>8</td>
<td>2020-01-11</td>
<td>Sonoma</td>
<td>558</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>652</td>
<td>2021-05-29</td>
<td>San Francisco</td>
<td>3878</td>
</tr>
<tr>
<td>653</td>
<td>2021-05-29</td>
<td>San Mateo</td>
<td>2346</td>
</tr>
<tr>
<td>654</td>
<td>2021-05-29</td>
<td>Santa Clara</td>
<td>5307</td>
</tr>
<tr>
<td>655</td>
<td>2021-05-29</td>
<td>Solano</td>
<td>1580</td>
</tr>
<tr>
<td>656</td>
<td>2021-05-29</td>
<td>Sonoma</td>
<td>1410</td>
</tr>
</tbody>
</table>

653 rows × 4 columns
```

```sql
# 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[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>2020-02-15</td>
<td>San Mateo</td>
<td>509</td>
</tr>
<tr>
<td>51</td>
<td>2020-02-15</td>
<td>Santa Clara</td>
<td>1395</td>
</tr>
<tr>
<td>52</td>
<td>2020-02-15</td>
<td>Solano</td>
<td>535</td>
</tr>
<tr>
<td>53</td>
<td>2020-02-15</td>
<td>Sonoma</td>
<td>411</td>
</tr>
<tr>
<td>54</td>
<td>2020-02-22</td>
<td>Alameda</td>
<td>1046</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>652</td>
<td>2021-05-29</td>
<td>San Francisco</td>
<td>3878</td>
</tr>
<tr>
<td>653</td>
<td>2021-05-29</td>
<td>San Mateo</td>
<td>2346</td>
</tr>
<tr>
<td>654</td>
<td>2021-05-29</td>
<td>Santa Clara</td>
<td>5307</td>
</tr>
<tr>
<td>655</td>
<td>2021-05-29</td>
<td>Solano</td>
<td>1580</td>
</tr>
<tr>
<td>656</td>
<td>2021-05-29</td>
<td>Sonoma</td>
<td>1410</td>
</tr>
</tbody>
</table>

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.

```python
region = "Bay Area"
bayArea_unemployment["Region Name"] = region
bayArea_unemployment
```

```
In [ ]: #
Out[ ]:  

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
<th>Region Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alameda</td>
<td>1487</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>1</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>2</td>
<td>Marin</td>
<td>144</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>3</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>4</td>
<td>San Francisco</td>
<td>945</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>652</td>
<td>San Francisco</td>
<td>3878</td>
<td>233</td>
<td>Bay Area</td>
</tr>
<tr>
<td>653</td>
<td>San Mateo</td>
<td>2346</td>
<td>140</td>
<td>Bay Area</td>
</tr>
<tr>
<td>654</td>
<td>Santa Clara</td>
<td>5307</td>
<td>419</td>
<td>Bay Area</td>
</tr>
<tr>
<td>655</td>
<td>Solano</td>
<td>1580</td>
<td>140</td>
<td>Bay Area</td>
</tr>
<tr>
<td>656</td>
<td>Sonoma</td>
<td>1410</td>
<td>126</td>
<td>Bay Area</td>
</tr>
</tbody>
</table>

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):

```python
state = "California"
bayArea_unemployment.insert(1, "State", state)
bayArea_unemployment
```
Insert another column called Country and set it to "United States". Put Country just after State.

In [ ]:

Out[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>Country</th>
<th>State</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
<th>Region Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2020-01-11</td>
<td>California</td>
<td>Alameda</td>
<td>1487</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>1</td>
<td>2020-01-11</td>
<td>California</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>2</td>
<td>2020-01-11</td>
<td>California</td>
<td>Marin</td>
<td>144</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>3</td>
<td>2020-01-11</td>
<td>California</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>4</td>
<td>2020-01-11</td>
<td>California</td>
<td>San Francisco</td>
<td>945</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>652</td>
<td>2021-05-29</td>
<td>California</td>
<td>San Francisco</td>
<td>3878</td>
<td>233</td>
<td>Bay Area</td>
</tr>
<tr>
<td>653</td>
<td>2021-05-29</td>
<td>California</td>
<td>San Mateo</td>
<td>2346</td>
<td>140</td>
<td>Bay Area</td>
</tr>
<tr>
<td>654</td>
<td>2021-05-29</td>
<td>California</td>
<td>Santa Clara</td>
<td>5307</td>
<td>419</td>
<td>Bay Area</td>
</tr>
<tr>
<td>655</td>
<td>2021-05-29</td>
<td>California</td>
<td>Solano</td>
<td>1580</td>
<td>140</td>
<td>Bay Area</td>
</tr>
<tr>
<td>656</td>
<td>2021-05-29</td>
<td>California</td>
<td>Sonoma</td>
<td>1410</td>
<td>126</td>
<td>Bay Area</td>
</tr>
</tbody>
</table>

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:

```python
          "County":"Marin", "UI_Claims": 450, "PUA_Claims":50, "Region Name":"Bay Area"}
```

Then check the usage of `.append` using either shift-tab or help. Note what it says about the usage and the first parameter `other`:

```markdown
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:

```python
bay_area_unemployment = bayArea_unemployment.append(new_row, ignore_index=True)
bay_area_unemployment
```
In [ ]:

```python
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[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>Country</th>
<th>State</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
<th>Region Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-01-11</td>
<td>United</td>
<td>California</td>
<td>Alameda</td>
<td>1487</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>2020-01-11</td>
<td>United</td>
<td>California</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>2020-01-11</td>
<td>United</td>
<td>California</td>
<td>Marin</td>
<td>144</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>2020-01-11</td>
<td>United</td>
<td>California</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>2020-01-11</td>
<td>United</td>
<td>California</td>
<td>San Francisco</td>
<td>945</td>
<td>0</td>
<td>Bay Area</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2021-05-29</td>
<td>United</td>
<td>California</td>
<td>San Mateo</td>
<td>2346</td>
<td>140</td>
<td>Bay Area</td>
</tr>
<tr>
<td>2021-05-29</td>
<td>United</td>
<td>California</td>
<td>Santa Clara</td>
<td>5307</td>
<td>419</td>
<td>Bay Area</td>
</tr>
<tr>
<td>2021-05-29</td>
<td>United</td>
<td>California</td>
<td>Solano</td>
<td>1580</td>
<td>140</td>
<td>Bay Area</td>
</tr>
<tr>
<td>2021-05-29</td>
<td>United</td>
<td>California</td>
<td>Sonoma</td>
<td>1410</td>
<td>126</td>
<td>Bay Area</td>
</tr>
<tr>
<td>2020-11-28</td>
<td>United</td>
<td>California</td>
<td>Marin</td>
<td>450</td>
<td>50</td>
<td>Bay Area</td>
</tr>
</tbody>
</table>

658 rows × 7 columns

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.
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:

```python
alameda = bayArea_unemployment["County"]=="Alameda"
alameda
```

Pass the filter to the larger dataframe to get only Alameda data returned:

```python
alameda_unemployment = bayArea_unemployment[alameda]
alameda_unemployment
```
Filtering using the `.eq()` method:

```python
alameda_unemployment = bayArea_unemployment[bayArea_unemployment.County.eq("Alameda")]
alameda_unemployment
```

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2020-01-11 Alameda</td>
<td>1487</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>2020-01-18 Alameda</td>
<td>1817</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>2020-01-25 Alameda</td>
<td>1350</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>2020-02-01 Alameda</td>
<td>1265</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>2020-02-08 Alameda</td>
<td>1328</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>612</td>
<td>2021-05-01 Alameda</td>
<td>5789</td>
<td>567</td>
</tr>
<tr>
<td>621</td>
<td>2021-05-08 Alameda</td>
<td>5544</td>
<td>580</td>
</tr>
<tr>
<td>630</td>
<td>2021-05-15 Alameda</td>
<td>5538</td>
<td>587</td>
</tr>
<tr>
<td>639</td>
<td>2021-05-22 Alameda</td>
<td>6892</td>
<td>507</td>
</tr>
<tr>
<td>648</td>
<td>2021-05-29 Alameda</td>
<td>5664</td>
<td>495</td>
</tr>
</tbody>
</table>

73 rows × 4 columns
Using either approach, create a san_francisco_unemployment

Be careful with spaces and underscores...

In [ ]:

Out[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2020-01-11</td>
<td>San Francisco</td>
<td>945</td>
</tr>
<tr>
<td>13</td>
<td>2020-01-18</td>
<td>San Francisco</td>
<td>965</td>
</tr>
<tr>
<td>22</td>
<td>2020-01-25</td>
<td>San Francisco</td>
<td>739</td>
</tr>
<tr>
<td>31</td>
<td>2020-02-01</td>
<td>San Francisco</td>
<td>801</td>
</tr>
<tr>
<td>40</td>
<td>2020-02-08</td>
<td>San Francisco</td>
<td>824</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>616</td>
<td>2021-05-01</td>
<td>San Francisco</td>
<td>4040</td>
</tr>
<tr>
<td>625</td>
<td>2021-05-08</td>
<td>San Francisco</td>
<td>3791</td>
</tr>
<tr>
<td>634</td>
<td>2021-05-15</td>
<td>San Francisco</td>
<td>3909</td>
</tr>
<tr>
<td>643</td>
<td>2021-05-22</td>
<td>San Francisco</td>
<td>4619</td>
</tr>
<tr>
<td>652</td>
<td>2021-05-29</td>
<td>San Francisco</td>
<td>3878</td>
</tr>
</tbody>
</table>

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)
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:

```python
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[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2020-01-11 Marin</td>
<td>144</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2020-01-11 San Francisco</td>
<td>945</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>2020-01-18 Marin</td>
<td>202</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>2020-01-18 San Francisco</td>
<td>965</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>2020-01-25 Marin</td>
<td>146</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>634</td>
<td>2021-05-15 San Francisco</td>
<td>3909</td>
<td>397</td>
</tr>
<tr>
<td>641</td>
<td>2021-05-22 Marin</td>
<td>812</td>
<td>79</td>
</tr>
<tr>
<td>643</td>
<td>2021-05-22 San Francisco</td>
<td>4619</td>
<td>281</td>
</tr>
<tr>
<td>650</td>
<td>2021-05-29 Marin</td>
<td>635</td>
<td>70</td>
</tr>
<tr>
<td>652</td>
<td>2021-05-29 San Francisco</td>
<td>3878</td>
<td>233</td>
</tr>
</tbody>
</table>

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
```
Create a dataframe from filtered data

Create a dataframe of the unemployment data that **does not** include Alameda County and Santa Clara.

```
In [ ]: #
Out[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alameda</td>
<td>1487</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>San Mateo</td>
<td>467</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Santa Clara</td>
<td>1443</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>651</td>
<td>Napa</td>
<td>392</td>
<td>42</td>
</tr>
<tr>
<td>653</td>
<td>San Mateo</td>
<td>2346</td>
<td>140</td>
</tr>
<tr>
<td>654</td>
<td>Santa Clara</td>
<td>5307</td>
<td>419</td>
</tr>
<tr>
<td>655</td>
<td>Solano</td>
<td>1580</td>
<td>140</td>
</tr>
<tr>
<td>656</td>
<td>Sonoma</td>
<td>1410</td>
<td>126</td>
</tr>
</tbody>
</table>

511 rows × 4 columns

```

```
In [ ]: #
Out[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Marín</td>
<td>144</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>San Francisco</td>
<td>945</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>San Mateo</td>
<td>467</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>651</td>
<td>Napa</td>
<td>392</td>
<td>42</td>
</tr>
<tr>
<td>652</td>
<td>San Francisco</td>
<td>3878</td>
<td>233</td>
</tr>
<tr>
<td>653</td>
<td>San Mateo</td>
<td>2346</td>
<td>140</td>
</tr>
<tr>
<td>655</td>
<td>Solano</td>
<td>1580</td>
<td>140</td>
</tr>
<tr>
<td>656</td>
<td>Sonoma</td>
<td>1410</td>
<td>126</td>
</tr>
</tbody>
</table>

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.

```python
In [ ]: #
Out[ ]:
```

<table>
<thead>
<tr>
<th>City</th>
<th>Elevation</th>
<th>Temperature</th>
<th>Latitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oroville</td>
<td>52</td>
<td>10.7</td>
<td>39.52</td>
</tr>
<tr>
<td>Auburn</td>
<td>394</td>
<td>9.7</td>
<td>38.91</td>
</tr>
<tr>
<td>Sonora</td>
<td>510</td>
<td>7.7</td>
<td>37.97</td>
</tr>
<tr>
<td>Placerville</td>
<td>564</td>
<td>9.2</td>
<td>38.70</td>
</tr>
<tr>
<td>Colfax</td>
<td>725</td>
<td>7.3</td>
<td>39.09</td>
</tr>
<tr>
<td>Nevada City</td>
<td>848</td>
<td>6.7</td>
<td>39.25</td>
</tr>
<tr>
<td>Quincy</td>
<td>1042</td>
<td>4.0</td>
<td>39.94</td>
</tr>
<tr>
<td>Yosemite</td>
<td>1225</td>
<td>5.0</td>
<td>37.75</td>
</tr>
<tr>
<td>Sierraville</td>
<td>1516</td>
<td>0.9</td>
<td>40.35</td>
</tr>
<tr>
<td>Truckee</td>
<td>1775</td>
<td>-1.1</td>
<td>39.33</td>
</tr>
<tr>
<td>Tahoe City</td>
<td>1899</td>
<td>-0.8</td>
<td>39.17</td>
</tr>
<tr>
<td>Bodie</td>
<td>2551</td>
<td>-4.4</td>
<td>38.21</td>
</tr>
</tbody>
</table>

