**Week 1: Python types and sequences**

* Define a function
* Python programming language has several types: strings (can be used single or double quotes), NoneType, integer, float, and function
* Python’s built around different kinds of sequences or collection types:
  + Tuples
    - sequence of variables which itself is *immutable*, i.e. items cannot change the order once created
    - write tuples using parentheses ()
    - can mix types for the contents of tuple
    - iterable types, can write loops to go through every value
    - can be accessed as arrays in other languages
    - basic mathematical operations:
      * plus sign concatenates lists (1, 2) + (3, 4) = (1, 2, 3, 4)
      * asterisks repeats the value (1)\*3 = (1, 1, 1)
      * `in` operator returns True or False: 1 in (1, 2, 3) = True
  + Lists
    - sequence of variables which itself is *mutable*
    - write lists using square brackets []
    - can change contents of list through append function
    - iterable types, can write loops to go through every value
    - can be accessed as arrays in other languages
    - basic mathematical operations: plus sign concatenates lists; asterisks repeats the value
      * plus sign concatenates lists [1, 2] + [3, 4] = [1, 2, 3, 4]
      * asterisks repeats the value [1]\*3 = [1, 1, 1]
      * `in` operator returns True or False: 1 in [1, 2, 3] = True
    - most interesting operations are called slicing
      * x = ‘This is a string’  
        print(x[0]) = T  
        print(x[0:1]) = T  
        print(x[0:2]) = Th  
        x[-1] = ‘g’  
        x[-4:-2] = ‘ri’  
        x[:3] = ‘Thi’  
        x[3:] = ‘s is a string’
  + Dictionaries
    - Collection of items that do not have an ordering, i.e. each value you insert into the dictionary, you must give a key to get that value out.
    - Write dictionaries using curly braces {}
    - Example:
      * x={‘Christopher Brooks’: ‘brooksch@umich.edu’, ‘Bill Gates’: ‘billg@microsoft.com’}  
        x{‘Christopher Brook’} = ‘brooksch@umich.edu’
      * We can insert new items:  
        x[‘Kevyn Collins-Thompson’] = None  
        x[‘Kevyn Ccollins-Thompson’]
    - Can iterate over all of the items in a dictionary in several ways
      * Iterate over all of the keys and just pull the contents out as you see fit  
        for name in x:  
         print(x[name])  
        [billg@microsoft.com](mailto:billg@microsoft.com)  
        [brooksch@umich.edu](mailto:brooksch@umich.edu)  
        None
      * Iterate over the values and just ignore the keys  
        for email in x.values():  
         print(email)  
        [billg@microsoft.com](mailto:billg@microsoft.com)  
        [brooksch@umich.edu](mailto:brooksch@umich.edu)  
        None
      * Iterate over both the values and the keys at once using item’s function  
        for name, email in x.items():  
         print(name)  
         print(email)  
        Bill Gates  
        [billg@microsoft.com](mailto:billg@microsoft.com)  
        Christopher Brooks  
        [brooksch@umich.edu](mailto:brooksch@umich.edu)  
        Kevyn Collins-Thompson  
        None
* An example of unpacking in python

x = (‘Christopher’, ‘Brooks’, ‘brooksch@umich.edu’)  
fname, lname, email = x  
fname = ‘Christopher’  
lname = ‘Brooks’

**Split in python**

firstname = 'Christopher Arthur Hansen Brooks'.split(' ')[0] # [0] selects the first element of the list

lastname = 'Christopher Arthur Hansen Brooks'.split(' ')[-1] # [-1] selects the last element of the list

print(firstname)

print(lastname)

people = ['Dr. Christopher Brooks', 'Dr. Kevyn Collins-Thompson', 'Dr. VG Vinod Vydiswaran', 'Dr. Daniel Romero']

def split\_title\_and\_name(person):

    title = person.split()[0]

    lastname = person.split()[-1]

    return '{} {}'.format(title, lastname)

list(map(split\_title\_and\_name, people))

**The Python Programming Language: More on Strings**

>>> print(‘Chris’ + 2)

TypeError: Can’t convert ‘int’ object to str implicitly

>>> print(‘Chris’ + str(2))

Chris2

**Python has a built in method for convenient string formatting**

sales\_record = {

'price': 3.24,

'num\_items': 4,

'person': 'Chris'}

sales\_statement = '{} bought {} item(s) at a price of {} each for a total of {}'

print(sales\_statement.format(sales\_record['person'],

sales\_record['num\_items'],

sales\_record['price'],

sales\_record['num\_items'] \* sales\_record['price']))

**Reading and Writing CSV files**

import csv

% precision 2

with open('datasets/mpg.csv') as csvfile:

mpg = list(csv.DictReader(csvfile)) **# csv.Dictreader has read in each row of our csv file as a dictionary**

mpg[:3] # The first three dictionaries in our list.

len(mpg) # len shows that our list is comprised of 234 dictionaries

mpg[0].keys() **# keys gives us the column names of our csv.**

sum(float(d['cty']) for d in mpg) / len(mpg) **# find average cty fuel economy across all cars**

sum(float(d['hwy']) for d in mpg) / len(mpg)

cylinders = set(d['cyl'] for d in mpg) # **Use set to return the unique values for the number of cylinders the cars in our dataset have**

cylinders

**Complex example where we are grouping the cars by number of cylinder, and finding the average cty mpg for each group**

CtyMpgByCyl = []

for c in cylinders: # iterate over all the cylinder levels

summpg = 0

cyltypecount = 0

for d in mpg: # iterate over all dictionaries

if d['cyl'] == c: # if the cylinder level type matches,

summpg += float(d['cty']) # add the cty mpg

cyltypecount += 1 # increment the count

CtyMpgByCyl.append((c, summpg / cyltypecount)) # append the tuple ('cylinder', 'avg mpg')

CtyMpgByCyl.sort(key=lambda x: x[0]) # sort the list from lowest to highest cylinder

CtyMpgByCyl

**Another example**

vehicleclass = set(d['class'] for d in mpg) # what are the class types

vehicleclass

HwyMpgByClass = []

for t in vehicleclass: # iterate over all the vehicle classes

summpg = 0

vclasscount = 0

for d in mpg: # iterate over all dictionaries

if d['class'] == t: # if the cylinder amount type matches,

summpg += float(d['hwy']) # add the hwy mpg

vclasscount += 1 # increment the count

HwyMpgByClass.append((t, summpg / vclasscount)) # append the tuple ('class', 'avg mpg')

HwyMpgByClass.sort(key=lambda x: x[1])

HwyMpgByClass

**Python programming language: Dates and Times**

import datetime as dt

import time as tm

tm.time() # `time` returns the current time in seconds since the Epoch. (January 1st, 1970)

dtnow = dt.datetime.fromtimestamp(tm.time()) # Convert the timestamp to datetime.

dtnow

dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second # get year, month, day, etc.from a datetime

delta = dt.timedelta(days=100) # create a timedelta of 100 days

delta # timedelta is a duration expressing the difference between two dates.

today = dt.date.today() # date.today returns the current local date

today – delta # the date 100 days ago

today > today – delta # compare dates

**Python Programming Language: Objects and map()**

**An example of class in python**

* Class variables are variables which are shared across all instances.
* To define a method, you just write it as you would have a function.
* The one change, is that to have access to the instance which a method is being invoked upon, you must include ‘self’ in the method signature
* If you want to refer to instance variables set on the object, you prepend them with the word self, with a full stop, e.g. self.name

class Person:

department = 'School of Information' #a class variable

def set\_name(self, new\_name): #a method

self.name = new\_name

def set\_location(self, new\_location):

self.location = new\_location

Implications of object-oriented programming in python

* Objects in Python do not have private or protected members. If you instantiate an object, you have full access to any of the methods or attributes of that object.
* There’s no need for an explicit constructor when creating objects in python. You can add a constructor if you want to by declaring the \_\_init\_\_ method

**The map function in Python**

* map built-in function is one example of a functional programming feature of python, that ties together a number of aspects of the language
* map(function, iterable, …): the first parameter is function that you want executed; the second parameter is something which can be iterated upon.
  + all iterable arguments are unpacked together, and passed into the given function.
* Example:

store1 = [10.00, 11.00, 12.34, 2.34]

store2 = [9.00, 11.10, 12.34, 2.01]

cheapest = map(min, store1, store2)

cheapest

for item in cheapest:

print(item)

**The lambda: anonymous function**

people = ['Dr. Christopher Brooks', 'Dr. Kevyn Collins-Thompson', 'Dr. VG Vinod Vydiswaran', 'Dr. Daniel Romero']

def split\_title\_and\_name(person):

    return person.split()[0] + ' ' + person.split()[-1]

#option 1

for person in people:

    print(split\_title\_and\_name(person) == (lambda x: x.split()[0] + ' ' + x.split()[-1])(person))

#option 2

list(map(split\_title\_and\_name, people)) == list(map(lambda person: person.split()[0] + ' ' + person.split()[-1], people))

**Lambda examples**

my\_function = lambda a, b, c: a + b

my\_function(1, 2, 3)

**Let's iterate from 0 to 999 and return the even numbers.**

my\_list = []

for number in range(0, 1000):

if number % 2 == 0:

my\_list.append(number)

my\_list

**Now the same thing but with list comprehension.**

my\_list = [number for number in range(0, 1000) if number % 2 == 0]

my\_list

def times\_tables():

    lst = []

    for i in range(10):

        for j in range (10):

            lst.append(i\*j)

    return lst

times\_tables() == [j\*i for i in range(10) for j in range(10)]

lowercase = 'abcdefghijklmnopqrstuvwxyz'

digits = '0123456789'

correct\_answer = [a+b+c+d for a in lowercase for b in lowercase for c in digits for d in digits]

correct\_answer[:50] # Display first 50 ids

**Numerical python library (NumPy)**

* Fundamental package for numeric computing with Python
* It provides powerful ways to create, store, and manipulate data 🡪 seamlessly and speedily integrate with a wide variety of databases and data formats.
* It’s foundation that Pandas is built on which is a high performance data-centric package.
* You'll recall that we import a library using the `import` keyword as numpy's common abbreviation is np

import numpy as np

import math

* Arrays are displayed as a list or list of lists and can be created through list as well. When creating an array, we pass in a list as an argument in numpy array

a = np.array([1, 2, 3])

print(a)

* We can print the number of dimensions of a list using the ndim attribute

print(a.ndim)

* We can print out the length of each dimension by calling the shape attribute, which returns a tuple

b.shape

* We can also check the type of items in the array

a.dtype

* Besides integers, floats are also accepted in numpy arrays

c = np.array([2.2, 5, 1.1])

c.dtype.name

* Note that numpy automatically converts integers, like 5, up to floats, since there is no loss of prescision.
* Numpy will try and give you the best data type format possible to keep your data types homogeneous, which means all the same, in the array
* Sometimes we know the shape of an array that we want to create, but not what we want to be in it. numpy offers several functions to create arrays with initial placeholders, such as zero's or one's.
* Lets create two arrays, both the same shape but with different filler values

d = np.zeros((2,3))

print(d)

e = np.ones((2,3))

print(e)

* We can also generate an array with random numbers

np.random.rand(2,3)

* You'll see zeros, ones, and rand used quite often to create example arrays, especially in stack overflow posts and other forums.
* We can also create a sequence of numbers in an array with the arrange() function. The fist argument is the starting bound and the second argument is the ending bound, and the third argument is the difference between each consecutive numbers
* Let's create an array of every even number from ten (inclusive) to fifty (exclusive)

f = np.arange(10, 50, 2)

f

* If we want to generate a sequence of floats, we can use the linspace() function. In this function the third argument isn't the difference between two numbers, but the total number of items you want to generate

np.linspace( 0, 2, 15 ) # 15 numbers from 0 (inclusive) to 2 (inclusive)

**Arithmetic operators on array apply elementwise.**

* Let's create a couple of arrays

a = np.array([10,20,30,40])

b = np.array([1, 2, 3,4])

* Now let's look at a minus b

c = a-b

print(c)

* And let's look at a times b

d = a\*b

print(d)

* With arithmetic manipulation, we can convert current data to the way we want it to be. Here's a real-world problem I face - I moved down to the United States about 6 years ago from Canada. In Canada we use celcius for temperatures, and my wife still hasn't converted to the US system which uses farenheit.
* With numpy I could easily convert a number of farenheit values, say the weather forecase, to ceclius
* Let's create an array of typical Ann Arbor winter farenheit values

farenheit = np.array([0,-10,-5,-15,0])

* And the formula for conversion is ((°F − 32) × 5/9 = °C)

celcius = (farenheit - 31) \* (5/9)

celcius

* Another useful and important manipulation is the boolean array. We can apply an operator on an array, and a boolean array will be returned for any element in the original, with True being emitted if it meets the condition and False otherwise.
* For instance, if we want to get a boolean array to check celcius degrees that are greater than -20 degrees

celcius > -20

* Here's another example, we could use the modulus operator to check numbers in an array to see if they are even. Recall that modulus does division but throws away everything but the remainder (decimal) portion)

celcius%2 == 0

* Besides elementwise manipulation, it is important to know that numpy supports matrix manipulation. Let's look at matrix product.
* If we want to do elementwise product, we use the "\*" sign

A = np.array([[1,1],[0,1]])

B = np.array([[2,0],[3,4]])

print(A\*B)

* If we want to do matrix product, we use the "@" sign or use the dot function

print(A@B)

* You don't have to worry about complex matrix operations for this course, but it's important to know that numpy is the underpinning of scientific computing libraries in python, and that it is capable of doing both element-wise operations (the asterix) as well as matrix-level operations (the @ sign). There's more on this in a subsequent course.
* A few more linear algebra concepts are worth layering in here. You might recall that the product of two matrices is only plausible when the inner dimensions of the two matrices are the same. The dimensions refer to the number of elements both horizontally and vertically in the rendered matricies you've seen here.
* We can use numpy to quickly see the shape of a matrix:

A.shape

* When manipulating arrays of different types, the type of the resulting array will correspond to the more general of the two types. This is called upcasting.
* Let's create an array of integers

array1 = np.array([[1, 2, 3], [4, 5, 6]])

print(array1.dtype)

* Now let's create an array of floats

array2 = np.array([[7.1, 8.2, 9.1], [10.4, 11.2, 12.3]])

print(array2.dtype)

* Integers (int) are whole numbers only, and Floating point numbers (float) can have a whole number portion and a decimal portion.
* The 64 in this example refers to the number of bits that the operating system is reserving to represent the number, which determines the size (or precision) of the numbers that can be represented.
* Numpy arrays have many interesting aggregation functions on them, such as sum(), max(), min(), and mean()

print(array3.sum())

print(array3.max())

print(array3.min())

print(array3.mean())

* For two dimensional arrays, we can do the same thing for each row or column
* Let's create an array with 15 elements, ranging from 1 to 15, with a dimension of 3X5

b = np.arange(1,16,1).reshape(3,5)

print(b)

* Now, we often think about two dimensional arrays being made up of rows and columns, but you can also think of these arrays as just a giant ordered list of numbers, and the \*shape\* of the array, the number of rows and columns, is just an abstraction that we have for a particular purpose.
* Actually, this is exactly how basic images are stored in computer environments.
* Let's take a look at an example and see how numpy comes into play.
* For this demonstration I'll use the python imaging library (PIL) and a function to display images in the Jupyter notebook

from PIL import Image

from IPython.display import display

* And let's just look at the image I'm talking about

im = Image.open('../chris.tiff')

display(im)

* Now, we can conver this PIL image to a numpy array

array=np.array(im)

print(array.shape)

array

* Here we see that we have a 200x200 array and that the values are all uint8. The uint means that they are unsigned integers (so no negative numbers) and the 8 means 8 bits per byte.
* This means that each value can be up to 2\*2\*2\*2\*2\*2\*2\*2=256 in size (well, actually 255, because we start at zero).
* For black and white images black is stored as 0 and white is stored as 255. So if we just wanted to invert this image we could use the numpy array to do so
* Let's create an array the same shape

mask=np.full(array.shape,255)

mask

* Now let's subtract that from the modified array

modified\_array=array-mask

* And lets convert all of the negative values to positive values

modified\_array=modified\_array\*-1

* And as a last step, let's tell numpy to set the value of the datatype correctly

modified\_array=modified\_array.astype(np.uint8)

modified\_array

**Indexing, Slicing and Iterating**

**Indexing**

* First we are going to look at integer indexing. A one-dimensional array, works in similar ways as a list.
* To get an element in a one-dimensional array, we simply use the offset index.

a = np.array([1,3,5,7])

a[2]

* For multidimensional array, we need to use integer array indexing, let's create a new multidimensional array

a = np.array([[1,2], [3, 4], [5, 6]])

a

* If we want to select one certain element, we can do so by entering the index, which is comprised of two integers the first being the row, and the second the column

a[1,1] # remember in python we start at 0!

* If we want to get multiple elements for example, 1, 4, and 6 and put them into a one-dimensional array, we can enter the indices directly into an array function

np.array([a[0, 0], a[1, 1], a[2, 1]])

* We can also do that by using another form of array indexing, which essential "zips" the first list and the second list up

print(a[[0, 1, 2], [0, 1, 1]])

**Slicing**

* Slicing is a way to create a sub-array based on the original array. For one-dimensional arrays, slicing works in similar ways to a list.
* To slice, we use the : sign. For instance, if we put :3 in the indexing brackets, we get elements from index 0 to index 3 (excluding index 3)

a = np.array([0,1,2,3,4,5])

print(a[:3])

* By putting 2:4 in the bracket, we get elements from index 2 to index 4 (excluding index 4)

print(a[2:4])

* For multi-dimensional arrays, it works similarly, let’s see an example

a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

a

* First, if we put one argument in the array, for example a[:2] then we would get all the elements from the first (0th) and second row (1th)

a[:2]

* If we add another argument to the array, for example a[:2, 1:3], we get the first two rows but then the second and third column values only

a[:2, 1:3]

* So, in multidimensional arrays, the first argument is for selecting rows, and the second argument is for selecting columns
* It is important to realize that a slice of an array is a view into the same data. This is called passing by reference.
* So modifying the sub array will consequently modify the original array
* Here I'll change the element at position [0, 0], which is 2, to 50, then we can see that the value in the original array is changed to 50 as well

sub\_array = a[:2, 1:3]

print("sub array index [0,0] value before change:", sub\_array[0,0])

sub\_array[0,0] = 50

print("sub array index [0,0] value after change:", sub\_array[0,0])

print("original array index [0,1] value after change:", a[0,1])

**Trying Numpy with Datasets**

* To load a dataset in Numpy, we can use the genfromtxt() function. We can specify data file name, delimiter (which is optional but often used), and number of rows to skip if we have a header row, hence it is 1 here
* The genfromtxt() function has a parameter called dtype for specifying data types of each column this parameter is optional. Without specifying the types, all types will be casted the same to the more general/precise type

wines = np.genfromtxt("datasets/winequality-red.csv", delimiter=";", skip\_header=1)

wines

* Recall that we can use integer indexing to get a certain column or a row. For example, if we want to select the fixed acidity column, which is the first column, we can do so by entering the index into the array.
* Also remember that for multidimensional arrays, the first argument refers to the row, and the second argument refers to the column, and if we just give one argument then we'll get a single dimensional list back.
* So all rows combined but only the first column from them would be

print("one integer 0 for slicing: ", wines[:, 0])

* But if we wanted the same values but wanted to preserve that they sit in their own rows we would write

print("0 to 1 for slicing: \n", wines[:, 0:1])

* This is another great example of how the shape of the data is an abstraction which we can layer intentionally on top of the data we are working with.
* If we want a range of columns in order, say columns 0 through 3 (recall, this means first, second, and third, since we start at zero and don't include the training index value), we can do that too

wines[:, 0:3]

* What if we want several non-consecutive columns? We can place the indices of the columns that we want into an array and pass the array as the second argument. Here's an example

wines[:, [0,2,4]]

* We can also do some basic summarization of this dataset. For example, if we want to find out the average quality of red wine, we can select the quality column.
* We could do this in a couple of ways, but the most appropriate is to use the -1 value for the index, as negative numbers mean slicing from the back of the list.
* We can then call the aggregation functions on this data.

wines[:,-1].mean()

* Let's take a look at another dataset, this time on graduate school admissions. It has fields such as GRE score, TOEFL score, university rating, GPA, having research experience or not, and a chance of admission.
* With this dataset, we can do data manipulation and basic analysis to infer what conditions are associated with higher chance of admission. Let's take a look.
* We can specify data field names when using genfromtxt() to loads CSV data. Also, we can have numpy try and infer the type of a column by setting the dtype parameter to None

graduate\_admission = np.genfromtxt('datasets/Admission\_Predict.csv', dtype=None, delimiter=',', skip\_header=1,

names=('Serial No','GRE Score', 'TOEFL Score', 'University Rating', 'SOP',

'LOR','CGPA','Research', 'Chance of Admit'))

graduate\_admission

* Notice that the resulting array is actually a one-dimensional array with 400 tuples

graduate\_admission.shape

* We can retrieve a column from the array using the column's name for example, let's get the CGPA column and only the first five values.

graduate\_admission['CGPA'][0:5]

* Since the GPA in the dataset range from 1 to 10, and in the US it's more common to use a scale of up to 4, a common task might be to convert the GPA by dividing by 10 and then multiplying by 4

graduate\_admission['CGPA'] = graduate\_admission['CGPA'] /10 \*4

graduate\_admission['CGPA'][0:20] #let's get 20 values

* Recall boolean masking. We can use this to find out how many students have had research experience by creating a boolean mask and passing it to the array indexing operator

len(graduate\_admission[graduate\_admission['Research'] == 1])

* Since we have the data field chance of admission, which ranges from 0 to 1, we can try to see if students with high chance of admission (>0.8) on average have higher GRE score than those with lower chance of admission (<0.4)
* So first we use boolean masking to pull out only those students we are interested in based on their chance of admission, then we pull out only their GPA scores, then we print the mean values.

print(graduate\_admission[graduate\_admission['Chance\_of\_Admit'] > 0.8]['GRE\_Score'].mean())

print(graduate\_admission[graduate\_admission['Chance\_of\_Admit'] < 0.4]['GRE\_Score'].mean())

* Take a moment to reflect here, do you understand what is happening in these calls?
* When we do the boolean masking we are left with an array with tuples in it still, and numpy holds underneath
* This a list of the columns we specified and their name and indexes

graduate\_admission[graduate\_admission['Chance\_of\_Admit'] > 0.8]

* Let's also do this with GPA

print(graduate\_admission[graduate\_admission['Chance\_of\_Admit'] > 0.8]['CGPA'].mean())

print(graduate\_admission[graduate\_admission['Chance\_of\_Admit'] < 0.4]['CGPA'].mean())

**Manipulating Text with Regular Expression**

* First we'll import the re module, which is where python stores regular expression libraries.

import re

* There are several main processing functions in re that you might use. The first, match() checks for a match that is at the beginning of the string and returns a boolean.
* Similarly, search(), checks for a match anywhere in the string, and returns a boolean.
* Let’s create some text for an example

text = "This is a good day."

* Now, let’s see if it's a good day or not:

if re.search("good", text): # the first parameter here is the pattern

print("Wonderful!")

else:

print("Alas :(")

* In addition to checking for conditionals, we can segment a string.
* The work that regex does here is called tokenizing, where the string is separated into substrings based on patterns.
* Tokenizing is a core activity in natural language processing, which we won't talk much about here but that you will study in the future
* The findall() and split() functions will parse the string for us and return chunks. Let’s try and example

text = "Amy works diligently. Amy gets good grades. Our student Amy is succesful."

* This is a bit of a fabricated example, but let’s split this on all instances of Amy

re.split("Amy", text)

* You'll notice that split has returned an empty string, followed by a number of statements about Amy, all as elements of a list.
* If we wanted to count how many times we have talked about Amy, we could use findall()

re.findall("Amy", text)

* Ok, so we've seen that .search() looks for some pattern and returns a boolean, that .split() will use a pattern for creating a list of substrings, and that .findall() will look for a pattern and pull out all occurrences.
* Now that we know how the python regex API works, let’s talk about more complex patterns.
* The regex specification standard defines a markup language to describe patterns in text.
* Let’s start with anchors. Anchors specify the start and/or the end of the string that you are trying to match.
* The caret character ^ means start and the dollar sign character $ means end. If you put ^ before a string, it means that the text the regex processor retrieves must start with the string you specify.
* For ending, you have to put the $ character after the string, it means that the text Regex retrieves must end with the string you specify.
* Here's an example

text = "Amy works diligently. Amy gets good grades. Our student Amy is successful."

* Let’s see if this begins with Amy

re.search("^Amy",text)

* Notice that re.search() actually returned to us a new object, called re.Match object.
* An re.Match object always has a boolean value of True, as something was found, so you can always evaluate it in an if statement as we did earlier.
* The rendering of the match object also tells you what pattern was matched, in this case the word Amy, and the location the match was in, as the span.