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.)
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.

```python
sierraElevGroup = sierra.groupby("highElev").mean()
sierraElevGroup
```

```
<table>
<thead>
<tr>
<th>elevation</th>
<th>temperature</th>
<th>latitude</th>
<th>highElev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oroville</td>
<td>52</td>
<td>10.7</td>
<td>False</td>
</tr>
<tr>
<td>Auburn</td>
<td>394</td>
<td>9.7</td>
<td>False</td>
</tr>
<tr>
<td>Sonora</td>
<td>510</td>
<td>7.7</td>
<td>False</td>
</tr>
<tr>
<td>Placerville</td>
<td>564</td>
<td>9.2</td>
<td>False</td>
</tr>
<tr>
<td>Colfax</td>
<td>725</td>
<td>7.3</td>
<td>False</td>
</tr>
<tr>
<td>Nevada City</td>
<td>848</td>
<td>6.7</td>
<td>False</td>
</tr>
<tr>
<td>Quincy</td>
<td>1042</td>
<td>4.0</td>
<td>False</td>
</tr>
<tr>
<td>Yosemite</td>
<td>1225</td>
<td>5.0</td>
<td>True</td>
</tr>
<tr>
<td>Sierraville</td>
<td>1516</td>
<td>0.9</td>
<td>True</td>
</tr>
<tr>
<td>Truckee</td>
<td>1775</td>
<td>-1.1</td>
<td>True</td>
</tr>
<tr>
<td>Tahoe City</td>
<td>1899</td>
<td>-0.8</td>
<td>True</td>
</tr>
<tr>
<td>Bodie</td>
<td>2551</td>
<td>-4.4</td>
<td>True</td>
</tr>
</tbody>
</table>
```

Continuing with the bayArea_unemployment data

- Sorting a dataframe on a field:

```python
# sorting dataframe
sorted_unemployment = bayArea_unemployment.sort_values(by='UI_Claims', ascending=True)
sorted_unemployment
```

```
<table>
<thead>
<tr>
<th>elevation</th>
<th>temperature</th>
<th>latitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>515.5</td>
<td>8.55</td>
</tr>
<tr>
<td>True</td>
<td>1668.0</td>
<td>0.60</td>
</tr>
</tbody>
</table>
```
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:

```python
for index, row in sierra.iterrows():
    print(index)
    print(row)
```
<table>
<thead>
<tr>
<th>City</th>
<th>Elevation</th>
<th>Temperature</th>
<th>Latitude</th>
<th>HighElev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oroville</td>
<td>52</td>
<td>10.7</td>
<td>39.52</td>
<td>False</td>
</tr>
<tr>
<td>Auburn</td>
<td>394</td>
<td>9.7</td>
<td>38.91</td>
<td>False</td>
</tr>
<tr>
<td>Sonora</td>
<td>510</td>
<td>7.7</td>
<td>37.97</td>
<td>False</td>
</tr>
<tr>
<td>Placerville</td>
<td>564</td>
<td>9.2</td>
<td>38.7</td>
<td>False</td>
</tr>
<tr>
<td>Colfax</td>
<td>725</td>
<td>7.3</td>
<td>39.09</td>
<td>False</td>
</tr>
<tr>
<td>Nevada City</td>
<td>848</td>
<td>6.7</td>
<td>39.25</td>
<td>False</td>
</tr>
<tr>
<td>Quincy</td>
<td>1042</td>
<td>4.0</td>
<td>39.94</td>
<td>True</td>
</tr>
<tr>
<td>Yosemite</td>
<td>1225</td>
<td>5.0</td>
<td>37.75</td>
<td>True</td>
</tr>
<tr>
<td>Sierraville</td>
<td>1516</td>
<td>0.9</td>
<td>40.35</td>
<td>True</td>
</tr>
<tr>
<td>Truckee</td>
<td>1775</td>
<td>-1.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

```python
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 created 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.

```python
#groupby in a dataframe
unemployment_by_county = bayArea_unemployment.groupby("County")
unemployment_by_county.groups
```
In [ ]:

In [ ]:

Out[ ]:

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?
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`.

We can also combine the operations to go directly from the unemployment data to the sum:

```python
bayArea_unemployment.groupby("County").sum()
```
... then we can sort to see the county with the most unemployment claims in order:

```python
sorted_unemployment_by_county = unemployment_by_county.sort_values(by='UI_Claims', ascending=False)
sorted_unemployment_by_county
```

In [ ]: #

Out[ ]:

<table>
<thead>
<tr>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alameda</td>
<td>740263</td>
<td>170528</td>
</tr>
<tr>
<td>Contra Costa</td>
<td>503377</td>
<td>111380</td>
</tr>
<tr>
<td>Marin</td>
<td>87088</td>
<td>18838</td>
</tr>
<tr>
<td>Napa</td>
<td>64668</td>
<td>9219</td>
</tr>
<tr>
<td>San Francisco</td>
<td>416169</td>
<td>84599</td>
</tr>
<tr>
<td>San Mateo</td>
<td>282089</td>
<td>43713</td>
</tr>
<tr>
<td>Santa Clara</td>
<td>691576</td>
<td>116923</td>
</tr>
<tr>
<td>Solano</td>
<td>217828</td>
<td>48366</td>
</tr>
<tr>
<td>Sonoma</td>
<td>207900</td>
<td>36821</td>
</tr>
</tbody>
</table>

... then we can sort to see the county with the most unemployment claims in order:

```python
sorted_unemployment_by_county = unemployment_by_county.sort_values(by='UI_Claims', ascending=False)
sorted_unemployment_by_county
```

In [ ]: #

Out[ ]:

<table>
<thead>
<tr>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alameda</td>
<td>740263</td>
<td>170528</td>
</tr>
<tr>
<td>Santa Clara</td>
<td>691576</td>
<td>116923</td>
</tr>
<tr>
<td>Contra Costa</td>
<td>503377</td>
<td>111380</td>
</tr>
<tr>
<td>San Francisco</td>
<td>416169</td>
<td>84599</td>
</tr>
<tr>
<td>San Mateo</td>
<td>282089</td>
<td>43713</td>
</tr>
<tr>
<td>Solano</td>
<td>217828</td>
<td>48366</td>
</tr>
<tr>
<td>Sonoma</td>
<td>207900</td>
<td>36821</td>
</tr>
<tr>
<td>Marin</td>
<td>87088</td>
<td>18838</td>
</tr>
<tr>
<td>Napa</td>
<td>64668</td>
<td>9219</td>
</tr>
</tbody>
</table>
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[ ]: 

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
<th>PUA_UI_DIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alameda</td>
<td>1487</td>
<td>0</td>
<td>1487</td>
</tr>
<tr>
<td>1</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
<td>1073</td>
</tr>
<tr>
<td>2</td>
<td>Marin</td>
<td>144</td>
<td>0</td>
<td>144</td>
</tr>
<tr>
<td>3</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
<td>162</td>
</tr>
<tr>
<td>4</td>
<td>San Francisco</td>
<td>945</td>
<td>0</td>
<td>945</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>652</td>
<td>San Francisco</td>
<td>3878</td>
<td>233</td>
<td>3645</td>
</tr>
<tr>
<td>653</td>
<td>San Mateo</td>
<td>2346</td>
<td>140</td>
<td>2206</td>
</tr>
<tr>
<td>654</td>
<td>Santa Clara</td>
<td>5307</td>
<td>419</td>
<td>4888</td>
</tr>
<tr>
<td>655</td>
<td>Solano</td>
<td>1580</td>
<td>140</td>
<td>1440</td>
</tr>
<tr>
<td>656</td>
<td>Sonoma</td>
<td>1410</td>
<td>126</td>
<td>1284</td>
</tr>
</tbody>
</table>
```

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[ ]:

<table>
<thead>
<tr>
<th>Week_Ending</th>
<th>County</th>
<th>UI_Claims</th>
<th>PUA_Claims</th>
<th>has_pua_claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alameda</td>
<td>1487</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>Contra Costa</td>
<td>1073</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>Marin</td>
<td>144</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>Napa</td>
<td>162</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>San Francisco</td>
<td>945</td>
<td>0</td>
<td>False</td>
</tr>
</tbody>
</table>

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:

```python
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")
```

Have a look at the dataframe of Americas population from 2000 to 2009.
In [ ]: #
<table>
<thead>
<tr>
<th>Name</th>
<th>Region</th>
<th>2000</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mexico</td>
<td>North America</td>
<td>98899845</td>
<td>100298153</td>
<td>101684758</td>
<td>103081020</td>
<td>104514932</td>
<td>106005203</td>
<td>10756C</td>
</tr>
<tr>
<td>Canada</td>
<td>North America</td>
<td>30588383</td>
<td>30880073</td>
<td>31178263</td>
<td>31488048</td>
<td>31815494</td>
<td>32164309</td>
<td>32536</td>
</tr>
<tr>
<td>Colombia</td>
<td>South America</td>
<td>39629968</td>
<td>40255967</td>
<td>40875360</td>
<td>41483869</td>
<td>42075955</td>
<td>42647723</td>
<td>4320C</td>
</tr>
<tr>
<td>Venezuela</td>
<td>South America</td>
<td>24192446</td>
<td>24646472</td>
<td>25100408</td>
<td>25551624</td>
<td>25996594</td>
<td>26432447</td>
<td>2685C</td>
</tr>
<tr>
<td>Costa Rica</td>
<td>Central America</td>
<td>3962372</td>
<td>4034074</td>
<td>4100925</td>
<td>4164053</td>
<td>4225155</td>
<td>4285502</td>
<td>4345</td>
</tr>
<tr>
<td>Guatemala</td>
<td>Central America</td>
<td>11650743</td>
<td>11924946</td>
<td>12208848</td>
<td>12500478</td>
<td>12796925</td>
<td>13096028</td>
<td>13397</td>
</tr>
<tr>
<td>Brazil</td>
<td>South America</td>
<td>174790340</td>
<td>177196054</td>
<td>179537520</td>
<td>181809246</td>
<td>184006481</td>
<td>186127103</td>
<td>188167</td>
</tr>
<tr>
<td>Argentina</td>
<td>South America</td>
<td>36870787</td>
<td>37275652</td>
<td>37681749</td>
<td>38087868</td>
<td>38491972</td>
<td>38892931</td>
<td>39285</td>
</tr>
<tr>
<td>Chile</td>
<td>South America</td>
<td>15342353</td>
<td>15516113</td>
<td>15684409</td>
<td>15849652</td>
<td>16014971</td>
<td>16182721</td>
<td>16354</td>
</tr>
<tr>
<td>Belize</td>
<td>Central America</td>
<td>247315</td>
<td>255063</td>
<td>262378</td>
<td>269425</td>
<td>276504</td>
<td>283800</td>
<td>291</td>
</tr>
<tr>
<td>El Salvador</td>
<td>Central America</td>
<td>5887936</td>
<td>5927006</td>
<td>5962136</td>
<td>5994077</td>
<td>6023797</td>
<td>6052123</td>
<td>6075</td>
</tr>
<tr>
<td>Honduras</td>
<td>Central America</td>
<td>6574509</td>
<td>6751912</td>
<td>6929265</td>
<td>7106319</td>
<td>7282953</td>
<td>7458985</td>
<td>7634</td>
</tr>
<tr>
<td>Nicaragua</td>
<td>Central America</td>
<td>5069302</td>
<td>5145366</td>
<td>5219328</td>
<td>5292118</td>
<td>5364935</td>
<td>5438690</td>
<td>5513</td>
</tr>
<tr>
<td>Panama</td>
<td>Central America</td>
<td>3030328</td>
<td>3089648</td>
<td>3149188</td>
<td>3209048</td>
<td>3269356</td>
<td>3330217</td>
<td>3391</td>
</tr>
<tr>
<td>Bolivia</td>
<td>South America</td>
<td>8418264</td>
<td>8580235</td>
<td>8742814</td>
<td>8905823</td>
<td>9069039</td>
<td>9232306</td>
<td>9395</td>
</tr>
<tr>
<td>Ecuador</td>
<td>South America</td>
<td>12681123</td>
<td>12914667</td>
<td>13143465</td>
<td>13369678</td>
<td>13596388</td>
<td>13825847</td>
<td>14055</td>
</tr>
<tr>
<td>Guyana</td>
<td>South America</td>
<td>746715</td>
<td>745206</td>
<td>744789</td>
<td>745143</td>
<td>745737</td>
<td>746163</td>
<td>746</td>
</tr>
<tr>
<td>Paraguay</td>
<td>South America</td>
<td>5323201</td>
<td>5428444</td>
<td>5531962</td>
<td>5632983</td>
<td>5730549</td>
<td>5824096</td>
<td>5913</td>
</tr>
<tr>
<td>Peru</td>
<td>South America</td>
<td>26459944</td>
<td>26799285</td>
<td>27100968</td>
<td>27372226</td>
<td>27624213</td>
<td>27866145</td>
<td>28102</td>
</tr>
<tr>
<td>Suriname</td>
<td>South America</td>
<td>470949</td>
<td>476579</td>
<td>482235</td>
<td>487942</td>
<td>493679</td>
<td>499464</td>
<td>505</td>
</tr>
<tr>
<td>Uruguay</td>
<td>South America</td>
<td>3319736</td>
<td>3325473</td>
<td>3326040</td>
<td>3326368</td>
<td>3321476</td>
<td>3321803</td>
<td>3325</td>
</tr>
<tr>
<td>Venezuela</td>
<td>South America</td>
<td>24192446</td>
<td>24646472</td>
<td>25100408</td>
<td>25551624</td>
<td>25996594</td>
<td>26432447</td>
<td>2685C</td>
</tr>
</tbody>
</table>
To see the datatypes of each field in your dataframe:

```python
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 dataframe. 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.