**Patterns and Character Classes**

* Let's talk more about patterns and start with character classes. Let's create a string of a single learners' grades over a semester in one course across all of their assignments

grades="ACAAAABCBCBAA"

* If we want to answer the question "How many B's were in the grade list?" we would just use B

re.findall("B",grades)

* If we wanted to count the number of A's or B's in the list, we can't use "AB" since this is used to match all A's followed immediately by a B. Instead, we put the characters A and B inside square brackets

re.findall("[AB]",grades)

* This is called the set operator. You can also include a range of characters, which are ordered alphanumerically.
* For instance, if we want to refer to all lower case letters we could use [a-z]
* Let’s build a simple regex to parse out all instances where this student receive an A followed by a B or a C

re.findall("[A][B-C]",grades)

* Notice how the [AB] pattern describes a set of possible characters which could be either (A OR B), while the [A][B-C] pattern denoted two sets of characters which must have been matched back to back.
* You can write this pattern by using the pipe operator, which means OR

re.findall("AB|AC",grades)

* We can use the caret with the set operator to negate our results.
* For instance, if we want to parse out only the grades which were not A's

re.findall("[^A]",grades)

* Note this carefully - the caret was previously matched to the beginning of a string as an anchor point, but inside of the set operator the caret, and the other special characters we will be talking about, lose their meaning.
* This can be a bit confusing. What do you think the result would be of this?

re.findall("^[^A]",grades)

* It's an empty list, because the regex says that we want to match any value at the beginning of the string which is not an A.
* Our string though starts with an A, so there is no match found.
* And remember when you are using the set operator you are doing character-based matching. So you are matching individual characters in an OR method.

**Quantifiers**

* Ok, so we've talked about anchors and matching to the beginning and end of patterns. And we've talked about characters and using sets with the [] notation.
* We've also talked about character negation, and how the pipe| character allows us to or operations.
* Let’s move on to quantifiers.
* Quantifiers are the number of times you want a pattern to be matched in order to match.
* The most basic quantifier is expressed as e{m,n}, where e is the expression or character we are matching, m is the minimum number of times you want it to matched, and n is the maximum number of times the item could be matched.
* Let's use these grades as an example. How many times has this student been on a back-to-back A's streak?

re.findall("A{2,10}",grades) # we'll use 2 as our min, but ten as our max

* So we see that there were two streaks, one where the student had four A's, and one where they had only two A's
* We might try and do this using single values and just repeating the pattern

re.findall("A{1,1}A{1,1}",grades)

* As you can see, this is different than the first example. The first pattern is looking for any combination of two A's up to ten A's in a row.
* So it sees four A's as a single streak. The second pattern is looking for two A's back to back, so it sees two A's followed immediately by two more A's.
* We say that the regex processor begins at the start of the string and consumes variables which match patterns as it does.
* It's important to note that the regex quantifier syntax does not allow you to deviate from the {m,n} pattern.
* In particular, if you have an extra space in between the braces you'll get an empty result

re.findall("A{2, 2}",grades)

* And as we have already seen, if we don't include a quantifier then the default is {1,1}

re.findall("AA",grades)

* Oh, and if you just have one number in the braces, it's considered to be both m and n

re.findall("A{2}",grades)

* Using this, we could find a decreasing trend in a student's grades

re.findall("A{1,10}B{1,10}C{1,10}",grades)

* Now, that's a bit of a hack, because we included a maximum that was just arbitrarily large.
* There are three other quantifiers that are used as short hand, an asterix \* to match 0 or more times, a question mark ? to match one or more times, or a + plus sign to match one or more times.
* Let’s look at a more complex example, and load some data scraped from wikipedia

with open("datasets/ferpa.txt","r") as file:

* We'll read that into a variable called wiki

wiki=file.read()

* And let’s print that variable out to the screen

wiki

* Scanning through this document one of the things we notice is that the headers all have the words [edit] behind them, followed by a newline character.
* So if we wanted to get a list of all of the headers in this article we could do so using re.findall

re.findall("[a-zA-Z]{1,100}\[edit\]",wiki)

* Ok, that didn't quite work. It got all of the headers, but only the last word of the header, and it really was quite clunky.
* Let’s iteratively improve this.
* First, we can use \w to match any letter, including digits and numbers.

re.findall("[\w]{1,100}\[edit\]",wiki)

* This is something new. \w is a metacharacter, and indicates a special pattern of any letter or digit.
* There are actually a number of different metacharacters listed in the documentation.
* For instance, \s matches any whitespace character.
* Next, there are three other quantifiers we can use which shorten up the curly brace syntax. We can use an asterix \* to match 0 or more times, so let's try that.

re.findall("[\w]\*\[edit\]",wiki)

* Now that we have shortened the regex, let's improve it a little bit. We can add in a spaces using the space character

re.findall("[\w ]\*\[edit\]",wiki)

* Ok, so this gets us the list of section titles in the wikipedia page! You can now create a list of titles by iterating through this and applying another regex

for title in re.findall("[\w ]\*\[edit\]",wiki):

* Now we will take that intermediate result and split on the square bracket [ just taking the first result

print(re.split("[\[]",title)[0])

**Groups**

* Ok, this works, but it's a bit of a pain. To this point we have been talking about a regex as a single pattern which is matched.
* But, you can actually match different patterns, called groups, at the same time, and then refer to the groups you want.
* To group patterns together you use parentheses, which is actually pretty natural.
* Let’s rewrite our findall using groups

re.findall("([\w ]\*)(\[edit\])",wiki)

* Nice - we see that the python re module breaks out the result by group. We can actually refer to groups by number as well with the match objects that are returned.
* But, how do we get back a list of match objects?
* Thus far we've seen that findall() returns strings, and search() and match() return individual Match objects. But what do we do if we want a list of Match objects? In this case, we use the function finditer()

for item in re.finditer("([\w ]\*)(\[edit\])",wiki):

print(item.groups())

* We see here that the groups() method returns a tuple of the group. We can get an individual group using group(number), where group(0) is the whole match, and each other number is the portion of the match we are interested in. In this case, we want group(1)

for item in re.finditer("([\w ]\*)(\[edit\])",wiki):

print(item.group(1))

* One more piece to regex groups that I rarely use but is a good idea is labeling or naming groups.
* In the previous example I showed you how you can use the position of the group. But giving them a label and looking at the results as a dictionary is pretty useful.
* For that we use the syntax (?P<name>), where the parenthesis starts the group, the ?P indicates that this is an extension to basic regexes, and <name> is the dictionary key we want to use wrapped in <>.

for item in re.finditer("(?P<title>[\w ]\*)(?P<edit\_link>\[edit\])",wiki):

* We can get the dictionary returned for the item with .groupdict()

print(item.groupdict()['title'])

* Of course, we can print out the whole dictionary for the item too, and see that the [edit] string is still in there. Here's the dictionary kept for the last match

print(item.groupdict())

* Ok, we have seen how we can match individual character patterns with [], how we can group matches together using (), and how we can use quantifiers such as \*, ?, or m{n} to describe patterns.
* Something I glossed over in the previous example was the \w, which standards for any word character.
* There are a number of short hands which are used with regexes for different kinds of characters, including: a . for any single character which is not a newline, a \d for any digit, and \s for any whitespace character, like spaces and tabs
* There are more, and a full list can be found in the python documentation for regexes

**Look-ahead and Look-behind**

* One more concept to be familiar with is called "look ahead" and "look behind" matching. In this case, the pattern being given to the regex engine is for text either before or after the text we are trying to isolate.
* For example, in our headers we want to isolate text which comes before the [edit] rendering, but we actually don't care about the [edit] text itself.
* Thus far we have been throwing the [edit] away, but if we want to use them to match but don't want to capture them we could put them in a group and use look ahead instead with ?= syntax

for item in re.finditer("(?P<title>[\w ]+)(?=\[edit\])",wiki):

* What this regex says is match two groups, the first will be named and called title, will have any amount of whitespace or regular word characters, the second will be the characters [edit] but we don't actually want this edit put in our output match objects

print(item)

**Example: Wikipedia Data**

* Let's look at some more wikipedia data. Here's some data on universities in the US which are buddhist-based

with open("datasets/buddhist.txt","r") as file:

* We'll read that into a variable called wiki

wiki=file.read()

* and lets print that variable out to the screen

wiki

* We can see that each university follows a fairly similar pattern, with the name followed by an – then the words "located in" followed by the city and state
* I'll actually use this example to show you the verbose mode of python regexes. The verbose mode allows you to write multi-line regexes and increases readability.
* For this mode, we have to explicitly indicate all whitespace characters, either by prepending them with a \ or by using the \s special value.
* However, this means we can write our regex a bit more like code, and can even include comments with #

pattern="""

(?P<title>.\*) #the university title

(–\ located\ in\ ) #an indicator of the location

(?P<city>\w\*) #city the university is in

(,\ ) #separator for the state

(?P<state>\w\*) #the state the city is located in"""

* Now when we call finditer() we just pass the re.VERBOSE flag as the last parameter, this makes it much easier to understand large regexes!

for item in re.finditer(pattern,wiki,re.VERBOSE):

* We can get the dictionary returned for the item with .groupdict()

print(item.groupdict())

**Example: New York Times and Hashtags**

* Here's another example from the New York Times which covers health tweets on news items. This data came from the UC Irvine Machine Learning Repository which is a great source of different kinds of data

with open("datasets/nytimeshealth.txt","r") as file:

* We'll read everything into a variable and take a look at it

health=file.read()

health

* So here we can see there are tweets with fields separated by pipes |.
* Let’s try and get a list of all of the hashtags that are included in this data. A hashtag begins with a pound sign (or hash mark) and continues until some whitespace is found
* So let’s create a pattern. We want to include the hash sign first, then any number of alphanumeric characters.
* And we end when we see some whitespace

pattern = '#[\w\d]\*(?=\s)'

* Notice that the ending is a look ahead. We're not actually interested in matching whitespace in the return value.
* Also notice that I use an asterix \* instead of the plus + for the matching of alphabetical characters or digits, because a + would require at least one of each
* Let’s searchg and display all of the hashtags

re.findall(pattern, health)

* We can see here that there were lots of ebola related tweeks in this particular dataset.

**REGEX (Regular Expression)**

* Used for pattern matching or string matching

|  |  |
| --- | --- |
| [a,b,c] | a, b, or c |
| [^abc] | any characters except a, b, c |
| [a-z] | a to z |
| [A-Z] | A to Z |
| [a-z A-Z] | a to z, A to Z |
| [0-9] | 0 to 9 |

|  |  |
| --- | --- |
| [ ]? | occurs 0 or 1 time |
| [ ]+ | occurs 1 or more times |
| [ ]\* | occurs 0 or more times |
| [ ]{n} | occurs *n* times |
| [ ]{n,} | occurs *n* or more times |
| [ ]{y,z} | occurs at least *y* times but less than *z* times |

|  |  |
| --- | --- |
| Regex | Metacharacters |
| \d | [0-9] |
| \D | [^0-9] |
| \w | [a-z A-Z \_ 0-9] |
| \W | [^\w] |

\ tells computer to treat the following character as search character, for `+`, `-`

Examples:

1. Mobile number start with 8 or 9 and total digits = 10.  
   [89][0-9]{9}
2. First character, uppercase, contains lower case alphabets, only one digit allowed in between.

[A-Z] [a-z]\*[0-9][a-z]\*

1. Email ID.

[a-z A-Z 0-9 \_ \- \.]+[@][a-z]+[\.][a-z]{2,3}

(?:abc): treats “abc” as a group, but not make re.findall only return the content in the group.

What is the correct regular expression to match a URL with letters, numbers, underscores and dots? A valid URL defined in this problem must meet the following requirements:

1. The URL consists of two or more strings made of letters, numbers, and underscores.
2. A dot is used in between the strings.
3. No two dots are allowed to appear consecutively.

For example, your regex should match URLs like: www.aBC.com, abc.com, ab\_c.de8f.com  
But your regex should not match: abc, abc..com

Answer: (\w+\.)+\w+

What is the correct regular expression to match an ISBN number from two publishers (World Scientific from Singapore, and Sigma Publications from Greece)? A valid ISBN code defined in this problem must meet the following requirements:

1. The ISBN number consists of 10 digits, with dashes(-) in between.
2. The ISBN number must match the patterns of one of the following publishers(x means a digit from 0 to 9): for World Scientific, the pattern should be xxxx-x-xxxx-x, and for Sigma Publications, the pattern should be xxx-xxx-xxx-x.

For example, your regex should match ISBNs like: 9971-5-0210-0, 960-425-059-0

Answer: \d{4}-\d-\d{4}-\d|\d{3}-\d{3}-\d{3}-\d

What is the correct regular expression to match a DOI registered by Crossref? A valid DOI(e.g. doi:10.1038/nphys1170) defined in this problem must meet the following requirements:

1. The DOI starts with **doi:**
2. The link has two parts divided by a “/”. In the first part, there can only be numbers and dots, and in the second part, there can be any characters. There should be at least one character in each part.

For example, your regex should match DOIs like: doi:10.1038/nphys1170, doi:10.1002/0470841559.ch1

Answer: doi:[\d.]+/.+

(?<=https:\/\/): match the character https literally

When ^ is used outside square brackets, it denotes that the expression inside the brackets should not be extracted from the string

‘^[AC]’ vs. ‘^AC’: negate vs. beginning of the string

**Week 2: Introduction to Pandas and Series Data**

* Series is one of the core data structures in pandas
* It’s a cross between a list and dictionary
* The items are all stored in an order and there’s labels with which you can retrieve them
* A way to visualize this is two columns of data
  + The first is the special index, a lot like keys in a dictionary.
  + The second is your actual data.