```python
In [ ]: #
              '2007', '2008', '2009'],
dtype='object')
```

```python
In [ ]: #
dtype='object', name='Name')
```

Let's make a dataframe that just contains the south american populations from 2000 - 2009

```python
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
```
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[ ]:

<table>
<thead>
<tr>
<th>Region</th>
<th>2000</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colombia</td>
<td>39629968</td>
<td>40255967</td>
<td>40875360</td>
<td>41483869</td>
<td>42075955</td>
<td>42647723</td>
<td>43200771</td>
</tr>
<tr>
<td>Venezuela</td>
<td>24192446</td>
<td>24646472</td>
<td>25100408</td>
<td>25551624</td>
<td>25996594</td>
<td>26432447</td>
<td>26850000</td>
</tr>
<tr>
<td>Brazil</td>
<td>174790340</td>
<td>177196054</td>
<td>179537520</td>
<td>181809246</td>
<td>184006481</td>
<td>186127103</td>
<td>18816730</td>
</tr>
<tr>
<td>Argentina</td>
<td>36870787</td>
<td>37275652</td>
<td>37681749</td>
<td>38087868</td>
<td>38491972</td>
<td>38892931</td>
<td>39289845</td>
</tr>
<tr>
<td>Chile</td>
<td>15342353</td>
<td>15516113</td>
<td>15684409</td>
<td>15849652</td>
<td>16014971</td>
<td>16182721</td>
<td>16354790</td>
</tr>
<tr>
<td>Bolivia</td>
<td>8418264</td>
<td>8580235</td>
<td>8742814</td>
<td>8905823</td>
<td>9069039</td>
<td>9232306</td>
<td>93957976</td>
</tr>
<tr>
<td>Ecuador</td>
<td>12681123</td>
<td>12914667</td>
<td>13143465</td>
<td>13369678</td>
<td>13569388</td>
<td>13825847</td>
<td>14059120</td>
</tr>
<tr>
<td>Guyana</td>
<td>746715</td>
<td>745206</td>
<td>744789</td>
<td>745143</td>
<td>745737</td>
<td>746163</td>
<td>7465867</td>
</tr>
<tr>
<td>Paraguay</td>
<td>5323201</td>
<td>5428444</td>
<td>5531962</td>
<td>5632983</td>
<td>5730549</td>
<td>5824096</td>
<td>5913459</td>
</tr>
<tr>
<td>Peru</td>
<td>26459944</td>
<td>26799285</td>
<td>27100968</td>
<td>27372226</td>
<td>27624213</td>
<td>27866145</td>
<td>28102343</td>
</tr>
<tr>
<td>Suriname</td>
<td>470949</td>
<td>476579</td>
<td>482235</td>
<td>487942</td>
<td>493679</td>
<td>499464</td>
<td>50547861</td>
</tr>
<tr>
<td>Uruguay</td>
<td>3319736</td>
<td>3325473</td>
<td>3326040</td>
<td>3323668</td>
<td>3321476</td>
<td>3321803</td>
<td>33254730</td>
</tr>
<tr>
<td>Venezuela</td>
<td>24192446</td>
<td>24646472</td>
<td>25100408</td>
<td>25551624</td>
<td>25996594</td>
<td>26432447</td>
<td>26850000</td>
</tr>
</tbody>
</table>
Next we're going to create a new dataframe for the purpose of plotting our data:

```python
south_america_forplot = south_america_00_09.drop(columns=['Region'])
south_america_forplot
```

```
In [ ]: #
      Name
Colombia  39629968  40255967  40875360  41483869  42075955  42647723  43200897  437
Venezuela  24192446  24646472  25100408  25551624  25996594  26432447  26850194  272
Brazil    174790340  177196054  179537520  181809246  184006481  186127103  188167356  1901
Argentina  36870787  37275652  37681749  38087868  38491972  38892931  39289878  396
Chile      15342353  15516113  15684409  15849652  16014971  16182721  16354504  165
Bolivia    8418264   8580235   8742814   8905823   9069039   9232306   9395446   95
Ecuador    12681123  12914667  13143465  13369678  13596388  13825847  14059384  142
Guyana     746715    745206    744789    745143    745737    746163    746343   7
Paraguay   5323201   5428444   5531962   5632983   5730549   5824096   5913209   59
Peru       26459944  26799285  27100968  27372226  27624213  27866145  28102056  283
Suriname   470949    476579    482235    487942    493679    499464    505295   £
Uruguay    3319736   3325473   3326040   3323668   3321476   3321803   3325401   33
Venezuela  24192446  24646472  25100408  25551624  25996594  26432447  26850194  272
```

Now just call the `.plot()` method against the data.

```python
south_america_forplot.plot()
```
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 tranpose the data:

```python
south_america_forplot.transpose().plot(title='South America Population 2000 - 2009', figsize=(20,15))
```
Create plot

Now create the same plot that we did above, but this time create it with the data from Central America.
In [ ]: #

Out[ ]: <AxesSubplot: xlabel='Name'>
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:

```python
north_america_00_09 = americas_pop_00_09[americas_pop_00_09.Region.eq("North America")]
north_america_00_09
```
Let's take a look at the US Population data now:

```python
us_pop_00_09
```
In [ ]: #
<table>
<thead>
<tr>
<th>Name</th>
<th>Region</th>
<th>2000</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>North America</td>
<td>4452173</td>
<td>4467634</td>
<td>4480089</td>
<td>4503491</td>
<td>4530729</td>
<td>4569805</td>
<td>4628981</td>
</tr>
<tr>
<td>Alaska</td>
<td>North America</td>
<td>627963</td>
<td>633714</td>
<td>642337</td>
<td>648414</td>
<td>659286</td>
<td>666946</td>
<td>675302</td>
</tr>
<tr>
<td>Arizona</td>
<td>North America</td>
<td>5160586</td>
<td>5273477</td>
<td>5396255</td>
<td>5510364</td>
<td>5652404</td>
<td>5839077</td>
<td>6029141</td>
</tr>
<tr>
<td>Arkansas</td>
<td>North America</td>
<td>2678588</td>
<td>2691571</td>
<td>2705927</td>
<td>2724816</td>
<td>2749686</td>
<td>2781097</td>
<td>2821761</td>
</tr>
<tr>
<td>California</td>
<td>North America</td>
<td>33987977</td>
<td>34479458</td>
<td>34871843</td>
<td>35253159</td>
<td>35574576</td>
<td>35827943</td>
<td>36021202</td>
</tr>
<tr>
<td>Colorado</td>
<td>North America</td>
<td>4326921</td>
<td>4425687</td>
<td>4490406</td>
<td>4528732</td>
<td>4575013</td>
<td>4631888</td>
<td>4720423</td>
</tr>
<tr>
<td>Connecticut</td>
<td>North America</td>
<td>3411777</td>
<td>3432835</td>
<td>3458749</td>
<td>3484336</td>
<td>3496094</td>
<td>3506956</td>
<td>3517460</td>
</tr>
<tr>
<td>Delaware</td>
<td>North America</td>
<td>786373</td>
<td>795699</td>
<td>806169</td>
<td>818003</td>
<td>830803</td>
<td>845150</td>
<td>859268</td>
</tr>
<tr>
<td>District of</td>
<td>North America</td>
<td>572046</td>
<td>574504</td>
<td>573158</td>
<td>568502</td>
<td>567754</td>
<td>567136</td>
<td>570681</td>
</tr>
<tr>
<td>Columbia</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Florida</td>
<td>North America</td>
<td>16047515</td>
<td>16356966</td>
<td>16689370</td>
<td>17004085</td>
<td>17415318</td>
<td>17842038</td>
<td>18166990</td>
</tr>
<tr>
<td>Georgia</td>
<td>North America</td>
<td>8227303</td>
<td>8377038</td>
<td>8508256</td>
<td>8622793</td>
<td>8769252</td>
<td>8925922</td>
<td>9155813</td>
</tr>
<tr>
<td>Hawaii</td>
<td>North America</td>
<td>1213519</td>
<td>1225948</td>
<td>1239613</td>
<td>1251154</td>
<td>1273569</td>
<td>1292729</td>
<td>1309731</td>
</tr>
<tr>
<td>Idaho</td>
<td>North America</td>
<td>1299430</td>
<td>1319962</td>
<td>1340372</td>
<td>1363380</td>
<td>1391802</td>
<td>1428241</td>
<td>1468669</td>
</tr>
<tr>
<td>Illinois</td>
<td>North America</td>
<td>12434161</td>
<td>12488445</td>
<td>12525556</td>
<td>12556006</td>
<td>12589773</td>
<td>12609903</td>
<td>12643955</td>
</tr>
<tr>
<td>Indiana</td>
<td>North America</td>
<td>6091866</td>
<td>6127760</td>
<td>6155967</td>
<td>6196638</td>
<td>6233007</td>
<td>6278616</td>
<td>6332669</td>
</tr>
<tr>
<td>Iowa</td>
<td>North America</td>
<td>2929067</td>
<td>2931997</td>
<td>2934234</td>
<td>2941999</td>
<td>2953635</td>
<td>2964454</td>
<td>2982644</td>
</tr>
<tr>
<td>Kansas</td>
<td>North America</td>
<td>2693681</td>
<td>2702162</td>
<td>2713535</td>
<td>2723004</td>
<td>2734373</td>
<td>2745299</td>
<td>2762931</td>
</tr>
<tr>
<td>Kentucky</td>
<td>North America</td>
<td>4049021</td>
<td>4068132</td>
<td>4089875</td>
<td>4117170</td>
<td>4146101</td>
<td>4182742</td>
<td>4219239</td>
</tr>
<tr>
<td>Louisiana</td>
<td>North America</td>
<td>4471885</td>
<td>4477875</td>
<td>4497267</td>
<td>4521042</td>
<td>4552238</td>
<td>4576628</td>
<td>4302665</td>
</tr>
<tr>
<td>Maine</td>
<td>North America</td>
<td>1277072</td>
<td>1285692</td>
<td>1295960</td>
<td>1306513</td>
<td>1313688</td>
<td>1318787</td>
<td>1323619</td>
</tr>
<tr>
<td>Maryland</td>
<td>North America</td>
<td>5311034</td>
<td>5374691</td>
<td>5440389</td>
<td>5496269</td>
<td>5546935</td>
<td>5592379</td>
<td>5627367</td>
</tr>
<tr>
<td>Massachusetts</td>
<td>North America</td>
<td>6361104</td>
<td>6397634</td>
<td>6417206</td>
<td>6422565</td>
<td>6412281</td>
<td>6403290</td>
<td>6410084</td>
</tr>
<tr>
<td>Region</td>
<td>Name</td>
<td>2000</td>
<td>2001</td>
<td>2002</td>
<td>2003</td>
<td>2004</td>
<td>2005</td>
<td>2006</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Michigan</td>
<td>North America</td>
<td>9952450</td>
<td>9991120</td>
<td>10015710</td>
<td>10041152</td>
<td>1005315</td>
<td>10051137</td>
<td>10036081</td>
</tr>
<tr>
<td>Minnesota</td>
<td>North America</td>
<td>4933692</td>
<td>4982796</td>
<td>5018935</td>
<td>5053572</td>
<td>5087713</td>
<td>5119598</td>
<td>5163555</td>
</tr>
<tr>
<td>Mississippi</td>
<td>North America</td>
<td>2848353</td>
<td>2852994</td>
<td>2858681</td>
<td>2868312</td>
<td>2889010</td>
<td>2905943</td>
<td>2904978</td>
</tr>
<tr>
<td>Missouri</td>
<td>North America</td>
<td>5607285</td>
<td>5641142</td>
<td>5674825</td>
<td>5709403</td>
<td>5747713</td>
<td>5790300</td>
<td>5842704</td>
</tr>
<tr>
<td>Montana</td>
<td>North America</td>
<td>903773</td>
<td>906961</td>
<td>911667</td>
<td>919630</td>
<td>930009</td>
<td>940102</td>
<td>952692</td>
</tr>
<tr>
<td>Nebraska</td>
<td>North America</td>
<td>1713820</td>
<td>1719836</td>
<td>1728292</td>
<td>1738643</td>
<td>1749370</td>
<td>1761497</td>
<td>1772693</td>
</tr>
<tr>
<td>Nevada</td>
<td>North America</td>
<td>2018741</td>
<td>2098399</td>
<td>2173791</td>
<td>2248850</td>
<td>2346222</td>
<td>2432143</td>
<td>2522658</td>
</tr>
<tr>
<td>New Hampshire</td>
<td>North America</td>
<td>1239882</td>
<td>1255517</td>
<td>1269089</td>
<td>1279840</td>
<td>1290121</td>
<td>1298492</td>
<td>1308389</td>
</tr>
<tr>
<td>New Jersey</td>
<td>North America</td>
<td>8430621</td>
<td>8492671</td>
<td>8552643</td>
<td>8601402</td>
<td>8634561</td>
<td>8651974</td>
<td>8661679</td>
</tr>
<tr>
<td>New Mexico</td>
<td>North America</td>
<td>1821204</td>
<td>1831690</td>
<td>1855309</td>
<td>1877574</td>
<td>1903808</td>
<td>1932274</td>
<td>1962137</td>
</tr>
<tr>
<td>New York</td>
<td>North America</td>
<td>19001780</td>
<td>1908238</td>
<td>19137800</td>
<td>1917593</td>
<td>19171567</td>
<td>19132610</td>
<td>19104631</td>
</tr>
<tr>
<td>North Carolina</td>
<td>North America</td>
<td>8081614</td>
<td>8210122</td>
<td>8326201</td>
<td>8422501</td>
<td>8553152</td>
<td>8705407</td>
<td>8917270</td>
</tr>
<tr>
<td>North Dakota</td>
<td>North America</td>
<td>642023</td>
<td>639062</td>
<td>638168</td>
<td>638817</td>
<td>644705</td>
<td>646089</td>
<td>649422</td>
</tr>
<tr>
<td>Ohio</td>
<td>North America</td>
<td>1136543</td>
<td>11387404</td>
<td>11407889</td>
<td>11434788</td>
<td>11452251</td>
<td>11463320</td>
<td>11481213</td>
</tr>
<tr>
<td>Oklahoma</td>
<td>North America</td>
<td>3454365</td>
<td>3467100</td>
<td>3489080</td>
<td>3504892</td>
<td>3525233</td>
<td>3548597</td>
<td>3594090</td>
</tr>
<tr>
<td>Oregon</td>
<td>North America</td>
<td>3429708</td>
<td>3467937</td>
<td>3513424</td>
<td>3547376</td>
<td>3569463</td>
<td>3613202</td>
<td>3670883</td>
</tr>
<tr>
<td>Pennsylvania</td>
<td>North America</td>
<td>12284173</td>
<td>12298970</td>
<td>12331031</td>
<td>12374658</td>
<td>12410722</td>
<td>12449990</td>
<td>12510809</td>
</tr>
<tr>
<td>Rhode Island</td>
<td>North America</td>
<td>1050268</td>
<td>1057142</td>
<td>1065995</td>
<td>1071342</td>
<td>1074579</td>
<td>1067916</td>
<td>1063096</td>
</tr>
<tr>
<td>South Carolina</td>
<td>North America</td>
<td>4024223</td>
<td>4064995</td>
<td>4107795</td>
<td>4150297</td>
<td>4210921</td>
<td>4270150</td>
<td>4357847</td>
</tr>
<tr>
<td>South Dakota</td>
<td>North America</td>
<td>755844</td>
<td>757972</td>
<td>760020</td>
<td>763729</td>
<td>770396</td>
<td>775493</td>
<td>783033</td>
</tr>
<tr>
<td>Tennessee</td>
<td>North America</td>
<td>5703719</td>
<td>5750789</td>
<td>5795918</td>
<td>5847812</td>
<td>5910809</td>
<td>5991057</td>
<td>6088766</td>
</tr>
<tr>
<td>Texas</td>
<td>North America</td>
<td>2094499</td>
<td>21319622</td>
<td>21690325</td>
<td>22030931</td>
<td>22390423</td>
<td>22778123</td>
<td>23359580</td>
</tr>
<tr>
<td>Utah</td>
<td>North America</td>
<td>2244502</td>
<td>2283715</td>
<td>2324815</td>
<td>2360137</td>
<td>2401580</td>
<td>2457719</td>
<td>2525507</td>
</tr>
</tbody>
</table>
We can use the group by function to group all data in "us_pop_00_09"