**Pandas series example: students enrolled in classes coming from a dictionary**

* A pandas Series can be queried either by the index position or the index label. If you don't give an index to the series when querying, the position and the label are effectively the same values.
* To query by numeric location, starting at zero, use the iloc attribute. To query by the index label, you can use the loc attribute.

# Let’s start with an example. We'll use students enrolled in classes coming from a dictionary

import pandas as pd

students\_classes = {'Alice': 'Physics',

'Jack': 'Chemistry',

'Molly': 'English',

'Sam': 'History'}

s = pd.Series(students\_classes)

s

* So, for this series, if you wanted to see the fourth entry we would we would use the iloc attribute with the parameter 3.

s.iloc[3]

* If you wanted to see what class Molly has, we would use the loc attribute with a parameter of Molly.

s.loc['Molly']

* Keep in mind that iloc and loc are not methods, they are attributes. So you don't use parentheses to query them, but square brackets instead, which is called the indexing operator.
* In Python this calls get or set for an item depending on the context of its use.
* Pandas tries to make our code a bit more readable and provides a sort of smart syntax using the indexing operator directly on the series itself.
* For instance, if you pass in an integer parameter, the operator will behave as if you want it to query via the iloc attribute

s[3]

* If you pass in an object, it will query as if you wanted to use the label based loc attribute.

s['Molly']

* So what happens if your index is a list of integers? This is a bit complicated and Pandas can't determine automatically whether you're intending to query by index position or index label.
* So you need to be careful when using the indexing operator on the Series itself. The safer option is to be more explicit and use the iloc or loc attributes directly.
* Here's an example using class and their classcode information, where classes are indexed by classcodes, in the form of integers

class\_code = {99: 'Physics',

100: 'Chemistry',

101: 'English',

102: 'History'}

s = pd.Series(class\_code)

* If we try and call s[0] we get a key error because there's no item in the classes list with an index of zero, instead we have to call iloc explicitly if we want the first item.

s[0]

* So, that didn't call s.iloc[0] underneath as one might expect, instead it generates an error.
* Pandas and the underlying numpy libraries support a method of computation called vectorization.
* Vectorization works with most of the functions in the numpy library, including the sum function.
* Here's how we would really write the code using the numpy sum method. First we need to import the numpy module

import numpy as np

* Then we just call np.sum and pass in an iterable item. In this case, our panda series.

total = np.sum(grades)

print(total/len(grades))

* Now both of these methods create the same value, but is one actually faster? The Jupyter Notebook has a magic function which can help.
* First, let's create a big series of random numbers. This is used a lot when demonstrating techniques with Pandas

numbers = pd.Series(np.random.randint(0,1000,10000))

* Now let’s look at the top five items in that series to make sure they actually seem random. We can do this with the head() function

numbers.head()

* Ok, we're confident now that we have a big series. The ipython interpreter has something called magic functions begin with a percentage sign.
* If we type this sign and then hit the Tab key, you can see a list of the available magic functions. You could write your own magic functions too, but that's a little bit outside of the scope of this course.
* Here, we're actually going to use what's called a cellular magic function. These start with two percentage signs and wrap the code in the current Jupyter cell. The function we're going to use is called timeit. This function will run our code a few times to determine, on average, how long it takes.
* Let's run timeit with our original iterative code. You can give timeit the number of loops that you would like to run.
* By default, it is 1,000 loops. I'll ask timeit here to use 100 runs because we're recording this.
* Note that in order to use a cellular magic function, it has to be the first line in the cell

%%timeit -n 100

total = 0

for number in numbers:

total+=number

total/len(numbers)

%%timeit -n 100

total = np.sum(numbers)

total/len(numbers)

* And now let’s just increase everything in the series by 2

numbers+=2

numbers.head()

* Up until now I've shown only examples of a series where the index values were unique. I want to end this lecture by showing an example where index values are not unique, and this makes pandas Series a little different conceptually then, for instance, a relational database.
* Let’s create a Series with students and the courses which they have taken

students\_classes = pd.Series({'Alice': 'Physics',

'Jack': 'Chemistry',

'Molly': 'English',

'Sam': 'History'})

students\_classes

* Now let’s create a Series just for some new student Kelly, which lists all of the courses she has taken. We'll set the index to Kelly, and the data to be the names of courses.

kelly\_classes = pd.Series(['Philosophy', 'Arts', 'Math'], index=['Kelly', 'Kelly', 'Kelly'])

kelly\_classes

* Finally, we can append all of the data in this new Series to the first using the .append() function.

all\_students\_classes = students\_classes.append(kelly\_classes)

* This creates a series which has our original people in it as well as all of Kelly's courses

all\_students\_classes

DataFrame

* DataFrame data structure is the heart of the Panda’s library.
* It’s a primary object that we will be working with in data analysis and cleaning tasks.
* DataFrame is conceptually a 2-d series object: there’s an index and multiple column contents
  + Each column has a label
* We can think of DataFrame itself as simply a two-axes labeled array.

**Examples**

* Let’s start by importing our pandas library

import pandas as pd

record1 = pd.Series({'Name': 'Alice',

'Class': 'Physics',

'Score': 85})

record2 = pd.Series({'Name': 'Jack',

'Class': 'Chemistry',

'Score': 82})

record3 = pd.Series({'Name': 'Helen',

'Class': 'Biology',

'Score': 90})

* Like a Series, the DataFrame object is index. Here I'll use a group of series, where each series represents a row of data.
* Just like the Series function, we can pass in our individual items in an array, and we can pass in our index values as a second arguments

df = pd.DataFrame([record1, record2, record3],

index=['school1', 'school2', 'school1'])

* And just like the Series we can use the head() function to see the first several rows of the dataframe, including indices from both axes, and we can use this to verify the columns and the rows

df.head()

* An alternative method is that you could use a list of dictionaries, where each dictionary represents a row of data.

students = [{'Name': 'Alice',

'Class': 'Physics',

'Score': 85},

{'Name': 'Jack',

'Class': 'Chemistry',

'Score': 82},

{'Name': 'Helen',

'Class': 'Biology',

'Score': 90}]

* Then we pass this list of dictionaries into the DataFrame function

df = pd.DataFrame(students, index=['school1', 'school2', 'school1'])

* And let’s print the head again

df.head()

* Similar to the series, we can extract data using the .iloc and .loc attributes.
* Because the DataFrame is two-dimensional, passing a single value to the loc indexing operator will return the series if there's only one row to return.
* For instance, if we wanted to select data associated with school2, we would just query the .loc attribute with one parameter.

df.loc['school2']

* It's important to remember that the indices and column names along either axes horizontal or vertical, could be non-unique. In this example, we see two records for school1 as different rows.
* If we use a single value with the DataFrame lock attribute, multiple rows of the DataFrame will return, not as a new series, but as a new DataFrame.
* Let’s query for school1 records

df.loc['school1']

* One of the powers of the Panda's DataFrame is that you can quickly select data based on multiple axes.
* For instance, if you wanted to just list the student names for school1, you would supply two parameters to .loc, one being the row index and the other being the column name.
* For instance, if we are only interested in school1's student names

df.loc['school1', 'Name']

* Remember, just like the Series, the pandas developers have implemented this using the indexing operator and not as parameters to a function.
* What would we do if we just wanted to select a single column though? Well, there are a few mechanisms.
* Firstly, we could transpose the matrix. This pivots all of the rows into columns and all of the columns into rows, and is done with the T attribute

df.T

* Then we can call .loc on the transpose to get the student names only

df.T.loc['Name']

* However, since iloc and loc are used for row selection, Panda reserves the indexing operator directly on the DataFrame for column selection. In a Panda's DataFrame, columns always have a name.
* So this selection is always label based, and is not as confusing as it was when using the square bracket operator on the series objects.
* For those familiar with relational databases, this operator is analogous to column projection.

df['Name']

* In practice, this works really well since you're often trying to add or drop new columns.
* However, this also means that you get a key error if you try and use .loc with a column name

df.loc['Name']

* Note too that the result of a single column projection is a Series object

type(df['Name'])

* Since the result of using the indexing operator is either a DataFrame or Series, you can chain operations together.
* For instance, we can select all of the rows which related to school1 using **.loc**, then project the name column from just those rows

df.loc['school1']['Name']

* If you get confused, use type to check the responses from resulting operations

print(type(df.loc['school1'])) #should be a DataFrame

print(type(df.loc['school1']['Name'])) #should be a Series

* Chaining, by indexing on the return type of another index, can come with some costs and is best avoided if you can use another approach.
* In particular, chaining tends to cause Pandas to return a copy of the DataFrame instead of a view on the DataFrame.
* For selecting data, this is not a big deal, though it might be slower than necessary.
* If you are changing data though this is an important distinction and can be a source of error.
* Here's another approach. As we saw, .loc does row selection, and it can take two parameters, the row index and the list of column names.
* The .loc attribute also supports slicing.
* If we wanted to select all rows, we can use a colon to indicate a full slice from beginning to end.
* This is just like slicing characters in a list in python. Then we can add the column name as the second parameter as a string. If we wanted to include multiple columns, we could do so in a list.
* And Pandas will bring back only the columns we have asked for.
* Here's an example, where we ask for all the names and scores for all schools using the .loc operator.

df.loc[:,['Name', 'Score']]

* Take a look at that again. The colon means that we want to get all of the rows, and the list in the second argument position is the list of columns we want to get back.
* That's selecting and projecting data from a DataFrame based on row and column labels. The key concepts to remember are that the rows and columns are really just for our benefit.
* Underneath this is just a two axes labeled array, and transposing the columns is easy.
* Also, consider the issue of chaining carefully, and try to avoid it, as it can cause unpredictable results, where your intent was to obtain a view of the data, but instead Pandas returns to you a copy.
* Before we leave the discussion of accessing data in DataFrames, let’s talk about dropping data.
* It's easy to delete data in Series and DataFrames, and we can use the drop function to do so.
* This function takes a single parameter, which is the index or row label, to drop. This is another tricky place for new users -- the drop function doesn't change the DataFrame by default!
* Instead, the drop function returns to you a copy of the DataFrame with the given rows removed.

df.drop('school1')

* But if we look at our original DataFrame we see the data is still intact.

df

* Drop has two interesting optional parameters:
  + inplace: and if it's set to true, the DataFrame will be updated in place, instead of a copy being returned.
  + axes: which should be dropped. By default, this value is 0, indicating the row axis. But you could change it to 1 if you want to drop a column.
* For example, let’s make a copy of a DataFrame using .copy()

copy\_df = df.copy()

* Now let’s drop the name column in this copy

copy\_df.drop("Name", inplace=True, axis=1)

copy\_df

* There is a second way to drop a column, and that's directly through the use of the indexing operator, using the del keyword.
* This way of dropping data, however, takes immediate effect on the DataFrame and does not return a view.

del copy\_df['Class']

copy\_df

* There is a second way to drop a column, and that's directly through the use of the indexing operator, using the del keyword.
* This way of dropping data, however, takes immediate effect on the DataFrame and does not return a view.

del copy\_df['Class']

copy\_df

* Read csv file

df = pd.read\_csv('datasets/Admission\_Predict.csv', index\_col=0)

* Rename the columns label

new\_df=df.rename(columns={'GRE Score':'GRE Score', 'TOEFL Score':'TOEFL Score',

'University Rating':'University Rating',

'SOP': 'Statement of Purpose','LOR': 'Letter of Recommendation',

'CGPA':'CGPA', 'Research':'Research',

'Chance of Admit':'Chance of Admit'})

* Get a list of columns attribute of dataframe

new\_df.columns

* We can rename one column only

new\_df=new\_df.rename(columns={'LOR ': 'Letter of Recommendation'})

new\_df.head()

* “strip()” is to strup white space. When we pass this in to rename we pass the function as mapper parameter, and then indicate whether axis should be columns or index (row labels)

new\_df=new\_df.rename(mapper=str.strip, axis='columns')

* We can also use the df.columns attribute by assigning to it a list of column names which will directly rename the columns.
* This will directly modify the original dataframe and is very efficient especially when you have a lot of columns and you only want to change a few.
* This technique is also not affected by subtle errors in the column names, a problem that we just encountered. With a list, you can use the list index to change a certain value or use list comprehension to change all of the values
* As an example, lets change all of the column names to lower case. First we need to get our list

cols = list(df.columns)

* Then a little list comprehenshion

cols = [x.lower().strip() for x in cols]

* Then we just overwrite what is already in the .columns attribute

df.columns=cols

* And take a look at our results

df.head()

* To “hide” data you don’t want, which is represented by all of the False values. We do this by using .where() function on the original DataFrame.

df.where(admit\_mask).head()

* If we don't want the NaN data, we use the dropna() function

df.where(admit\_mask).dropna().head()

* Despite being really handy, where() isn't actually used that often. Instead, the pandas devs created a shorthand syntax which combines where() and dropna(), doing both at once. And, in typical fashion, the just overloaded the indexing operator to do this!