```python
all_us_00_09 = us_pop_00_09.groupby("Region", as_index = False).sum()
all_us_00_09
```

```
In [39]:
Out[39]:
    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".

```python
all_us_00_09.set_index("Region")
```

```
In [40]:
Out[40]:
   Region
North America    282162411  284968955  287625193  290107933  292805298  295516599  298379912
```
Set index on north america data

```python
north_america_00_09.set_index("Region")
```

```
In [ ]: #
Out[ ]:

<table>
<thead>
<tr>
<th>Region</th>
<th>2000</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>98899845</td>
<td>100298153</td>
<td>101684758</td>
<td>103081020</td>
<td>104514932</td>
<td>106005203</td>
<td>107560153</td>
</tr>
<tr>
<td>North America</td>
<td>30588383</td>
<td>30880073</td>
<td>31178263</td>
<td>31488048</td>
<td>31815494</td>
<td>32164309</td>
<td>32536987</td>
</tr>
</tbody>
</table>

Now we can concat the two datasets to get Canada, Mexico and the US:

```python
all_nAmerica = pd.concat([north_america_00_09, all_us_00_09])
all_nAmerica.set_index("Region")
```

```
In [ ]: #
Out[ ]:

<table>
<thead>
<tr>
<th>Region</th>
<th>2000</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>98899845</td>
<td>100298153</td>
<td>101684758</td>
<td>103081020</td>
<td>104514932</td>
<td>106005203</td>
<td>107560153</td>
</tr>
<tr>
<td>North America</td>
<td>30588383</td>
<td>30880073</td>
<td>31178263</td>
<td>31488048</td>
<td>31815494</td>
<td>32164309</td>
<td>32536987</td>
</tr>
<tr>
<td>North America</td>
<td>28216241</td>
<td>284968955</td>
<td>287625193</td>
<td>290107933</td>
<td>292805298</td>
<td>295516599</td>
<td>298379912</td>
</tr>
</tbody>
</table>

Then do a final sum to get all population in North America:

```python
total_north_america_pop = all_nAmerica.groupby("Region", as_index = False).sum()
total_north_america_pop
```

```
In [ ]: 
Out[ ]:

<table>
<thead>
<tr>
<th>Region</th>
<th>2000</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>North America</td>
<td>411650639</td>
<td>416147181</td>
<td>420488214</td>
<td>424677001</td>
<td>429135724</td>
<td>433686111</td>
<td>438477052</td>
</tr>
</tbody>
</table>
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:

```python
all_00_09 = pd.concat([americas_pop_00_09, us_pop_00_09])
all_00_09
```

Then we'll concat data tables for the same regions from 2010 to 2019:

```python
all_10_19 = pd.concat([americas_pop_10_19, us_pop_10_19])
all_10_19
```
In [ ]:
#

Out [ ]:

<table>
<thead>
<tr>
<th>Region</th>
<th>2010</th>
<th>2011</th>
<th>2012</th>
<th>2013</th>
<th>2014</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mexico</strong></td>
<td>North America</td>
<td>114092963</td>
<td>115695473</td>
<td>117274155</td>
<td>118827161</td>
<td>120355128</td>
</tr>
<tr>
<td><strong>Canada</strong></td>
<td>North America</td>
<td>34147564</td>
<td>34539159</td>
<td>34922030</td>
<td>35296528</td>
<td>35664337</td>
</tr>
<tr>
<td><strong>Colombia</strong></td>
<td>South America</td>
<td>45222700</td>
<td>45662748</td>
<td>46075718</td>
<td>46495493</td>
<td>46967696</td>
</tr>
<tr>
<td><strong>Venezuela</strong></td>
<td>South America</td>
<td>28439940</td>
<td>28887874</td>
<td>29360837</td>
<td>29781040</td>
<td>30042968</td>
</tr>
<tr>
<td><strong>Costa Rica</strong></td>
<td>Central America</td>
<td>4577378</td>
<td>4633086</td>
<td>4688000</td>
<td>4742107</td>
<td>4795396</td>
</tr>
<tr>
<td><strong>Virginia</strong></td>
<td>North America</td>
<td>8023699</td>
<td>8101155</td>
<td>8185080</td>
<td>8252427</td>
<td>8310993</td>
</tr>
<tr>
<td><strong>Washington</strong></td>
<td>North America</td>
<td>6742830</td>
<td>6826627</td>
<td>6897058</td>
<td>6963985</td>
<td>7054655</td>
</tr>
<tr>
<td><strong>West Virginia</strong></td>
<td>North America</td>
<td>1854239</td>
<td>1856301</td>
<td>1856872</td>
<td>1853914</td>
<td>1849489</td>
</tr>
<tr>
<td><strong>Wisconsin</strong></td>
<td>North America</td>
<td>5690475</td>
<td>5705288</td>
<td>5719960</td>
<td>5736754</td>
<td>5751525</td>
</tr>
<tr>
<td><strong>Wyoming</strong></td>
<td>North America</td>
<td>564487</td>
<td>567299</td>
<td>576305</td>
<td>582122</td>
<td>582531</td>
</tr>
</tbody>
</table>

73 rows × 11 columns

Now that we have these two new datasets, we can run a pd.merge() operation, which has the usage:

```python
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.

```python
merged_df = pd.merge(all_00_09, all_10_19, left_index=True, right_index=True, how='right')
merged_df
```
In [ ]:

```
#
```

Out[ ]:

```
#### Region_x  2000  2001  2002  2003  2004  2005  2006
Name

<table>
<thead>
<tr>
<th>Name</th>
<th>Region_x</th>
<th>2000</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>North America</td>
<td>4452173</td>
<td>4467634</td>
<td>4480089</td>
<td>4503491</td>
<td>4530729</td>
<td>4569805</td>
<td>4629881</td>
</tr>
<tr>
<td>Alaska</td>
<td>North America</td>
<td>627963</td>
<td>633714</td>
<td>642337</td>
<td>648414</td>
<td>659286</td>
<td>666946</td>
<td>675302</td>
</tr>
<tr>
<td>Argentina</td>
<td>South America</td>
<td>36870787</td>
<td>37275652</td>
<td>37681749</td>
<td>38087868</td>
<td>38491972</td>
<td>38892931</td>
<td>39289878</td>
</tr>
<tr>
<td>Arizona</td>
<td>North America</td>
<td>5160586</td>
<td>5273477</td>
<td>5396255</td>
<td>5510364</td>
<td>5652404</td>
<td>5839077</td>
<td>6029141</td>
</tr>
<tr>
<td>Arkansas</td>
<td>North America</td>
<td>2678588</td>
<td>2691571</td>
<td>2705927</td>
<td>2724816</td>
<td>2749686</td>
<td>2781097</td>
<td>2821761</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Virginia</td>
<td>North America</td>
<td>7105817</td>
<td>7198362</td>
<td>7286873</td>
<td>7369977</td>
<td>7475575</td>
<td>7577105</td>
<td>7673725</td>
</tr>
<tr>
<td>Washington</td>
<td>North America</td>
<td>5910512</td>
<td>5985722</td>
<td>6052349</td>
<td>6104115</td>
<td>6178645</td>
<td>6257305</td>
<td>6370753</td>
</tr>
<tr>
<td>West Virginia</td>
<td>North America</td>
<td>1807021</td>
<td>1801481</td>
<td>1805414</td>
<td>1812295</td>
<td>1816438</td>
<td>1820492</td>
<td>1827912</td>
</tr>
<tr>
<td>Wisconsin</td>
<td>North America</td>
<td>5373999</td>
<td>5406835</td>
<td>5445162</td>
<td>5479203</td>
<td>5514026</td>
<td>5546166</td>
<td>5577655</td>
</tr>
<tr>
<td>Wyoming</td>
<td>North America</td>
<td>494300</td>
<td>494657</td>
<td>500017</td>
<td>503453</td>
<td>509106</td>
<td>514157</td>
<td>522667</td>
</tr>
</tbody>
</table>
```

75 rows x 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
```
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.
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.

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

First, import the pandas library as pd and matplotlib.pyplot as plt

```python
import pandas as pd
import matplotlib.pyplot as plt
```

```python
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) 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[ ]:
```

<table>
<thead>
<tr>
<th></th>
<th>iso_code</th>
<th>continent</th>
<th>location</th>
<th>date</th>
<th>total_cases</th>
<th>new_cases</th>
<th>new_cases_smoothed</th>
<th>tc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AFG</td>
<td>Asia</td>
<td>Afghanistan</td>
<td>2020-02-24</td>
<td>5.0</td>
<td>5.0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>AFG</td>
<td>Asia</td>
<td>Afghanistan</td>
<td>2020-02-25</td>
<td>5.0</td>
<td>0.0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>AFG</td>
<td>Asia</td>
<td>Afghanistan</td>
<td>2020-02-26</td>
<td>5.0</td>
<td>0.0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AFG</td>
<td>Asia</td>
<td>Afghanistan</td>
<td>2020-02-27</td>
<td>5.0</td>
<td>0.0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>AFG</td>
<td>Asia</td>
<td>Afghanistan</td>
<td>2020-02-28</td>
<td>5.0</td>
<td>0.0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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.