df[df['chance of admit'] > 0.7].head()

* A string parameter to project a single column

df["gre score"].head()

* Or you can send it a list of columns as strings

df[["gre score","toefl score"]].head()

* Or you can send it a boolean mask

df[df["gre score"]>320].head()

* Comparison
  + working version

(df['chance of admit'] > 0.7) & (df['chance of admit'] < 0.9)

df['chance of admit'].gt(0.7) & df['chance of admit'].lt(0.9)

df['chance of admit'].gt(0.7).lt(0.9)

* + not working versions

(df['chance of admit'] > 0.7) and (df['chance of admit'] < 0.9)

df['chance of admit'] > 0.7 & df['chance of admit'] < 0.9

* So we copy the indexed data into its own column

df['Serial Number'] = df.index

* Then we set the index to another column

df = df.set\_index('Chance of Admit ')

df.head()

* We can get rid of the index completely by calling the function reset\_index(). This promotes the index into a column and creates a default numbered index.

df = df.reset\_index()

df.head()

* Here we can run unique on the sum level of our current DataFrame

df['SUMLEV'].unique()

* Let's exclue all of the rows that are summaries at the state level and just keep the county data.

df=df[df['SUMLEV'] == 50]

df.head()

* Also while this data set is interesting for a number of different reasons, let's reduce the data that we're going to look at to just the total population estimates and the total number of births.
* We can do this by creating a list of column names that we want to keep then project those and assign the resulting DataFrame to our df variable.

columns\_to\_keep = ['STNAME','CTYNAME','BIRTHS2010','BIRTHS2011','BIRTHS2012','BIRTHS2013',

'BIRTHS2014','BIRTHS2015','POPESTIMATE2010','POPESTIMATE2011',

'POPESTIMATE2012','POPESTIMATE2013','POPESTIMATE2014','POPESTIMATE2015']

df = df[columns\_to\_keep]

df.head()

* The US Census data breaks down population estimates by state and county. We can load the data and set the index to be a combination of the state and county values and see how pandas handles it in a DataFrame. We do this by creating a list of the column identifiers we want to have indexed. And then calling set index with this list and assigning the output as appropriate. We see here that we have a dual index, first the state name and second the county name.

df = df.set\_index(['STNAME', 'CTYNAME'])

df.head()

df.loc['Michigan', 'Washtenaw County']

df.loc[ [('Michigan', 'Washtenaw County'),

('Michigan', 'Wayne County')] ]

* We can actually use the function .isnull() to create a boolean mask of the whole dataframe. This effectively broadcasts the isnull() function to every cell of data.

mask=df.isnull()

mask.head(10)

* This can be useful for processing rows based on certain columns of data. Another useful operation is to be able to drop all of those rows which have any missing data, which can be done with the dropna() function.

df.dropna().head(10)

* Note how the rows indexed with 2, 3, 7, and 11 are now gone. One of the handy functions that Pandas has for working with missing values is the filling function, fillna(). This function takes a number or parameters.
* You could pass in a single value which is called a scalar value to change all of the missing data to one value. This isn't really applicable in this case, but it's a pretty common use case.
* So, if we wanted to fill all missing values with 0, we would use fillna

df.fillna(0, inplace=True)

df.head(10)

* In Pandas we can sort either by index or by values. Here we'll just promote the time stamp to an index then sort on the index.

df = df.set\_index('time')

df = df.sort\_index()

df.head(20)

* If we look closely at the output though we'll notice that the index isn't really unique. Two users seem to be able to use the system at the same time.
* Again, a very common case. Let's reset the index, and use some multi-level indexing on time AND user together instead, promote the user name to a second level of the index to deal with that issue.

df = df.reset\_index()

df = df.set\_index(['time', 'user'])

df

* Two common fill values are ffill and bfill.
  + ffill is for forward filling and it updates an na value for a particular cell with the value from the previous row.
  + bfill is backward filling, which is the opposite of ffill. It fills the missing values with the next valid value.
* It's important to note that your data needs to be sorted in order for this to have the effect you might want.
* Data which comes from traditional database management systems usually has no order guarantee, just like this data. So be careful.

df = df.fillna(method='ffill')

df.head()

* We can also do customized fill-in to replace values with the replace() function. It allows replacement from several approaches: value-to-value, list, dictionary, regex Let's generate a simple example

df = pd.DataFrame({'A': [1, 1, 2, 3, 4],

'B': [3, 6, 3, 8, 9],

'C': ['a', 'b', 'c', 'd', 'e']})

df

* We can replace 1's with 100, let's try the value-to-value approach

df.replace(1, 100) # change 2 numbers

* How about changing two values? Let's try the list approach For example, we want to change 1's to 100 and 3's to 300

df.replace([1, 3], [100, 300])

* To replace using a regex we make the first parameter to replace the regex pattern we want to match, the second parameter the value we want to emit upon match, and then we pass in a third parameter "regex=True".
* Take a moment to pause this video and think about this problem: imagine we want to detect all html pages in the "video" column, let’s say that just means they end with ".html", and we want to overwrite that with the keyword "webpage". How could we accomplish this?
* Here's my solution, first matching any number of characters then ending in .html

df.replace(to\_replace=".\*.html$", value="webpage", regex=True)

**Example of Manipulating DataFrames**

* Let's start by bringing in pandas

import pandas as pd

* And load our dataset. We're going to be cleaning the list of presidents in the US from wikipedia

df=pd.read\_csv("datasets/presidents.csv")

* And let’s just take a look at some of the data

df.head()

* Ok, we have some presidents, some dates, I see a bunch of footnotes in the "Born" column which might cause issues. Let's start with cleaning up that name into firstname and lastname.
* I'm going to tackle this with a regex. So I want to create two new columns and apply a regex to the projection of the "President" column.
* Here's one solution, we could make a copy of the President column

df["First"]=df['President']

* Then we can call replace() and just have a pattern that matches the last name and set it to an empty string

df["First"]=df["First"].replace("[ ].\*", "", regex=True)

* Now let's take a look

df.head()

* That works, but it's kind of gross. And it's slow, since we had to make a full copy of a column then go through and update strings. There are a few other ways we can deal with this. Let me show you the most general one first, and that's called the apply() function. Let's drop the column we made first

del(df["First"])

* The apply() function on a dataframe will take some arbitrary function you have written and apply it to either a Series (a single column) or DataFrame across all rows or columns. Let’s write a function which just splits a string into two pieces using a single row of data

def splitname(row):

* The row is a single Series object which is a single row indexed by column values
* Let's extract the firstname and create a new entry in the series

row['First']=row['President'].split(" ")[0]

* Let's do the same with the last word in the string

row['Last']=row['President'].split(" ")[-1]

* Now we just return the row and the pandas .apply() will take of merging them back into a DataFrame

return row

* Now if we apply this to the dataframe indicating we want to apply it across columns

df=df.apply(splitname, axis='columns')

df.head()

* Extract takes a regular expression as input and specifically requires you to set capture groups that correspond to the output columns you are interested in.
* Write a regular expression that returned groups and just had the firstname and lastname in it: where we match three groups but only return two, the first and the last name

pattern="(^[\w]\*)(?:.\* )([\w]\*$)"

* Now the extract function is built into the str attribute of the Series object, so we can call it using Series.str.extract(pattern)

df["President"].str.extract(pattern).head()

* So that looks pretty nice, other than the column names. But if we name the groups we get named columns out

pattern="(?P<First>^[\w]\*)(?:.\* )(?P<Last>[\w]\*$)"

* Now call extract

names=df["President"].str.extract(pattern).head()

names

* And we can just copy these into our main dataframe if we want to

df["First"]=names["First"]

df["Last"]=names["Last"]

df.head()

* Now let’s move on to clean up that Born column. First, let's get rid of anything that isn't in the pattern of Month Day and Year.

df["Born"]=df["Born"].str.extract("([\w]{3} [\w]{1,2}, [\w]{4})")

df["Born"].head()

* So, that cleans up the date format. But I'm going to foreshadow something else here - the type of this column is object, and we know that's what pandas uses when it is dealing with string.
* But pandas actually has really interesting date/time features - in fact, that's one of the reasons Wes McKinney put his efforts into the library, to deal with financial transactions.
* So if I were building this out, I would actually update this column to the write data type as well

df["Born"]=pd.to\_datetime(df["Born"])

df["Born"].head()

**Week 3: More Data Processing with Pandas**

**Terminology**

* Full outer join (database terminology) / union (set theory terminology): everyone in any circle.
* Inner join (database terminology)/ intersection (set theory terminology): represents the overlapping parts of each circle.

**Merging in Pandas**

* With that background, let's see an example of how we would do this in pandas, where we would use the merge function.

import pandas as pd

* First we create two DataFrames, staff and students.

staff\_df = pd.DataFrame([{'Name': 'Kelly', 'Role': 'Director of HR'},

{'Name': 'Sally', 'Role': 'Course liasion'},

{'Name': 'James', 'Role': 'Grader'}])

* And let’s index these staff by name

staff\_df = staff\_df.set\_index('Name')

* Now we'll create a student dataframe

student\_df = pd.DataFrame([{'Name': 'James', 'School': 'Business'},

{'Name': 'Mike', 'School': 'Law'},

{'Name': 'Sally', 'School': 'Engineering'}])

* And we'll index this by name too

student\_df = student\_df.set\_index('Name')

* And let’s just print out the dataframes

print(staff\_df.head())

print(student\_df.head())

* There's some overlap in these DataFrames in that James and Sally are both students and staff, but Mike and Kelly are not.
* Importantly, both DataFrames are indexed along the value we want to merge them on, which is called Name.
* If we want the union of these, we would call merge() passing in the DataFrame on the left and the DataFrame on the right and telling merge that we want it to use an outer join.
* We want to use the left and right indices as the joining columns.

pd.merge(staff\_df, student\_df, how='outer', left\_index=True, right\_index=True)

* We see in the resulting DataFrame that everyone is listed. And since Mike does not have a role, and John does not have a school, those cells are listed as missing values.
* If we wanted to get the intersection, that is, just those who are a student AND a staff, we could set the how attribute to inner. Again, we set both left and right indices to be true as the joining columns

pd.merge(staff\_df, student\_df, how='inner', left\_index=True, right\_index=True)

* Now there are two other common use cases
* When merging DataFrames, and both are examples of what we would call set addition. The first is when we would want to get a list of all staff regardless of whether they were students or not.
* But if they were students, we would want to get their student details as well. To do this we would use a left join.
* It is important to note the order of dataframes in this function: the **first dataframe is the left dataframe** and **the second is the right**

pd.merge(staff\_df, student\_df, how='left', left\_index=True, right\_index=True)

* We can also do it another way. The merge method has a couple of other interesting parameters.
* First, you don't need to use indices to join on, you can use columns as well. Here's an example.
* Here we have a parameter called "on", and we can assign a column that both dataframe has as the joining column
* First, let’s remove our index from both of our dataframes

staff\_df = staff\_df.reset\_index()

student\_df = student\_df.reset\_index()

* Now let’s merge using the on parameter

pd.merge(staff\_df, student\_df, how='right', on='Name')

* Using the "on" parameter instead of a the index is how I find myself using merge() the most.
* So what happens when we have conflicts between the DataFrames? Let's take a look by creating new staff and student DataFrames that have a location information added to them.

staff\_df = pd.DataFrame([{'Name': 'Kelly', 'Role': 'Director of HR',

'Location': 'State Street'},

{'Name': 'Sally', 'Role': 'Course liasion',

'Location': 'Washington Avenue'},

{'Name': 'James', 'Role': 'Grader',

'Location': 'Washington Avenue'}])

student\_df = pd.DataFrame([{'Name': 'James', 'School': 'Business',

'Location': '1024 Billiard Avenue'},

{'Name': 'Mike', 'School': 'Law',

'Location': 'Fraternity House #22'},

{'Name': 'Sally', 'School': 'Engineering',

'Location': '512 Wilson Crescent'}])

* In the staff DataFrame, this is an office location where we can find the staff person. And we can see the Director of HR is on State Street, while the two students are on Washington Avenue, and these locations just happen to be right outside my window as I film this. But for the student DataFrame, the location information is actually their home address.
* The merge function preserves this information, but appends an \_x or \_y to help differentiate between which index went with which column of data. The \_x is always the left DataFrame information, and the \_y is always the right DataFrame information.
* Here, if we want all the staff information regardless of whether they were students or not. But if they were students, we would want to get their student details as well.
* Then we can do a left join and on the column of Name

pd.merge(staff\_df, student\_df, how='left', on='Name')

# From the output, we can see there are columns Location\_x and Location\_y. Location\_x refers to the Location

# column in the left dataframe, which is staff dataframe and Location\_y refers to the Location column in the

# right dataframe, which is student dataframe.

# Before we leave merging of DataFrames, let's talk about multi-indexing and multiple columns. It's quite

# possible that the first name for students and staff might overlap, but the last name might not. In this

# case, we use a list of the multiple columns that should be used to join keys from both dataframes on the on

# parameter. Recall that the column name(s) assigned to the on parameter needs to exist in both dataframes.

# Here's an example with some new student and staff data

staff\_df = pd.DataFrame([{'First Name': 'Kelly', 'Last Name': 'Desjardins',

'Role': 'Director of HR'},

{'First Name': 'Sally', 'Last Name': 'Brooks',

'Role': 'Course liasion'},

{'First Name': 'James', 'Last Name': 'Wilde',

'Role': 'Grader'}])

student\_df = pd.DataFrame([{'First Name': 'James', 'Last Name': 'Hammond',

'School': 'Business'},

{'First Name': 'Mike', 'Last Name': 'Smith',

'School': 'Law'},

{'First Name': 'Sally', 'Last Name': 'Brooks',

'School': 'Engineering'}])

* As you see here, James Wilde and James Hammond don't match on both keys since they have different last names. So we would expect that an inner join doesn't include these individuals in the output, and only Sally Brooks will be retained.

pd.merge(staff\_df, student\_df, how='inner', on=['First Name','Last Name'])

* If we think of merging as joining "horizontally", meaning we join on similar values in a column found in two dataframes then concatenating is joining "vertically", meaning we put dataframes on top or at the bottom of each other
* Let's understand this from an example. You have a dataset that tracks some information over the years. And each year's record is a separate CSV and every CSV or every year's record has the exactly same columns.
* What happens if you want to put all the data, from all years' record, together? You can concatenate them.