```python
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
```

```
<table>
<thead>
<tr>
<th>location</th>
<th>date</th>
<th>TotalCases</th>
<th>TotalDeaths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>2020</td>
<td>52332.0</td>
<td>2189.0</td>
</tr>
<tr>
<td></td>
<td>2021</td>
<td>105754.0</td>
<td>5167.0</td>
</tr>
<tr>
<td></td>
<td>2022</td>
<td>16136.0</td>
<td>263.0</td>
</tr>
<tr>
<td>Africa</td>
<td>2020</td>
<td>2760926.0</td>
<td>65507.0</td>
</tr>
<tr>
<td></td>
<td>2021</td>
<td>6977423.0</td>
<td>163031.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Zambia</td>
<td>2021</td>
<td>233549.0</td>
<td>3346.0</td>
</tr>
<tr>
<td></td>
<td>2022</td>
<td>59339.0</td>
<td>224.0</td>
</tr>
<tr>
<td>Zimbabwe</td>
<td>2020</td>
<td>13873.0</td>
<td>363.0</td>
</tr>
<tr>
<td></td>
<td>2021</td>
<td>199391.0</td>
<td>4641.0</td>
</tr>
<tr>
<td></td>
<td>2022</td>
<td>25481.0</td>
<td>393.0</td>
</tr>
</tbody>
</table>

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.

```python
grouped_data = year_grpd.groupby('location').sum()
grouped_data
```

In [ ]: #

Out[ ]:

<table>
<thead>
<tr>
<th>location</th>
<th>TotalCases</th>
<th>TotalDeaths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>174222.0</td>
<td>7619.0</td>
</tr>
<tr>
<td>Africa</td>
<td>11247473.0</td>
<td>248869.0</td>
</tr>
<tr>
<td>Albania</td>
<td>272030.0</td>
<td>3478.0</td>
</tr>
<tr>
<td>Algeria</td>
<td>265186.0</td>
<td>6852.0</td>
</tr>
<tr>
<td>Andorra</td>
<td>38434.0</td>
<td>151.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Wallis and Futuna</td>
<td>454.0</td>
<td>7.0</td>
</tr>
<tr>
<td>World</td>
<td>442613253.0</td>
<td>5964020.0</td>
</tr>
<tr>
<td>Yemen</td>
<td>11775.0</td>
<td>2135.0</td>
</tr>
<tr>
<td>Zambia</td>
<td>313613.0</td>
<td>3958.0</td>
</tr>
<tr>
<td>Zimbabwe</td>
<td>238745.0</td>
<td>5397.0</td>
</tr>
</tbody>
</table>

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.

```python
world_covid_cases.index.names = ['NAME']
world_covid_cases
```
In [ ]: #

Out[ ]:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TotalCases</th>
<th>TotalDeaths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>174222.0</td>
<td>7619.0</td>
</tr>
<tr>
<td>Africa</td>
<td>11247473.0</td>
<td>248869.0</td>
</tr>
<tr>
<td>Albania</td>
<td>272030.0</td>
<td>3478.0</td>
</tr>
<tr>
<td>Algeria</td>
<td>265186.0</td>
<td>6852.0</td>
</tr>
<tr>
<td>Andorra</td>
<td>38434.0</td>
<td>151.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Wallis and Futuna</td>
<td>454.0</td>
<td>7.0</td>
</tr>
<tr>
<td>World</td>
<td>442613253.0</td>
<td>5964020.0</td>
</tr>
<tr>
<td>Yemen</td>
<td>11775.0</td>
<td>2135.0</td>
</tr>
<tr>
<td>Zambia</td>
<td>313613.0</td>
<td>3958.0</td>
</tr>
<tr>
<td>Zimbabwe</td>
<td>238745.0</td>
<td>5397.0</td>
</tr>
</tbody>
</table>

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.

```python
import geopandas as gpd
world_data = gpd.read_file('geodata/World_Map.shp')
world_data.plot()
```
In [ ]: #

Out[ ]: <AxesSubplot: >

![World map](image)

Then as a dataframe, just `world_data`

In [ ]: #

Out[ ]:

<table>
<thead>
<tr>
<th>NAME</th>
<th>geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Antigua and Barbuda</td>
<td>MULTIPOLYGON (((-61.68667 17.02444, -61.73806 ...</td>
</tr>
<tr>
<td>1 Algeria</td>
<td>POLYGON ((2.96361 36.80222, 2.98139 36.80694, ...</td>
</tr>
<tr>
<td>2 Azerbaijan</td>
<td>MULTIPOLYGON (((45.08332 39.76804, 45.26639 39...</td>
</tr>
<tr>
<td>3 Albania</td>
<td>POLYGON ((19.43621 41.02107, 19.45055 41.06000...</td>
</tr>
<tr>
<td>4 Armenia</td>
<td>MULTIPOLYGON (((45.57305 40.63249, 45.52888 40...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>240 Saint Barthelemy</td>
<td>POLYGON ((-63.02834 18.01555, -63.03334 18.015...</td>
</tr>
<tr>
<td>241 Guernsey</td>
<td>POLYGON ((-2.59083 49.42249, -2.59722 49.42249...</td>
</tr>
<tr>
<td>242 Jersey</td>
<td>POLYGON ((-2.01500 49.21416, -2.02111 49.17722...</td>
</tr>
<tr>
<td>243 South Georgia South Sandwich Islands</td>
<td>MULTIPOLYGON (((-27.32584 -59.42722, -27.29806...</td>
</tr>
<tr>
<td>244 Taiwan</td>
<td>MULTIPOLYGON (((121.57639 22.00139, 121.57027 ...</td>
</tr>
</tbody>
</table>

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 Shape file and only print out the values that do not exist within both lists.

```python
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".

```python
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.

```python
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)
```

```python
In [ ]: #
```

Now we can use the .merge() method to join our tabular World covid cases to our World shapefile.

```python
combined = world_data.merge(world_covid_cases, on = 'NAME')
combined
```
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)

```python
define the plots
    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()
```
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)
Read in the BA_Counties.shp feature class, and then view it

```python
import geopandas as gpd
import pandas as pd
ba_counties = gpd.read_file('geodata/BA_Counties.shp')
ba_counties
```

In [ ]:

Out[ ]:

<table>
<thead>
<tr>
<th>OBJECTID</th>
<th>Acres</th>
<th>County</th>
<th>Shape_Leng</th>
<th>Shape_Area</th>
<th>geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.763008e+05</td>
<td>Alameda</td>
<td>351719.536494</td>
<td>1.927521e+09</td>
<td>MULTIPOLYGON</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(((6033505.488, 2112049.179, 6033189.251, 2057866.467, 5999063.754, 2057866.467, 5999063.754, 2112049.179, 6033189.251))...</td>
</tr>
<tr>
<td>1</td>
<td>4.811194e+05</td>
<td>Contra</td>
<td>292676.346086</td>
<td>1.947021e+09</td>
<td>MULTIPOLYGON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Costa</td>
<td></td>
<td></td>
<td>(((6027405.300, 2153185.401, 6027275.040, 2153075.499, 6027275.040, 2153075.499, 6027405.300, 2153185.401))...</td>
</tr>
<tr>
<td>2</td>
<td>3.360223e+05</td>
<td>Marin</td>
<td>419122.293142</td>
<td>1.359834e+09</td>
<td>MULTIPOLYGON</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(((5988028.191, 2211673.341, 5987863.117, 2211673.341, 5988028.191, 2211673.341, 5987863.117, 2211673.341))...</td>
</tr>
<tr>
<td>3</td>
<td>5.054966e+05</td>
<td>Napa</td>
<td>278766.105739</td>
<td>2.045672e+09</td>
<td>POLYGON</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(6021779.628, 2506831.778, 6021843.051, 2506831.778, 6021779.628))...</td>
</tr>
<tr>
<td>4</td>
<td>3.021589e+04</td>
<td>San</td>
<td>106843.644006</td>
<td>1.222794e+08</td>
<td>MULTIPOLYGON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Francisco</td>
<td></td>
<td></td>
<td>(((6006624.799, 2128525.324, 6006154.530, 2128525.324, 6006624.799, 2128525.324, 6006154.530, 2128525.324))...</td>
</tr>
<tr>
<td>5</td>
<td>2.902726e+05</td>
<td>San</td>
<td>306693.656488</td>
<td>1.174692e+09</td>
<td>MULTIPOLYGON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mateo</td>
<td></td>
<td></td>
<td>(((6068478.537, 2017484.502, 6068008.537, 2017484.502, 6068478.537, 2017484.502, 6068008.537, 2017484.502))...</td>
</tr>
<tr>
<td>6</td>
<td>8.312778e+05</td>
<td>Santa</td>
<td>380752.936529</td>
<td>3.364062e+09</td>
<td>POLYGON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clara</td>
<td></td>
<td></td>
<td>(5987850.662, 2233054.492, 5987868.159, 2233054.492, 5987850.662))...</td>
</tr>
<tr>
<td>7</td>
<td>5.437972e+05</td>
<td>Solano</td>
<td>330514.170440</td>
<td>2.200669e+09</td>
<td>MULTIPOLYGON</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(((6121744.455, 2214795.253, 6121640.077, 2214795.253, 6121744.455, 2214795.253, 6121640.077, 2214795.253))...</td>
</tr>
<tr>
<td>8</td>
<td>1.015939e+06</td>
<td>Sonoma</td>
<td>400983.408337</td>
<td>4.111360e+09</td>
<td>POLYGON</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(5987850.662, 2233054.492, 5987868.159, 2233054.492, 5987850.662))...</td>
</tr>
</tbody>
</table>

How many features in the dataset?

```python
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?
List the unique geometry types in this geodataframe.

```python
ba_counties.type.unique()
```

```
Out[ ]:
array(['MultiPolygon', 'Polygon'], dtype=object)
```

Why do you think there are these geometry types?

Examine the attributes for the first feature

```python
ba_counties.iloc[0]
```

```
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).
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.

```python
ba_counties_UTM = ba_counties.to_crs('EPSG:32610')
```

View the crs after we've assigned the projection

```python
ba_counties_UTM.crs
```
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.

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.

We can show just the first value, which will appear as a shape.

```
ba_counties_UTM['geometry'][0]
```
What does the following code produce and why?

```python
for i in range(len(ba_counties_UTM)):
    ba_counties_UTM['geometry'][i]
```
In [ ]: #
Extract one feature geometry to a variable and show its type (not the same as dtype).

```python
thePoly = ba_counties_UTM['geometry'][0]
type(thePoly)
```

In [ ]: #

Out[ ]: `shapely.geometry.multipolygon.MultiPolygon`

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).

```python
thePoly
```

In [ ]: #

Out[ ]: 

Look at other geometric properties for `thePoly` using the properties `.area` and `.length`.

What do you think this length represents?

```python
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.

```python
In [ ]: #
Out[ ]: 

Out[ ]: shapely.geometry.multilinestring.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.

```python
In [ ]: #

 POINT (598249.2377859637 4167169.885425969)
Out[ ]: shapely.geometry.point.Point
```

```python
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.

```python
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.

```python
#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.

```python
theDistances.mean()

In [ ]: #
Out[ ]: 21.463153699118223
```
Then we can derive a histogram of these distances.

```python
theDistances.hist(figsize=(15,3))
```

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...

```python
countyMask = ba_counties_UTM['County'] == 'Alameda'
Alameda = ba_counties_UTM[countyMask]
```

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.

```python
feature_geometry = Alameda['geometry']
type(feature_geometry)
features_geometry = ba_counties_UTM['geometry']
type(features_geometry)
```

What do you get if you pull out the `geometry` field?

```python
In [1]:
In [2]:
Out[2]:
```

<table>
<thead>
<tr>
<th>OBJECTID</th>
<th>Acres</th>
<th>County</th>
<th>Shape_Leng</th>
<th>Shape_Area</th>
<th>geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>476300.784182</td>
<td>Alameda</td>
<td>351719.536494</td>
<td>1.927521e+09     ((559203.322 4181745.002, 559093...)</td>
</tr>
</tbody>
</table>
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.

```python
ba_counties_copy = ba_counties_UTM.copy()
ba_counties_copy['geometry'] = ba_counties_copy['geometry'].centroid.buffer(5000)
ba_counties_copy.plot()
```

```
In [ ]: #
Out[ ]: geopandas.geoseries.GeoSeries
Out[ ]: geopandas.geoseries.GeoSeries
```

![Plot of replaced geometries](image)
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.

```python
Use `pd.read_csv()` to read this file and assign it to a new dataframe `sf_311`.
```
In [ ]: #

Out[ ]:

<table>
<thead>
<tr>
<th></th>
<th>CaseID</th>
<th>Opened</th>
<th>Closed</th>
<th>Updated</th>
<th>Status</th>
<th>Status_Notes</th>
<th>Responsible_Ag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13138553</td>
<td>11/13/2020</td>
<td>NaN</td>
<td>11/13/2020</td>
<td>Open</td>
<td>NaN</td>
<td>Port Author</td>
</tr>
<tr>
<td>1</td>
<td>13119442</td>
<td>11/8/2020</td>
<td>11/8/2020</td>
<td>11/8/2020</td>
<td>Closed</td>
<td>Case Resolved - Police Officer responded to re...</td>
<td>Parking Enforc Dispatch</td>
</tr>
<tr>
<td>2</td>
<td>13091973</td>
<td>11/1/2020</td>
<td>11/1/2020</td>
<td>11/1/2020</td>
<td>Closed</td>
<td>Case Resolved - POLICE MATTER. PLEASE NOTIFY ...</td>
<td>311 Supervisor</td>
</tr>
<tr>
<td>3</td>
<td>13155970</td>
<td>11/18/2020</td>
<td>11/18/2020</td>
<td>11/18/2020</td>
<td>Closed</td>
<td>Case Resolved - Officer responded to request u...</td>
<td>Parking Enforc Dispatch</td>
</tr>
<tr>
<td>4</td>
<td>13180413</td>
<td>11/24/2020</td>
<td>11/24/2020</td>
<td>11/24/2020</td>
<td>Closed</td>
<td>Case Resolved - Officer responded to request u...</td>
<td>Parking Enforc Dispatch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47991</td>
<td>13096891</td>
<td>11/2/2020</td>
<td>11/4/2020</td>
<td>11/4/2020</td>
<td>Closed</td>
<td>Comment Noted - To view the status, see\n13097...</td>
<td></td>
</tr>
<tr>
<td>47992</td>
<td>13096288</td>
<td>11/2/2020</td>
<td>11/3/2020</td>
<td>11/3/2020</td>
<td>Closed</td>
<td>Case Transferred - DPH Environmental Health\n1...</td>
<td></td>
</tr>
<tr>
<td>47993</td>
<td>13094573</td>
<td>11/1/2020</td>
<td>11/2/2020</td>
<td>11/2/2020</td>
<td>Closed</td>
<td>Case is a Duplicate - \n13094578,11/01/2020 0...</td>
<td></td>
</tr>
<tr>
<td>47994</td>
<td>13093656</td>
<td>11/1/2020</td>
<td>11/17/2020</td>
<td>11/17/2020</td>
<td>Closed</td>
<td>Cancelled - \n13041136,10/19/2020 11:25:00 AM...</td>
<td></td>
</tr>
<tr>
<td>47995</td>
<td>13092881</td>
<td>11/1/2020</td>
<td>11/6/2020</td>
<td>11/6/2020</td>
<td>Closed</td>
<td>Comment Noted - To view the status, see\n11470...</td>
<td></td>
</tr>
</tbody>
</table>

47996 rows × 19 columns

Display all the field names within our dataframe using the .columns property.
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 [ ]: #
Out[ ]: 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 [ ]: #
Out[ ]: 46
```

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.)

```python
sf_311 = sf_311[sf_311['Latitude'].notna() & sf_311['Longitude'].notna()]
```

Use the missing_coordinates method from above to confirm that there are no missing coordinates.

```python
In []: #
Out[ ]: 0
```

Next, use `gpd.readfile()` and `.plot()` to assign a new GeoDataFrame `sf_neighborhoods` from `geodata/sf_neighborhoods.shp` and plot it.

```python
In []: #
Out[ ]: <AxesSubplot:>
```
Examine the `sf_neighborhoods` GeoDataFrame as a table.

```
In [ ]: #

Out[ ]:

<table>
<thead>
<tr>
<th>link</th>
<th>name</th>
<th>geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://en.wikipedia.org/wiki/Sea_Cliff,_San_Fr">http://en.wikipedia.org/wiki/Sea_Cliff,_San_Fr</a>...</td>
<td>Seacliff</td>
<td>POLYGON ((-122.49346 37.78352, -122.49373 37.7...</td>
</tr>
<tr>
<td>None</td>
<td>Lake Street</td>
<td>POLYGON ((-122.48715 37.78379, -122.48729 37.7...</td>
</tr>
<tr>
<td><a href="http://www.nps.gov/prsf/index.htm">http://www.nps.gov/prsf/index.htm</a></td>
<td>Presidio National Park</td>
<td>POLYGON ((-122.47758 37.81099, -122.47712 37.8...</td>
</tr>
<tr>
<td>None</td>
<td>Presidio Terrace</td>
<td>POLYGON ((-122.47241 37.78735, -122.47100 37.7...</td>
</tr>
<tr>
<td><a href="http://www.sfgate.com/neighborhoods/sf/innerri">http://www.sfgate.com/neighborhoods/sf/innerri</a>...</td>
<td>Inner Richmond</td>
<td>POLYGON ((-122.47263 37.78631, -122.46683 37.7...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><a href="http://en.wikipedia.org/wiki/Corona_Heights,_S">http://en.wikipedia.org/wiki/Corona_Heights,_S</a>...</td>
<td>Corona Heights</td>
<td>POLYGON ((-122.43519 37.76267, -122.43532 37.7...</td>
</tr>
<tr>
<td><a href="http://en.wikipedia.org/wiki/Haight-Ashbury">http://en.wikipedia.org/wiki/Haight-Ashbury</a></td>
<td>Ashbury Heights</td>
<td>POLYGON ((-122.45196 37.76148, -122.45210 37.7...</td>
</tr>
<tr>
<td><a href="http://en.wikipedia.org/wiki/Eureka_Valley,_Sa">http://en.wikipedia.org/wiki/Eureka_Valley,_Sa</a>...</td>
<td>Eureka Valley</td>
<td>POLYGON ((-122.43734 37.76235, -122.43704 37.7...</td>
</tr>
<tr>
<td><a href="http://en.wikipedia.org/wiki/St._Francis_Wood">http://en.wikipedia.org/wiki/St._Francis_Wood</a>,...</td>
<td>St. Francis Wood</td>
<td>POLYGON ((-122.47157 37.73471, -122.46831 37.7...</td>
</tr>
<tr>
<td><a href="http://en.wikipedia.org/wiki/Neighborhoods_in">http://en.wikipedia.org/wiki/Neighborhoods_in</a>_...</td>
<td>Sherwood Forest</td>
<td>POLYGON ((-122.45890 37.74054, -122.45877 37.7...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

117 rows × 3 columns
```

What's the crs of `sf_neighborhoods`?
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`?

```
In [ ]: #
```

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 

```ipython
# 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]
```

```ipython
# 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
```

```ipython
# 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.

```ipython
import contextily as cx
```

```ipython
# In
cx
```
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.

```python
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:

```python
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=ctx.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).

```python
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.

```python
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=ax1, 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, 4555646.885796504)

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.

```python
sf_311.Category.unique()
```

```
In [ ]: #
```

We can create a new geodataframe that only contains the Category 'Parking Enforcement' and call it "parking_issues".

```python
parking_issues = sf_311.query("Category == 'Parking Enforcement'")
parking_issues
```

```
```
<table>
<thead>
<tr>
<th>CaseID</th>
<th>Opened</th>
<th>Updated</th>
<th>Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>13119442</td>
<td>11/8/2020 9:40</td>
<td>11/8/2020 9:45</td>
<td>Closed</td>
<td>Case Resolved - Police Officer responded to re...</td>
</tr>
<tr>
<td>13155970</td>
<td>11/18/2020 6:27</td>
<td>11/18/2020 7:27</td>
<td>Closed</td>
<td>Case Resolved - Officer responded to request u...</td>
</tr>
<tr>
<td>13180413</td>
<td>11/24/2020 14:19</td>
<td>11/24/2020 17:06</td>
<td>Closed</td>
<td>Case Resolved - Officer responded to request u...</td>
</tr>
<tr>
<td>13119107</td>
<td>11/8/2020 8:01</td>
<td>11/8/2020 8:22</td>
<td>Closed</td>
<td>Case Resolved - Officer responded to request u...</td>
</tr>
<tr>
<td>13119507</td>
<td>11/8/2020 10:04</td>
<td>11/8/2020 10:12</td>
<td>Closed</td>
<td>Case Resolved - Police Officer responded to re...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>47687</td>
<td>11/25/2020 6:45</td>
<td>11/25/2020 6:50</td>
<td>Closed</td>
<td>Case is a Duplicate - This issue has already b...</td>
</tr>
<tr>
<td>47688</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>47689</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>47737</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>47924</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Responsible_Agency</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parking Enforcement Dispatch Queue</td>
<td>Parking Enforcement</td>
</tr>
<tr>
<td>Parking Enforcement Dispatch Queue</td>
<td>Parking Enforcement</td>
</tr>
<tr>
<td>Parking Enforcement Dispatch Queue</td>
<td>Parking Enforcement</td>
</tr>
<tr>
<td>Parking Enforcement Dispatch Queue</td>
<td>Parking Enforcement</td>
</tr>
<tr>
<td>Parking Enforcement Dispatch Queue</td>
<td>Parking Enforcement</td>
</tr>
<tr>
<td>Parking Enforcement Dispatch Queue</td>
<td>Parking Enforcement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Request_Type</th>
<th>Request_Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other_Illegal_Parking</td>
<td>Parking Enforcement</td>
</tr>
<tr>
<td>Other_Illegal_Parking</td>
<td>Blue - Chrysler - 51rx746</td>
</tr>
<tr>
<td>Other_Illegal_Parking</td>
<td>Blue - Honda - 8duj830</td>
</tr>
<tr>
<td>Other_Illegal_Parking</td>
<td>Blue - Honda - Aw70c09</td>
</tr>
<tr>
<td>Other_Illegal_Parking</td>
<td>Blue - Honda - Aw70c09</td>
</tr>
<tr>
<td>Parking_on_Sidewalk</td>
<td>silver - cadillac escalade - 7wxn339</td>
</tr>
<tr>
<td>Parking_on_Sidewalk</td>
<td>silver - cadillac escalade - 7wxn339</td>
</tr>
<tr>
<td>Parking_on_Sidewalk</td>
<td>silver - cadillac escalade - 7wxn339</td>
</tr>
<tr>
<td>Other_Illegal_Parking</td>
<td>White - Chevy van - None</td>
</tr>
<tr>
<td>Other_Illegal_Parking</td>
<td>Blue - Ford suburban - Aksj</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Street</th>
</tr>
</thead>
<tbody>
<tr>
<td>57 INNES CT, SAN FRANCISCO, CA, 94124</td>
<td>INNES CT</td>
</tr>
<tr>
<td>51 INNES CT, SAN FRANCISCO, CA, 94124</td>
<td>INNES CT</td>
</tr>
<tr>
<td>51 INNES CT, SAN FRANCISCO, CA, 94124</td>
<td>INNES CT</td>
</tr>
<tr>
<td>Supervisor_District</td>
<td>Neighborhood Police District</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>1</td>
<td>10.0</td>
</tr>
<tr>
<td>3</td>
<td>10.0</td>
</tr>
<tr>
<td>4</td>
<td>10.0</td>
</tr>
<tr>
<td>5</td>
<td>10.0</td>
</tr>
<tr>
<td>6</td>
<td>10.0</td>
</tr>
<tr>
<td>47687</td>
<td>4.0</td>
</tr>
<tr>
<td>47688</td>
<td>4.0</td>
</tr>
<tr>
<td>47689</td>
<td>4.0</td>
</tr>
<tr>
<td>47737</td>
<td>1.0</td>
</tr>
<tr>
<td>47924</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Longitude</th>
<th>Point</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 -122.367097 (37.72707865, -122.36709653)</td>
<td>Mobile/Open311</td>
<td></td>
</tr>
<tr>
<td>3 -122.367930 (37.72736328, -122.36792978)</td>
<td>Mobile/Open311</td>
<td></td>
</tr>
<tr>
<td>4 -122.367940 (37.72735638, -122.36793984)</td>
<td>Mobile/Open311</td>
<td></td>
</tr>
<tr>
<td>5 -122.367988 (37.72725562, -122.36798812)</td>
<td>Mobile/Open311</td>
<td></td>
</tr>
<tr>
<td>6 -122.367992 (37.72733809, -122.36799181)</td>
<td>Mobile/Open311</td>
<td></td>
</tr>
<tr>
<td>47687 -122.509000 (37.7607147, -122.5090004)</td>
<td>Web</td>
<td></td>
</tr>
<tr>
<td>47688 -122.509000 (37.7607147, -122.5090004)</td>
<td>Web</td>
<td></td>
</tr>
<tr>
<td>47689 -122.509000 (37.7607147, -122.5090004)</td>
<td>Web</td>
<td></td>
</tr>
<tr>
<td>47737 -122.509327 (37.77137094, -122.50932645)</td>
<td>Mobile/Open311</td>
<td></td>
</tr>
<tr>
<td>47924 -122.511296 (37.77557256, -122.51129619)</td>
<td>Mobile/Open311</td>
<td></td>
</tr>
</tbody>
</table>

geometry

1 POINT (-13621842.872 4540942.599)
3 POINT (-13621935.635 4540982.659)
4 POINT (-13621936.748 4540981.687)
5 POINT (-13621942.125 4540967.506)
6 POINT (-13621942.537 4540979.113)
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 4547778.075)
Let's plot the parking issues only and then set the extent of our map to the total extent of the parking issues geodataframe.

```python
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()
```
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.