**Example of concatenate**

* Let's take a look at the US Department of Education College Scorecard data It has each US university's data on student completion, student debt, after-graduation income, etc.
* The data is stored in separate CSV's with each CSV containing a year's record
* Let's say we want the records from 2011 to 2013 we first create three dataframe, each containing one year's record. And, because the csv files we're working with are messy, I want to supress some of the jupyter warning messages and just tell read\_csv to ignore bad lines, so I'm going to start the cell with a cell magic called %%capture

%%capture

df\_2011 = pd.read\_csv("datasets/college\_scorecard/MERGED2011\_12\_PP.csv", error\_bad\_lines=False)

df\_2012 = pd.read\_csv("datasets/college\_scorecard/MERGED2012\_13\_PP.csv", error\_bad\_lines=False)

df\_2013 = pd.read\_csv("datasets/college\_scorecard/MERGED2013\_14\_PP.csv", error\_bad\_lines=False)

* Let's get a view of one of the dataframes

df\_2011.head(3)

* We see that there is a whopping number of columns - more than 1900! We can calculate the length of each dataframe as well

print(len(df\_2011))

print(len(df\_2012))

print(len(df\_2013))

* That's a bit surprising that the number of schools in the scorecard for 2011 is almost double that of the next two years. But let's not worry about that. Instead, let's just put all three dataframes in a list and call that list frames and pass the list into the concat() function Let's see what it looks like

frames = [df\_2011, df\_2012, df\_2013]

pd.concat(frames)

* As you can see, we have more observations in one dataframe and columns remain the same. If we scroll down to the bottom of the output, we see that there are a total of 30,832 rows after concatenating three dataframes.
* Let's add the number of rows of the three dataframes and see if the two numbers match

len(df\_2011)+len(df\_2012)+len(df\_2013)

* The two numbers match! Which means our concatenation is successful. But wait, now that all the data is concatenated together, we don't know what observations are from what year anymore!
* Actually the concat function has a parameter that solves such problem with the keys parameter, we can set an extra level of indices, we pass in a list of keys that we want to correspond to the dataframes into the keys parameter
* Now let's try it out

pd.concat(frames, keys=['2011','2012','2013'])

* Now we have the indices as the year so we know what observations are from what year. You should know that concatenation also has inner and outer method. If you are concatenating two dataframes that do not have identical columns, and choose the outer method, some cells will be NaN. If you choose to do inner, then some observations will be dropped due to NaN values. You can think of this as analogous to the left and right joins of the merge() function.

**Pandas idioms**

**The first Pandas idiom: method cleaning**

* The first of the pandas idioms I would like to talk about is called method chaining.
  + The general idea behind method chaining is that every method on an object returns a reference to that object.
  + The beauty of this is that you can condense many different operations on a DataFrame, for instance, into one line or at least one statement of code.
* Here's the pandorable way to write code with method chaining. In this code I'm going to pull out the state and city names as a multiple index, and I'm going to do so only for data which has a summary level of 50, which in this dataset is county-level data. I'll rename a column too, just to make it a bit more readable.

(df.where(df['SUMLEV']==50)

.dropna()

.set\_index(['STNAME','CTYNAME'])

.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'}))

* Let’s walk through this.
  + First, we use the where() function on the dataframe and pass in a boolean mask which is only true for those rows where the SUMLEV is equal to 50. This indicates in our source data that the data is summarized at the county level. With the result of the where() function evaluated, we drop missing values. Remember that .where() doesn't drop missing values by default.
  + Then we set an index on the result of that. In this case I've set it to the state name followed by the county name.
  + Finally. I rename a column to make it more readable. Note that instead of writing this all on one line, as I could have done, I began the statement with a parenthesis, which tells python I'm going to span the statement over multiple lines for readability.
* Here's a more traditional, non-pandorable way, of writing this. There's nothing wrong with this code in the functional sense, you might even be able to understand it better as a new person to the language. It's just not as pandorable as the first example.
* First create a new dataframe from the original

df = df[df['SUMLEV']==50] # I'll use the overloaded indexing operator [] which drops nans

* Update the dataframe to have a new index, we use inplace=True to do this in place

df.set\_index(['STNAME','CTYNAME'], inplace=True)

* Set the column names

df.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'})

* Now, the key with any good idiom is to understand when it isn't helping you. In this case, you can actually time both methods and see which one runs faster
* We can put the approach into a function and pass the function into the timeit function to count the time the parameter number allows us to choose how many times we want to run the function.
* Here we will just set it to 10
* Let’s write a wrapper for our first function

def first\_approach():

global df

* And we'll just paste our code right here

return (df.where(df['SUMLEV']==50)

.dropna()

.set\_index(['STNAME','CTYNAME'])

.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'}))

* Read in our dataset anew

df = pd.read\_csv('datasets/census.csv')

* And now let’s run it

timeit.timeit(first\_approach, number=10)

* Now let's test the second approach. As you may notice, we use our global variable df in the function.
* However, changing a global variable inside a function will modify the variable even in a global scope and we do not want that to happen in this case. Therefore, for selecting summary levels of 50 only, I create a new dataframe for those records
* Let's run this for once and see how fast it is

def second\_approach():

global df

new\_df = df[df['SUMLEV']==50]

new\_df.set\_index(['STNAME','CTYNAME'], inplace=True)

return new\_df.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'})

* Read in our dataset anew

df = pd.read\_csv('datasets/census.csv')

* And now let’s run it

timeit.timeit(second\_approach, number=10)

* As you can see, the second approach is much faster! So, this is a particular example of a classic time readability trade off.
* You'll see lots of examples on stack overflow and in documentation of people using method chaining in their pandas. And so, I think being able to read and understand the syntax is really worth your time.
* But keep in mind that following what appears to be stylistic idioms might have performance issues that you need to consider as well.

**The second Pandas idiom: map**

* Here's another pandas idiom. Python has a wonderful function called map, which is sort of a basis for functional programming in the language. When you want to use map in Python, you pass it some function you want called, and some iterable, like a list, that you want the function to be applied to. The results are that the function is called against each item in the list, and there's a resulting list of all of the evaluations of that function.
* Pandas has a similar function called applymap. In applymap, you provide some function which should operate on each cell of a DataFrame, and the return set is itself a DataFrame. Now I think applymap is fine, but I actually rarely use it. Instead, I find myself often wanting to map across all of the rows in a DataFrame.
* And pandas has a function that I use heavily there, called apply. Let's look at an example.
* Let's take a look at our census DataFrame. In this DataFrame, we have five columns for population estimates, with each column corresponding with one year of estimates. It's quite reasonable to want to create some new columns for minimum or maximum values, and the apply function is an easy way to do this.
* First, we need to write a function which takes in a particular row of data, finds a minimum and maximum values, and returns a new row of data nd returns a new row of data. We'll call this function min\_max, this is pretty straight forward. We can create some small slice of a row by projecting the population columns.
* Then use the NumPy min and max functions, and create a new series with a label values represent the new values we want to apply.

def min\_max(row):

data = row[['POPESTIMATE2010',

'POPESTIMATE2011',

'POPESTIMATE2012',

'POPESTIMATE2013',

'POPESTIMATE2014',

'POPESTIMATE2015']]

return pd.Series({'min': np.min(data), 'max': np.max(data)})

* Then we just need to call apply on the DataFrame.
* Apply takes the function and the axis on which to operate as parameters. Now, we have to be a bit careful, we've talked about axis zero being the rows of the DataFrame in the past.
* But this parameter is really the parameter of the index to use. So, to apply across all rows, which is applying on all columns, you pass axis equal to 'columns'.

df.apply(min\_max, axis='columns').head()

* Here's an example where we have a revised version of the function min\_max Instead of returning a separate series to display the min and max we add two new columns in the original dataframe to store min and max

def min\_max(row):

data = row[['POPESTIMATE2010',

'POPESTIMATE2011',

'POPESTIMATE2012',

'POPESTIMATE2013',

'POPESTIMATE2014',

'POPESTIMATE2015']]

* Create a new entry for max

row['max'] = np.max(data)

* Create a new entry for min

row['min'] = np.min(data)

return row

* Now just apply the function across the dataframe

df.apply(min\_max, axis='columns')

* Apply is an extremely important tool in your toolkit. The reason I introduced apply here is because you rarely see it used with large function definitions, like we did. Instead, you typically see it used with lambdas. To get the most of the discussions you'll see online, you're going to need to know how to at least read lambdas.
* Here, you can imagine how you might chain several apply calls with lambdas together to create a readable yet succinct data manipulation script. One line example of how you might calculate the max of the columns using the apply function.

rows = ['POPESTIMATE2010', 'POPESTIMATE2011', 'POPESTIMATE2012', 'POPESTIMATE2013','POPESTIMATE2014',

'POPESTIMATE2015']

* Now we'll just apply this across the dataframe with a lambda

df.apply(lambda x: np.max(x[rows]), axis=1).head()

* The beauty of the apply function is that it allows flexibility in doing whatever manipulation that you desire, as the function you pass into apply can be any customized however you want.
* Let's say we want to divide the states into four categories: Northeast, Midwest, South, and West
* We can write a customized function that returns the region based on the state the state regions information is obtained from Wikipedia

def get\_state\_region(x):

northeast = ['Connecticut', 'Maine', 'Massachusetts', 'New Hampshire',

'Rhode Island','Vermont','New York','New Jersey','Pennsylvania']

midwest = ['Illinois','Indiana','Michigan','Ohio','Wisconsin','Iowa',

'Kansas','Minnesota','Missouri','Nebraska','North Dakota',

'South Dakota']

south = ['Delaware','Florida','Georgia','Maryland','North Carolina',

'South Carolina','Virginia','District of Columbia','West Virginia',

'Alabama','Kentucky','Mississippi','Tennessee','Arkansas',

'Louisiana','Oklahoma','Texas']

west = ['Arizona','Colorado','Idaho','Montana','Nevada','New Mexico','Utah',

'Wyoming','Alaska','California','Hawaii','Oregon','Washington']

if x in northeast:

return "Northeast"

elif x in midwest:

return "Midwest"

elif x in south:

return "South"

else:

return "West"

* Now we have the customized function, let's say we want to create a new column called Region, which shows the state's region, we can use the customized function and the apply function to do so.
* The customized function is supposed to work on the state name column STNAME. So we will set the apply function on the state name column and pass the customized function into the apply function

df['state\_region'] = df['STNAME'].apply(lambda x: get\_state\_region(x))

* Now let's see the results

df[['STNAME','state\_region']].head()

**Grouping data**

* We have seen that even though Pandas allows us to iterate over every row in a data , it is generally very slow to do so.
* Fortunately Pandas has a groupby() function to speed up such task. The idea behind the groupby() function is that
  + it takes some dataframe, splits it into chunks based on some key values,
  + applies computation on those chunks,
  + then combines the results back together into another dataframe. In pandas this is referred to as the split-apply-combine pattern.