```python
sf_neighborhoods = sf_neighborhoods.to_crs('EPSG:3857')
```
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.

```python
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[ ]:

<table>
<thead>
<tr>
<th></th>
<th>link</th>
<th>name</th>
<th>geometry</th>
<th>parking_incidents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><a href="http://en.wikipedia.org/wiki/Sea_Cliff,_San_Fr">http://en.wikipedia.org/wiki/Sea_Cliff,_San_Fr</a>...</td>
<td>Seacliff</td>
<td>POLYGON ((-13635909.066, 4548889.167, -13635939...</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>None</td>
<td>Lake Street</td>
<td>POLYGON ((-13635207.246, 4548926.811, -13635222...</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td><a href="http://www.nps.gov/prsf/index.htm">http://www.nps.gov/prsf/index.htm</a></td>
<td>Presidio National Park</td>
<td>POLYGON ((-13634141.858, 4552759.797, -13634091...</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>None</td>
<td>Presidio Terrace</td>
<td>POLYGON ((-13633566.376, 4549428.413, -13633409...</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td><a href="http://www.sfgate.com/neighborhoods/sf/innerri">http://www.sfgate.com/neighborhoods/sf/innerri</a>...</td>
<td>Inner Richmond</td>
<td>POLYGON ((-13633590.339, 4549283.086, -13632945...</td>
<td>13</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>112</td>
<td><a href="http://en.wikipedia.org/wiki/Corona_Heights,_S">http://en.wikipedia.org/wiki/Corona_Heights,_S</a>...</td>
<td>Corona Heights</td>
<td>POLYGON ((-13629422.779, 4545953.181, -13629437...</td>
<td>5</td>
</tr>
<tr>
<td>113</td>
<td><a href="http://en.wikipedia.org/wiki/Haight-Ashbury">http://en.wikipedia.org/wiki/Haight-Ashbury</a></td>
<td>Ashbury Heights</td>
<td>POLYGON ((-13631289.585, 4545785.263, -13631305...</td>
<td>91</td>
</tr>
<tr>
<td>114</td>
<td><a href="http://en.wikipedia.org/wiki/Eureka_Valley,_San">http://en.wikipedia.org/wiki/Eureka_Valley,_San</a>...</td>
<td>Eureka Valley</td>
<td>POLYGON ((-13629662.606, 4545908.690, -13629628...</td>
<td>0</td>
</tr>
<tr>
<td>115</td>
<td><a href="http://en.wikipedia.org/wiki/St._Francis_Wood">http://en.wikipedia.org/wiki/St._Francis_Wood</a>,...</td>
<td>St. Francis Wood</td>
<td>POLYGON ((-13633472.736, 4542016.274, -13633109...</td>
<td>9</td>
</tr>
<tr>
<td>116</td>
<td><a href="http://en.wikipedia.org/wiki/Neighborhoods_in">http://en.wikipedia.org/wiki/Neighborhoods_in</a>...</td>
<td>Sherwood Forest</td>
<td>POLYGON ((-13632062.910, 4542836.706, -13632047...</td>
<td>0</td>
</tr>
</tbody>
</table>

117 rows × 4 columns
Here we can plot all our 311 parking issues by neighborhood.

```python
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.
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.

```python
import arcpy
```

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.

```python
from arcpy import env
```

If you're working in ArcGIS Pro, it doesn't hurt to include it anyway just so your notebook will work in either place.

```python
print(env.workspace)
```

```
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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.

```python
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
```

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.

```python
ENV = arcpy.EnvManager
with ENV(workspace="hmb.gdb"): arcpy.ListRasters()
```

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.
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`.

```python
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.

```python
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`.

```python
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_e38fbd3359cf_2453623504', 'numpyarraytoraster_445f664f_38d8_4a15_be0b_f770a0be3df1_401611712', 'numpyarraytoraster_fc5bc198_377a_4502_ae67_77a6c4662b91_401611712', 'numpyarraytoraster_249bae79_4b16_4f0b_b510_93f6efada75_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_bca4b549928_2385214608', 'highElev_np', 'numpyarraytoraster_3513d5c7_61af_488f_0a7d_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 0x0000019A2E587EF0>
<geoprocessing describe field object object at 0x0000019A2E587EO0>
<geoprocessing describe field object object at 0x0000019A2E587ED0>

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".

```python
for ws in arcpy.ListWorkspaces("*","FileGDB"):  
    print(ws)  
    with ENV(workspace = ws):  
        if arcpy.ListFeatureClasses():  
            print("There are {} FeatureClasses:").format(len(arcpy.ListFeatureClasses()))
        for dtype in ["point", "line", "polygon"]:  
            fs = arcpy.ListFeatureClasses(feature_type=dtype)  
            if fs:  
                print("\t{}").format(dtype))
                for f in fs:  
                    print("\tt{}").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_12ff456bdc6_1806419640
   numpyarraytoraster_254a82f0_c9f2_4f87_ab4b_2f25d21acada4_1787618684
   numpyarraytoraster_e0f844ec_c5ee_44b1_af76_37c43839e3ae_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_dbdb92283d1c0_284017208
   numpyarraytoraster_dbe317f_ddb6_440b_8a5e_36af55b7a07b_284017208
   numpyarraytoraster_13b48e1f_9a2f_4694_b3b6_e22866798da1_2453623504
   numpyarraytoraster_95fa8d4c_0930_4ab2_a851_e38fbd3359cf_2453623504
   numpyarraytoraster_445f664f_38d8_4a15_be0b_f770a0e3df1_4016111712
   numpyarraytoraster_fc5bc198_377a_4502_ae67_77a6c4662b91_4016111712
   numpyarraytoraster_249bae79_4b16_4f0b_b510_93f6be5a2b75_2888418060
   numpyarraytoraster_2d8b9707_cc96_47d1_92ad_c7a458a74fee_2888418060
   numpyarraytoraster_83003b86_b865_4c3b_a796_98bd8e4c52b3_2888418060
   numpyarraytoraster_0270248f_2469_4a08_b20a_0d192898984f_2470410924
   numpyarraytoraster_8cb26adc_892d_437c_a49b_b9f3ef15169b_2470410924
   numpyarraytoraster_d5ff6af0_c750_428f_bb6e_a6ad6d32c54_2470410924
   steepurban
   steepurban
   steepurban
   steepurban
   steepurban
   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\exercise_html\results\Ex07_arcpyIntro_results.html
There are 11 FeatureClasses:
  point
    co2july95
    samples
    marblePts
  line
    streams
    trails
    contours10m
    cont
  polygon
    geology
    veg
    water
    watrshd

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:

<table>
<thead>
<tr>
<th>toolbox</th>
<th>alias</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td>analysis</td>
<td>AN</td>
</tr>
<tr>
<td>Conversion</td>
<td>conversion</td>
<td>CV</td>
</tr>
<tr>
<td>Data Management</td>
<td>management</td>
<td>DM</td>
</tr>
</tbody>
</table>

So to set up these shortcuts, we could include the following code:

```python
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:

```python
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:

```python
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.

```python
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`:

```python
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})

Expects 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 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 explicitly (so `clip_features="areaclip"` for instance), since even though verbose it makes your code more readable, and you can skip over unneeded parameters.

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.

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.

```python
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.

```python
In [ ]: #
Out[ ]:
```

Use a series of map algebra statements to end with a `steepurban` raster object that represents slopes > 10 and landuse < 20 (urban):

```python
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.
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.

```python
# 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.

```python
elev1D = np.reshape(elev2D, -1)
```

```python
# Create a histogram from the 1D ndarray.
```
Now we can make the histogram in matplotlib:

```python
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`:

```python
sr = elev.spatialReference
lowleft = elev.extent.lowerLeft
cellsize = elev.meanCellHeight
```

```
In [ ]: #
    #
Out[ ]: 60.0
```

Then we'll convert to a ndarray

```python
elev2D = arcpy.RasterToNumPyArray(elev)
```

```
In [ ]: #
```

Then we'll see what we have as a 2D ndarray:

```python
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:

```python
highthreshold = 300
highElev_np = elev2D > highthreshold
```

Then display that:

```python
fig = plt.figure()
plt.imshow(highElev_np)
plt.colorbar()
plt.title("high elev (elev > {})".format(highthreshold))
plt.show()
```

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.

```python
highElev_np1 = (highElev_np * 2)/2
```

Then we can bring it back in and apply the various environment settings we grabbed earlier:

```python
highElev0 = arcpy.NumPyArrayToRaster(highElev_np1, lowleft, cellsize, cellsize)
DM.DefineProjection(highElev0, sr)
```

For some reason, we need to make the result discrete, so the following works for that:

```python
highElev = Con(highElev0==1,1,0)
highElev.save("highElev_np")
highElev
```
Here's a simpler example that doesn't need unique values, and stands to represent a process involving continuous numerical data:

```python
elevft2D = elev2D / 0.3048
elevft2D

elevft = arcpy.NumPyArrayToRaster(elevft2D, lowleft, cellsize, cellsize)
arcpy.DefineProjection_management(elevft, sr)
elevft
```

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`.
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).


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', 'landing', 'geo1str200', '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[ ]:

catalogPath C:\Users\900008452\Box\course\625\exer\Ex07_08_arcpy\hmb.gdb\elev
dataType RasterDataset
bandCount 1
format FGDBR

fields

spatialReference

name (Projected Coordinate System) NAD_1983_UTM_Zone_10N
factoryCode (WKID) 26910
linearUnitName (Linear Unit) Meter

spatialReference.GCS

name (Geographic Coordinate System) GCS_North_American_1983
factoryCode (WKID) 4269
angularUnitName (Angular Unit) Degree
datumName (Datum) 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:

```python
ext = dsc.extent
ext
```
In [ ]: #

Out[ ]: 

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>XMin (Left)</td>
<td>545692.537124</td>
</tr>
<tr>
<td>XMax (Right)</td>
<td>557692.537124</td>
</tr>
<tr>
<td>YMin (Bottom)</td>
<td>4141427.326978</td>
</tr>
<tr>
<td>YMax (Top)</td>
<td>4153427.326978</td>
</tr>
</tbody>
</table>

spatialReference

| name (Projected Coordinate System) | NAD_1983_UTM_Zone_10N |
| factoryCode (WKID) | 26910 |
| linearUnitName (Linear Unit) | Meter |

spatialReference.GCS

| name (Geographic Coordinate System) | GCS_North_American_1983 |
| factoryCode (WKID) | 4269 |
| angularUnitName (Angular Unit) | Degree |
| datumName (Datum) | 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:

```python
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:

```python
arcpy.Describe("elev").extent.XMin
```

e tc., 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

   ```python
   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.

```python
In [ ]:
Out[ ]: '549692.5371239004 4145427.3269781955 553692.5371239004 4149427.3269781955'
```

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.

```python
Interpret what the above is doing, including comparing with what you see with `Raster("elev")`.
```
Still with the workspace set to `hmb.gdb`, see what you get with:

```python
arcpy.Describe("streams").DataType
arcpy.Describe("landing").ShapeType
arcpy.Describe("landuse").DataType
arcpy.Describe("geology").DataType
```

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:

  ```python
  arcpy.Describe("imagery/landsatHMB201707.tif/Band_1").meanCellWidth
  ```

  ... but to get the bandCount, this would be:

  ```python
  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 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.
Exist

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.

```python
if arcpy.Exists("empty"):  DM.Delete("empty")
```

```python
['streams', 'areaclip', 'faults', 'pourpoint', 'publands', 'roads', 'landing', 'geolstr200', 'geol', 'stbuff200', 'empty']
```

```python
['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.

```python
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', 'landing', 'geolstr200', 'geol', 'stbuff200', 'empty']

In [ ]: #
['streams', 'areaclip', 'faults', 'pourpoint', 'publands', 'roads', 'landing', '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.

```python
if not arcpy.ListFields("streams","stream_class"):  
    DM.AddField('streams','stream_class', 'LONG')
```

Check to see if this worked with

```python
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.
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.

```python
In [ ]:
#

OBJECTID
Shape
FNODE_
TNODE_
L_POLY_
R_POLY_
LENGTH
STREAMS_
STREAMS_ID
ST_CODE
Shape_Length
stream_class

OBJECTID
Shape
FNODE_
TNODE_
L_POLY_
R_POLY_
LENGTH
STREAMS_
STREAMS_ID
ST_CODE
Shape_Length
OBJECTID
Shape
FNODE_
TNODE_
L_POLY_
R_POLY_
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.

```python
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.)
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.

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.
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))`

```python
feat = ["geol", "landusePen"]
fld = ["TYPE-ID", "LU-CODE"]
```

```
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.

Here's 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:

```python
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.

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.

<table>
<thead>
<tr>
<th>Toolbox</th>
<th>Tool</th>
<th>What it does</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td>Frequency</td>
<td>Calculates frequency statistics for field(s) in the input table</td>
<td>table</td>
</tr>
<tr>
<td>-</td>
<td>Statistics</td>
<td>Calculates summary statistics for field(s) in the input table.</td>
<td>table</td>
</tr>
<tr>
<td>-</td>
<td>Select</td>
<td>Uses a where clause to selects features from an input to create an output</td>
<td>new feature class</td>
</tr>
<tr>
<td>-</td>
<td>TableSelect</td>
<td>Extracts selected attributes from an input table, using an SQL Where clause</td>
<td>table</td>
</tr>
<tr>
<td>Data Management</td>
<td>AddField</td>
<td>Adds a field to an data table</td>
<td>field in existing table</td>
</tr>
<tr>
<td>-</td>
<td>CalculateField</td>
<td>Calculates a value for a field using an expression</td>
<td>values in an existing field</td>
</tr>
<tr>
<td>-</td>
<td>DeleteField</td>
<td>Removes a field from a table</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>AddXY</td>
<td>Adds an x &amp; y (and maybe z) fields to a point feature class</td>
<td>x, y, etc. fields with values</td>
</tr>
<tr>
<td>-</td>
<td>CopyFeatures</td>
<td>Copies selected features to a new feature class</td>
<td>new feature class</td>
</tr>
<tr>
<td>-</td>
<td>DeleteFeatures</td>
<td>Deletes selected features</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>AddJoin</td>
<td>Joins a table to a layer (or a table to a table) based on a common field</td>
<td>join relationship</td>
</tr>
<tr>
<td>-</td>
<td>RemoveJoin</td>
<td>Removes an existing join</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>SelectLayerByAttribute</td>
<td>Creates, updates, or removes a selection on the layer or table view</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>CreateFeatureclass</td>
<td>Create a new feature class (including shapefiles)</td>
<td>feature class</td>
</tr>
</tbody>
</table>
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 \( \frac{\text{elev}}{0.3048} \). (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:

  ```python
  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 "\( \frac{\text{Contour}}{0.3048} \)" 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.
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 using 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:

```python
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 using 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:

```python
In [ ]: 

```
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.

```python
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", 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)
In [ ]: #

whalesGGate exists.

Out[ ]:

<table>
<thead>
<tr>
<th>FID</th>
<th>Shape</th>
<th>Observer</th>
<th>Year</th>
<th>Month</th>
<th>Day</th>
<th>Hour</th>
<th>Minute</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(-122.493592 37.794274 NaN NaN)</td>
<td>Allison</td>
<td>2019</td>
<td>4</td>
<td>2</td>
<td>12</td>
<td>45</td>
<td>37.794274</td>
<td>-122.493592</td>
</tr>
<tr>
<td>1</td>
<td>(-122.459953 37.830697 NaN NaN)</td>
<td>Allison</td>
<td>2019</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>30</td>
<td>37.830697</td>
<td>-122.459953</td>
</tr>
<tr>
<td>2</td>
<td>(-122.599558 37.81561 NaN NaN)</td>
<td>Allison</td>
<td>2019</td>
<td>5</td>
<td>4</td>
<td>12</td>
<td>30</td>
<td>37.815610</td>
<td>-122.599558</td>
</tr>
<tr>
<td>3</td>
<td>(-122.604379 37.797698 NaN NaN)</td>
<td>Allison</td>
<td>2019</td>
<td>5</td>
<td>4</td>
<td>15</td>
<td>30</td>
<td>37.797698</td>
<td>-122.604379</td>
</tr>
<tr>
<td>4</td>
<td>(-122.527239 37.7292360000001 NaN NaN)</td>
<td>Allison</td>
<td>2019</td>
<td>5</td>
<td>14</td>
<td>11</td>
<td>30</td>
<td>37.729236</td>
<td>-122.527239</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>172</td>
<td>(-122.515109 37.8055390000001 NaN NaN)</td>
<td>Allison</td>
<td>2019</td>
<td>10</td>
<td>13</td>
<td>8</td>
<td>30</td>
<td>37.805535</td>
<td>-122.515109</td>
</tr>
<tr>
<td>173</td>
<td>(-122.809715 37.6611310000001 NaN NaN)</td>
<td>Allison</td>
<td>2019</td>
<td>10</td>
<td>13</td>
<td>10</td>
<td>0</td>
<td>37.661131</td>
<td>-122.809715</td>
</tr>
<tr>
<td>174</td>
<td>(-122.947331 37.730912 NaN NaN)</td>
<td>Allison</td>
<td>2019</td>
<td>10</td>
<td>13</td>
<td>2</td>
<td>0</td>
<td>37.730912</td>
<td>-122.947331</td>
</tr>
<tr>
<td>175</td>
<td>(-122.620435 37.782788 NaN NaN)</td>
<td>Allison</td>
<td>2019</td>
<td>11</td>
<td>9</td>
<td>12</td>
<td>30</td>
<td>37.782788</td>
<td>-122.620435</td>
</tr>
<tr>
<td>176</td>
<td>(-122.569333 37.768944 NaN NaN)</td>
<td>Allison</td>
<td>2019</td>
<td>11</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>37.768944</td>
<td>-122.569333</td>
</tr>
</tbody>
</table>

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`.

```python
import arcpy, os
projdir = os.getcwd()
from arcpy import env
ever.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 [ ]:

```python
#
484700.7000000002, 4601827.0
484164.2999999998, 4601425.0
484164.2999999998, 4601425.0
484164.2999999998, 4601425.0
484903.2999999998, 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
482971.7000000002, 4602427.0
482816.7000000002, 4602599.0
483486.0, 4601072.0
482983.2999999998, 4602562.0
483349.0, 4602502.0
483534.0, 4601546.0
483509.4000000004, 4601475.0
485077.0, 4599968.0
483662.2999999998, 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 [ ]:

```python
#
OBJECTID
Shape
DATE_
PDT
CO2_
SOIL°C
AIR°C
d_cm
id
loc
ELEV
PM
DESCRIPTIO
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.
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-string 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 [ ]:

```python
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.

```python
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]}")
```
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.

```python
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,(324.04447351342624, 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. 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
    ```python
    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)
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)
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:

```
txpath = ws + "/samples.csv"
if arcpy.Exists(txpath): arcpy.Delete_management(txpath)
txtout = open(txpath, "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]}
")
```

- 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.
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 [ ]:

```python
['co2july95.shp', 'contour.shp', 'geology.shp', 'marblePts.shp', 'mvalley_pts.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 [ ]:

```python
['co2july95.shp', 'contour.shp', 'geology.shp', 'marblePts.shp', 'mvalley_pts.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))

```python
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)`

```python
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)`

```python
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:

```python
import geopandas as gpd
import matplotlib.pyplot as plt
marblepts = gpd.read_file('curdata/' + ptfeatname)
marblepts
```

Then to plot the two points just requires:

```python
marblepts.plot()
```

In [ ]: #

Out[ ]:

<table>
<thead>
<tr>
<th>Id</th>
<th>name</th>
<th>POINT_X</th>
<th>POINT_Y</th>
<th>geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Upper Meadow</td>
<td>483473.0</td>
<td>4601523.0</td>
<td>POINT (483473.000 4601523.000)</td>
</tr>
<tr>
<td>1</td>
<td>Sky High Camp</td>
<td>485339.0</td>
<td>4600001.0</td>
<td>POINT (485339.000 4600001.000)</td>
</tr>
</tbody>
</table>
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.
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 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.s.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.s.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

  ```python
  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`:

  ```python
  firstrow = True
  ```

- The loop will be different: use

  ```python
  for pt in textin:
  ```

  then within the loop, use

  ```python
  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:

  ```python
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.

  ```python
  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`.

```python
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.

```python
import geopandas as gpd
import pandas as pd
import matplotlib.pyplot as plt
samplepts = gpd.read_file('curdata/ptfeatname')
samplepts

column="CATOT",cmap=("YlOrBr")
```

In [ ]:
#

Out[ ]:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>49</td>
<td>0.83</td>
<td>485956.28125</td>
<td>4602944.5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>47</td>
<td>0.83</td>
<td>485589.78125</td>
<td>4602934.0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0.63</td>
<td>485857.50000</td>
<td>4602919.0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>46</td>
<td>0.67</td>
<td>485403.43750</td>
<td>4602791.5</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0.70</td>
<td>485711.37500</td>
<td>4602716.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>56</td>
<td>0</td>
<td>36</td>
<td>0.05</td>
<td>485408.84375</td>
<td>4600386.5</td>
</tr>
<tr>
<td>57</td>
<td>0</td>
<td>35</td>
<td>0.07</td>
<td>485482.43750</td>
<td>4600217.5</td>
</tr>
<tr>
<td>58</td>
<td>0</td>
<td>8</td>
<td>0.05</td>
<td>484966.03125</td>
<td>4600185.5</td>
</tr>
<tr>
<td>59</td>
<td>0</td>
<td>6</td>
<td>0.02</td>
<td>485371.46875</td>
<td>4599995.5</td>
</tr>
<tr>
<td>60</td>
<td>0</td>
<td>7</td>
<td>0.02</td>
<td>484911.09375</td>
<td>4599726.5</td>
</tr>
</tbody>
</table>

61 rows × 6 columns
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: \( p = \text{arcpy.Point}(\text{rnd()} \times \text{xrg} + \text{xmin}, \text{rnd()} \times \text{yrg} + \text{ymin}) \)
- Append the point to `pointlist`

Then create an interior `for` 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 \( \text{xstp} \) and \( \text{ystp} \) distances: \( p = \text{arcpy.Point}(p.X + \text{xstp} \times \text{rnd()} - \text{xstp} \times \text{rnd()}, \ p.Y + \text{ystp} \times \text{rnd()} - \text{ystp} \times \text{rnd()}()) \)
- Append that point to `pointlist` Then create an array from `pointlist`, then make a polyline from it:

\[
\text{array} = \text{arcpy.Array(pointlist)} \\
\text{polyline} = \text{arcpy.Polyline(array)}
\]

Insert that polyline as an inserted row in the cursor

\[
\text{cur.insertRow([i, polyline])}
\]

1. Finally, delete the cursor with `del cur`.

\[
\text{In [ ]: #}
\]

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:

```python
import geopandas as gpd
import matplotlib.pyplot as plt
randomlines = gpd.read_file('curdata/'+ featfile)
randomlines
randomlines.plot(column="Id", cmap="Spectral")
```

file:///C:/Users/900008452/Box/course/625/exer/exercise_html/results/Ex08_arcpyDataCursorsPandas_results.html 27/42
<table>
<thead>
<tr>
<th>Id</th>
<th>geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LINESTRING (484855.538 4600963.158, 484775.111...</td>
</tr>
<tr>
<td>1</td>
<td>LINESTRING (484536.985 4602234.953, 484615.485...</td>
</tr>
<tr>
<td>2</td>
<td>LINESTRING (483574.810 4600782.501, 483610.263...</td>
</tr>
<tr>
<td>3</td>
<td>LINESTRING (485918.059 4602715.329, 485935.273...</td>
</tr>
<tr>
<td>4</td>
<td>LINESTRING (483380.354 4601547.181, 483406.847...</td>
</tr>
<tr>
<td>5</td>
<td>LINESTRING (484018.669 4600813.960, 484086.832...</td>
</tr>
<tr>
<td>6</td>
<td>LINESTRING (482749.179 4600774.847, 482775.303...</td>
</tr>
<tr>
<td>7</td>
<td>LINESTRING (485698.447 4599836.956, 485646.521...</td>
</tr>
<tr>
<td>8</td>
<td>LINESTRING (485237.285 4600636.199, 485286.713...</td>
</tr>
<tr>
<td>9</td>
<td>LINESTRING (483974.995 4601215.661, 484062.971...</td>
</tr>
<tr>
<td>10</td>
<td>LINESTRING (484039.708 4602754.249, 483981.857...</td>
</tr>
<tr>
<td>11</td>
<td>LINESTRING (484808.663 4600900.560, 484918.152...</td>
</tr>
</tbody>
</table>

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.

Geopandas display

Again, only if you're running this outside ArcGIS Pro, you can display your results in Geopandas with:

```python
import geopandas as gpd
import matplotlib.pyplot as plt
randompolys = gpd.read_file('curdata/' + featfile)
randompolys
```
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:

```python
import pandas as pd
import arcpy, os
ws = os.getcwd()
```

```python
In []: #
Out[]: <AxesSubplot: >
```
Loading various data types in a pandas dataframe

Next, we'll want to assign the schools variable to our "SF_Schools" feature class:

```python
schools = ws + r"\SF.gdb"
```

Then let's see what the field names are:

```python
fields = []
for fld in arcpy.ListFields(schools):
    fields.append(fld.name)
fields
```

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

 rowIndex 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]

# 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]
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]
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.

```python
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.

```python
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:

```python
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.

```python
schools = ws + r"\SF.gdb\SF_Schools"
```

Create a numpy array from the feature class

Next we'll use that featureclass path to create a numpy array:

```python
arr = arcpy.da.TableToNumPyArray(schools, ("Name", "District", "City", "Type"))
arr[:3]
```

Create a dataframe from the numpy array

After you've created your NumPy array you can load it into a dataframe using `pd.DataFrame()`.

```python
schools_df = pd.DataFrame(arr)
schools_df
```
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.

```python
import geopandas as gpd
sf_schools = gpd.GeoDataFrame(schools_df, geometry=gpd.points_from_xy(schools_df.X, schools_df.Y), crs="EPSG:4326")
```
We can also reproject it to web mercator to use with a contextily basemap.

```python
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

```python
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)
**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 [ ]:
```