**Splitting**

* Let's look at an example. First, we'll bring in our pandas and numpy libraries

import pandas as pd

import numpy as np

* Let's look at some US census data

df = pd.read\_csv('datasets/census.csv')

* And exclude state level summarizations, which have sum level value of 40

df = df[df['SUMLEV']==50]

df.head()

* In the first example for groupby() I want to use the census date. Let's get a list of the unique states, then we can iterate over all the states and for each state we reduce the data frame and calculate the average.
* Let's run such task for 3 times and time it. For this we'll use the cell magic function %%timeit

%%timeit -n 3

for state in df['STNAME'].unique():

# We'll just calculate the average using numpy for this particular state

avg = np.average(df.where(df['STNAME']==state).dropna()['CENSUS2010POP'])

# And we'll print it to the screen

print('Counties in state ' + state +

' have an average population of ' + str(avg))

* If you scroll down to the bottom of that output you can see it takes a fair bit of time to finish.
* Now let's try another approach using groupby()

%%timeit -n 3

# For this method, we start by telling pandas we're interested in grouping by state name, this is the "split"

for group, frame in df.groupby('STNAME'):

# You'll notice there are two values we set here. groupby() returns a tuple, where the first value is the

# value of the key we were trying to group by, in this case a specific state name, and the second one is

# projected dataframe that was found for that group

# Now we include our logic in the "apply" step, which is to calculate an average of the census2010pop

avg = np.average(frame['CENSUS2010POP'])

# And print the results

print('Counties in state ' + group +

' have an average population of ' + str(avg))

* And we don't have to worry about the combine step in this case, because all of our data transformation is actually printing out results.
* Wow, what a huge difference in speed. An improve by roughly by two factors!
* Now, 99% of the time, you'll use group by on one or more columns. But you can also provide a function to group by and use that to segment your data.
* This is a bit of a fabricated example but let’s say that you have a big batch job with lots of processing and you want to work on only a third or so of the states at a given time.
* We could create some function which returns a number between zero and two based on the first character of the state name.
* Then we can tell group by to use this function to split up our data frame. It's important to note that in order to do this you need to set the index of the data frame to be the column that you want to group by first.
* We'll create some new function called set\_batch\_number and if the first letter of the parameter is a capital M we'll return a 0. If it's a capital Q we'll return a 1 and otherwise we'll return a 2. Then we'll pass this function to the data frame

df = df.set\_index('STNAME')

def set\_batch\_number(item):

if item[0]<'M':

return 0

if item[0]<'Q':

return 1

return 2

# The dataframe is supposed to be grouped by according to the batch number And we will loop through each batch

# group

for group, frame in df.groupby(set\_batch\_number):

print('There are ' + str(len(frame)) + ' records in group ' + str(group) + ' for processing.')

* Notice that this time I didn't pass in a column name to groupby(). Instead, I set the index of the dataframe to be STNAME, and if no column identifier is passed groupby() will automatically use the index.
* Let's take one more look at an example of how we might group data. In this example, I want to use a dataset of housing from airbnb.
* In this dataset there are two columns of interest, one is the cancellation\_policy and the other is the review\_scores\_value.

df=pd.read\_csv("datasets/listings.csv")

df.head()

* So, how would I group by both of these columns? A first approach might be to promote them to a multiindex and just call groupby()

df=df.set\_index(["cancellation\_policy","review\_scores\_value"])

* When we have a multiindex we need to pass in the levels we are interested in grouping by

for group, frame in df.groupby(level=(0,1)):

print(group)

* This seems to work ok. But what if we wanted to group by the cancelation policy and review scores, but separate out all the 10's from those under ten? In this case, we could use a function to manage the groupings

def grouping\_fun(item):

# Check the "review\_scores\_value" portion of the index. item is in the format of

# (cancellation\_policy,review\_scores\_value

if item[1] == 10.0:

return (item[0],"10.0")

else:

return (item[0],"not 10.0")

for group, frame in df.groupby(by=grouping\_fun):

print(group)

* This seems to work ok. But what if we wanted to group by the cancelation policy and review scores, but separate out all the 10's from those under ten? In this case, we could use a function to manage the groupings

def grouping\_fun(item):

# Check the "review\_scores\_value" portion of the index. item is in the format of

# (cancellation\_policy,review\_scores\_value

if item[1] == 10.0:

return (item[0],"10.0")

else:

return (item[0],"not 10.0")

for group, frame in df.groupby(by=grouping\_fun):

print(group)

df.head()

**Applying**

* To this point we have applied very simple processing to our data after splitting, really just outputting some print statements to demonstrate how the splitting works.
* The pandas developers have three broad categories of data processing to happen during the apply step
  + Aggregation of group data,
  + Transformation of group data, and
  + Filtration of group data

**Aggregation**

* The most straight forward apply step is the aggregation of data, and uses the method agg() on the groupby object. Thus far we have only iterated through the groupby object, unpacking it into a label (the group name) and a dataframe.
* But with agg we can pass in a dictionary of the columns we are interested in aggregating along with the function we are looking to apply to aggregate.
* Let's reset the index for our airbnb data

df=df.reset\_index()

# Now let’s group by the cancellation policy and find the average review\_scores\_value by group

df.groupby("cancellation\_policy").agg({"review\_scores\_value":np.average})

* Hrm. That didn't seem to work at all. Just a bunch of not a numbers. The issue is actually in the function that we sent to aggregate. np.average does not ignore nans! However, there is a function we can use for this

df.groupby("cancellation\_policy").agg({"review\_scores\_value":np.nanmean})

* We can just extend this dictionary to aggregate by multiple functions or multiple columns.

df.groupby("cancellation\_policy").agg({"review\_scores\_value":(np.nanmean,np.nanstd),

"reviews\_per\_month":np.nanmean})

* Take a moment to make sure you understand the previous cell, since it's somewhat complex.
  + First we're doing a group by on the dataframe object by the column "cancellation\_policy". This creates a new GroupBy object.
  + Then we are invoking the agg() function on that object. The agg function is going to apply one or more functions we specify to the group dataframes and return a single row per dataframe/group.
  + When we called this function we sent it two dictionary entries, each with the key indicating which column we wanted functions applied to.
  + For the first column we actually supplied a tuple of two functions. Note that these are not function invocations, like np.nanmean(), or function names, like "nanmean" they are references to functions which will return single values.
  + The groupby object will recognize the tuple and call each function in order on the same column.
  + The results will be in a heirarchical index, but since they are columns they don't show as an index per se. Then we indicated another column and a single function we wanted to run.

**Transformation**

* Transformation is different from aggregation. Where agg() returns a single value per column, so one row per group, tranform() returns an object that is the same size as the group. Essentially, it broadcasts the function you supply over the grouped dataframe, returning a new dataframe. This makes combining data later easy.
* For instance, suppose we want to include the average rating values in a given group by cancellation policy, but preserve the dataframe shape so that we could generate a difference between an individual observation and the sum.

# First, lets define just some subset of columns we are interested in

cols=['cancellation\_policy','review\_scores\_value']

# Now let’s transform it, I'll store this in its own dataframe

transform\_df=df[cols].groupby('cancellation\_policy').transform(np.nanmean)

transform\_df.head()

* So we can see that the index here is actually the same as the original dataframe. So let’s just join this in. Before we do that, let’s rename the column in the transformed version

transform\_df.rename({'review\_scores\_value':'mean\_review\_scores'}, axis='columns', inplace=True)

df=df.merge(transform\_df, left\_index=True, right\_index=True)

df.head()

* Great, we can see that our new column is in place, the mean\_review\_scores. So now we could create, for instance, the difference between a given row and it's group (the cancellation policy) means.

df['mean\_diff']=np.absolute(df['review\_scores\_value']-df['mean\_review\_scores'])

df['mean\_diff'].head()

**Filtering**

* The GroupBy object has built-in support for filtering groups as well. It's often that you'll want to group by some feature, then make some transformation to the groups, then drop certain groups as part of your cleaning routines.
* The filter() function takes in a function which it applies to each group dataframe and returns either a True or a False, depending upon whether that group should be included in the results.
* For instance, if we only want those groups which have a mean rating above 9 included in our results

df.groupby('cancellation\_policy').filter(lambda x: np.nanmean(x['review\_scores\_value'])>9.2)

* Notice that the results are still indexed, but that any of the results which were in a group with a mean review score of less than or equal to 9.2 were not copied over.

**Applying**

* By far the most common operation I invoke on groupby objects is the apply() function. This allows you to apply an arbitrary function to each group, and stitch the results back for each apply() into a single dataframe where the index is preserved.
* Let’s look at an example using our airbnb data, I'm going to get a clean copy of the dataframe

df=pd.read\_csv("datasets/listings.csv")

# And lets just include some of the columns we were interested in previously

df=df[['cancellation\_policy','review\_scores\_value']]

df.head()

* In previous work we wanted to find the average review score of a listing and its deviation from the group mean.
* This was a two step process,
  + first we used transform() on the groupby object and
  + then we had to broadcast to create a new column.
* With apply() we could wrap this logic in one place

def calc\_mean\_review\_scores(group):

# group is a dataframe just of whatever we have grouped by, e.g. cancellation policy, so we can treat

# this as the complete dataframe

avg=np.nanmean(group["review\_scores\_value"])

# now broadcast our formula and create a new column

group["review\_scores\_mean"]=np.abs(avg-group["review\_scores\_value"])

return group

# Now just apply this to the groups

df.groupby('cancellation\_policy').apply(calc\_mean\_review\_scores).head()

* Using apply can be slower than using some of the specialized functions, especially agg(). But, if your dataframes are not huge, it's a solid general purpose approach.

**Scales**

* There are at least four scales that’s worth knowing about
  + Ratio scale:
    - units are equally spaced
    - mathematical operations +-/\* are all valid
    - e.g. height and weight
  + Interval scale:
    - units are equally spaced, but there is no true zero
    - so operations such as \*/ are not valid
    - e.g. temperature measured in Celsius or Fahrenheit since there’s never an absence of temperature, and zero degrees is actually a meaningful value itself.
    - e.g2. direction of compass, where zero degrees on the compass doesn’t indicate a lack of direction, but instead describes a direction itself.
  + Ordinal scale
    - the order of the units is important, but not evenly spaced.
    - common in ML and sometimes can be a bit of a challenge to work with
    - e.g. letter grades such as A+, A are good examples.
  + Nominal scale
    - categories of data, but the categories have no order with respect to one another.
    - categorical values refer to categories where there are only two possible values as binary categories.
    - e.g. teams of a sport. There are a limited number of teams, but changing their order or applying mathematical functions to them is meaningless.

**Scales in python**

* Let's bring in pandas as normal

import pandas as pd

* Here’s an example. Lets create a dataframe of letter grades in descending order. We can also set an index value and here we'll just make it some human judgement of how good a student was, like "excellent" or "good"

df=pd.DataFrame(['A+', 'A', 'A-', 'B+', 'B', 'B-', 'C+', 'C', 'C-', 'D+', 'D'],

index=['excellent', 'excellent', 'excellent', 'good', 'good', 'good',

'ok', 'ok', 'ok', 'poor', 'poor'],

columns=["Grades"])

df

* Now, if we check the datatype of this column, we see that it's just an object, since we set string values

df.dtypes

* We can, however, tell pandas that we want to change the type to category, using the astype() function

df["Grades"].astype("category").head()

* We see now that there are eleven categories, and pandas is aware of what those categories are.
* More interesting though is that our data isn't just categorical, but that it's ordered. That is, an A- comes after a B+, and B comes before a B+.
* We can tell pandas that the data is ordered by first creating a new categorical data type with the list of the categories (in order) and the ordered=True flag

my\_categories=pd.CategoricalDtype(categories=['D', 'D+', 'C-', 'C', 'C+', 'B-', 'B', 'B+', 'A-', 'A', 'A+'],

ordered=True)

# then we can just pass this to the astype() function

grades=df["Grades"].astype(my\_categories)

grades.head()

* Now we see that pandas is not only aware that there are 11 categories, but it is also aware of the order of those categoreies. So, what can you do with this?
* Well because there is an ordering this can help with comparisons and boolean masking. For instance, if we have a list of our grades and we compare them to a “C” we see that the lexicographical comparison returns results we were not intending.

df[df["Grades"]>"C"]

* We see that the operator works as we would expect. We can then use a certain set of mathematical operators, like minimum, maximum, etc., on the ordinal data.
* Sometimes it is useful to represent categorical values as each being a column with a true or a false as to whether the category applies.
* This is especially common in feature extraction, which is a topic in the data mining course.
* Variables with a boolean value are typically called dummy variables, and pandas has a built in function called get\_dummies which will convert the values of a single column into multiple columns of zeros and ones indicating the presence of the dummy variable. I rarely use it, but when I do it's very handy.
* There’s one more common scale-based operation I’d like to talk about, and that’s on converting a scale from something that is on the interval or ratio scale, like a numeric grade, into one which is categorical.
* Now, this might seem a bit counter intuitive to you, since you are losing information about the value.
* But it’s commonly done in a couple of places. For instance, if you are visualizing the frequencies of categories, this can be an extremely useful approach, and histograms are regularly used with converted interval or ratio data.
* In addition, if you’re using a machine learning classification approach on data, you need to be using categorical data, so reducing dimensionality may be useful just to apply a given technique.
* Pandas has a function called cut which takes as an argument some array-like structure like a column of a dataframe or a series. It also takes a number of bins to be used, and all bins are kept at equal spacing.
* Let’s go back to our census data for an example. We saw that we could group by state, then aggregate to get a list of the average county size by state. If we further apply cut to this with, say, ten bins, we can see the states listed as categoricals using the average county size.

# let's bring in numpy

import numpy as np

# Now we read in our dataset

df=pd.read\_csv("datasets/census.csv")

# And we reduce this to country data

df=df[df['SUMLEV']==50]

# And for a few groups

df=df.set\_index('STNAME').groupby(level=0)['CENSUS2010POP'].agg(np.average)

df.head()

* Now if we just want to make "bins" of each of these, we can use cut()

pd.cut(df,10)

* Here we see that states like alabama and alaska fall into the same category, while california and the disctrict of columbia fall in a very different category.
* Now, cutting is just one way to build categories from your data, and there are many other methods.
* For instance, cut gives you interval data, where the spacing between each category is equal sized.
* But sometimes you want to form categories based on frequency – you want the number of items in each bin to be the same, instead of the spacing between bins. It really depends on what the shape of your data is, and what you’re planning to do with it.

**Pivot Tables**

* A pivot table is a way of summarizing data in a DataFrame for a particular purpose. It makes heavy use of the aggregation function.
* A pivot table is itself a DataFrame, where
  + the rows represent one variable that you're interested in,
  + the columns another, and the cell's some aggregate value.
* A pivot table also tends to includes marginal values as well, which are the sums for each column and row. This allows you to be able to see the relationship between two variables at just a glance.

**Pivot table example in python**

# Lets take a look at pivot tables in pandas

import pandas as pd

import numpy as np

* Here we have the Times Higher Education World University Ranking dataset, which is one of the most influential university measures. Let's import the dataset and see what it looks like

df = pd.read\_csv('datasets/cwurData.csv')

df.head()

* Here we can see each institution's rank, country, quality of education, other metrics, and overall score.
* Let's say we want to create a new column called Rank\_Level, where institutions with world ranking 1-100 are categorized as first tier and those with world ranking 101 - 200 are second tier, ranking 201 - 300 are third tier, after 301 is other top universities.
* I'm going to create a function called create\_category which will operate on the first column in the dataframe, world\_rank

def create\_category(ranking):

# Since the rank is just an integer, I'll just do a bunch of if/elif statements

if (ranking >= 1) & (ranking <= 100):

return "First Tier Top Unversity"

elif (ranking >= 101) & (ranking <= 200):

return "Second Tier Top Unversity"

elif (ranking >= 201) & (ranking <= 300):

return "Third Tier Top Unversity"

return "Other Top Unversity"

# Now we can apply this to a single column of data to create a new series

df['Rank\_Level'] = df['world\_rank'].apply(lambda x: create\_category(x))

# And let’s look at the result

df.head()

* A pivot table allows us to pivot out one of these columns a new column headers and compare it against another column as row indices.
* Let's say we want to compare rank level versus country of the universities and we want to compare in terms of overall score
* To do this, we tell Pandas we want the values to be Score, and index to be the country and the columns to be the rank levels.
* Then we specify that the aggregation function, and here we'll use the NumPy mean to get the average rating for universities in that country

df.pivot\_table(values='score', index='country', columns='Rank\_Level', aggfunc=[np.mean]).head()

* We can see a hierarchical dataframe where the index, or rows, are by country and the columns have two levels, the top level indicating that the mean value is being used and the second level being our ranks. In this example we only have one variable, the mean, that we are looking at, so we don't really need a heirarchical index.
* We notice that there are some NaN values, for example, the first row, Argentia. The NaN values indicate that Argentia has only observations in the "Other Top Unversities" category
* Now, pivot tables aren't limited to one function that you might want to apply. You can pass a named parameter, aggfunc, which is a list of the different functions to apply, and pandas will provide you with the result using hierarchical column names.
* Let's try that same query, but pass in the max() function too

df.pivot\_table(values='score', index='country', columns='Rank\_Level', aggfunc=[np.mean, np.max]).head()

* So now we see we have both the mean and the max. As mentioned earlier, we can also summarize the values within a given top level column.
* For instance, if we want to see an overall average for the country for the mean and we want to see the max of the max, we can indicate that we want pandas to provide marginal values

df.pivot\_table(values='score', index='country', columns='Rank\_Level', aggfunc=[np.mean, np.max],

margins=True).head()

* A pivot table is just a multi-level dataframe, and we can access series or cells in the dataframe in a similar way as we do so for a regular dataframe.
* Let's create a new dataframe from our previous example

new\_df=df.pivot\_table(values='score', index='country', columns='Rank\_Level', aggfunc=[np.mean, np.max],

margins=True)

# Now let's look at the index

print(new\_df.index)

# And let's look at the columns

print(new\_df.columns)

* We can see the columns are hierarchical. The top level column indices have two categories: mean and max, and the lower level column indices have four categories, which are the four rank levels.
* How would we query this if we want to get the average scores of First Tier Top Unversity levels in each country? We would just need to make two dataframe projections, the first for the mean, then the second for the top tier

new\_df['mean']['First Tier Top Unversity'].head()

* We can see that the output is a series object which we can confirm by printing the type. Remember that when you project a single column of values out of a DataFrame you get a series.

type(new\_df['mean']['First Tier Top Unversity'])

* What if we want to find the country that has the maximum average score on First Tier Top University level?

# We can use the idxmax() function.

new\_df['mean']['First Tier Top Unversity'].idxmax()

* Now, the idxmax() function isn't special for pivot tables, it's a built in function to the Series object.
* We don't have time to go over all pandas functions and attributes, and I want to encourage you to explore the API to learn more deeply what is available to you.
* If you want to achieve a different shape of your pivot table, you can do so with the stack and unstack functions.
* Stacking is pivoting the lowermost column index to become the innermost row index. Unstacking is the inverse of stacking, pivoting the innermost row index to become the lowermost column index. An example will help make this clear

# Let's look at our pivot table first to refresh what it looks like

new\_df.head()

* Now let's try stacking, this should move the lowermost column, so the tiers of the university rankings, to the inner most row

new\_df=new\_df.stack()

new\_df.head()

* In the original pivot table, rank levels are the lowermost column, after stacking, rank levels become the innermost index, appearing to the right after country

# Now let's try unstacking

new\_df.unstack().head()

* That seems to restore our dataframe to its original shape. What do you think would happen if we unstacked twice in a row?

new\_df.unstack().unstack().head()

* It unstacks everything into one single column.

**Date Functionality**

In today's lecture, where we'll be looking at the time series and date functionally in pandas.

* Manipulating dates and time is quite flexible in Pandas and thus allows us to conduct more analysis such as time series analysis, which we will talk about soon.
* Actually, pandas was originally created by Wed McKinney to handle date and time data when he worked as a consultant for hedge funds.

# Let's bring in pandas and numpy as usual

import pandas as pd

import numpy as np

* Pandas has four main time related classes.
  + Timestamp,
  + DatetimeIndex,
  + Period, and
  + PeriodIndex

**Timestamp**

* First, let's look at Timestamp. It represents a single timestamp and associates values with points in time.
* For example, let's create a timestamp using a string 9/1/2019 10:05AM, and here we have our timestamp.

# Timestamp is interchangeable with Python's datetime in most cases.

pd.Timestamp('9/1/2019 10:05AM')

* We can also create a timestamp by passing multiple parameters such as year, month, date, hour, minute, separately

pd.Timestamp(2019, 12, 20, 0, 0)

* Timestamp also has some useful attributes, such as isoweekday(), which shows the weekday of the timestamp. Note that 1 represents Monday and 7 represents Sunday

pd.Timestamp(2019, 12, 20, 0, 0).isoweekday()

* You can find extract the specific year, month, day, hour, minute, second from a timestamp

pd.Timestamp(2019, 12, 20, 5, 2,23).second

**Period**

* Suppose we weren't interested in a specific point in time and instead wanted a span of time. This is where the Period class comes into play.
* Period represents a single time span, such as a specific day or month.
* Here we are creating a period that is January 2016,

pd.Period('1/2016')

* You'll notice when we print that out that the granularity of the period is M for month, since that was the finest grained piece we provided. Here's an example of a period that is March 5th, 2016.

pd.Period('3/5/2016')

* Period objects represent the full timespan that you specify. Arithmetic on period is very easy and intuitive, for instance, if we want to find out 5 months after January 2016, we simply plus 5

pd.Period('1/2016') + 5

* From the result, you can see we get June 2016. If we want to find out two days before March 5th 2016, we simply subtract 2

pd.Period('3/5/2016') – 2

* The key here is that the period object encapsulates the granularity for arithmetic

**DatetimeIndex and PeriodIndex**

* The index of a timestamp is DatetimeIndex. Let's look at a quick example.
* First, let's create our example series t1, we'll use the Timestamp of September 1st, 2nd and 3rd of 2016. When we look at the series, each Timestamp is the index and has a value associated with it, in this case, a, b and c.

t1 = pd.Series(list('abc'), [pd.Timestamp('2016-09-01'), pd.Timestamp('2016-09-02'),

pd.Timestamp('2016-09-03')])

t1

* Looking at the type of our series index, we see that it's DatetimeIndex.

type(t1.index)

* Similarly, we can create a period-based index as well.

t2 = pd.Series(list('def'), [pd.Period('2016-09'), pd.Period('2016-10'),

pd.Period('2016-11')])

t2

* Looking at the type of the ts2.index, we can see that it's PeriodIndex.

type(t2.index)

**Converting to Datetime**

* Now, let's look into how to convert to Datetime. Suppose we have a list of dates as strings and we want to create a new dataframe
* I'm going to try a bunch of different date formats

d1 = ['2 June 2013', 'Aug 29, 2014', '2015-06-26', '7/12/16']

* And just some random data

ts3 = pd.DataFrame(np.random.randint(10, 100, (4,2)), index=d1,

columns=list('ab'))

ts3

* Using pandas to\_datetime, pandas will try to convert these to Datetime and put them in a standard format.

ts3.index = pd.to\_datetime(ts3.index)

ts3

* to\_datetime also() has options to change the date parse order. For example, we can pass in the argument dayfirst = True to parse the date in European date.

pd.to\_datetime('4.7.12', dayfirst=True)

**Timedelta**

* Timedeltas are differences in times. This is not the same as a period, but conceptually similar.
* For instance, if we want to take the difference between September 3rd and September 1st, we get a Timedelta of two days.

pd.Timestamp('9/3/2016')-pd.Timestamp('9/1/2016')

* We can also do something like find what the date and time is for 12 days and three hours past September 2nd, at 8:10 AM.

pd.Timestamp('9/2/2016 8:10AM') + pd.Timedelta('12D 3H')

**Offset**

* Offset is similar to timedelta, but it follows specific calendar duration rules.
* Offset allows flexibility in terms of types of time intervals. Besides hour, day, week, month, etc it also has business day, end of month, semi month begin etc
* Let's create a timestamp, and see what day is that

pd.Timestamp('9/4/2016').weekday()

* Now we can now add the timestamp with a week ahead

pd.Timestamp('9/4/2016') + pd.offsets.Week()

* Now let's try to do the month end, then we would have the last day of Septemer

pd.Timestamp('9/4/2016') + pd.offsets.MonthEnd()

**Working with Dates in a Dataframe**

* Next, let's look at a few tricks for working with dates in a DataFrame.
* Suppose we want to look at nine measurements, taken bi-weekly, every Sunday, starting in October 2016. Using date\_range, we can create this DatetimeIndex.
* In data\_range, we have to either specify the start or end date. If it is not explicitly specified, by default, the date is considered the start date. Then we have to specify number of periods, and a frequency.
* Here, we set it to "2W-SUN", which means biweekly on Sunday

dates = pd.date\_range('10-01-2016', periods=9, freq='2W-SUN')

dates

* There are many other frequencies that you can specify. For example, you can do business day

pd.date\_range('10-01-2016', periods=9, freq='B')

* Or you can do quarterly, with the quarter start in June

pd.date\_range('04-01-2016', periods=12, freq='QS-JUN')

* Now, let's go back to our weekly on Sunday example and create a DataFrame using these dates, and some random data, and see what we can do with it.

dates = pd.date\_range('10-01-2016', periods=9, freq='2W-SUN')

df = pd.DataFrame({'Count 1': 100 + np.random.randint(-5, 10, 9).cumsum(),

'Count 2': 120 + np.random.randint(-5, 10, 9)}, index=dates)

df

* First, we can check what day of the week a specific date is. For example, here we can see that all the dates in our index are on a Sunday. Which matches the frequency that we set

df.index.weekday

* We can also use diff() to find the difference between each date's value.

df.diff()

* Suppose we want to know what the mean count is for each month in our DataFrame. We can do this using resample.
* Converting from a higher frequency from a lower frequency is called downsampling (we'll talk about this in a moment)

df.resample('M').mean()

* Now let's talk about datetime indexing and slicing, which is a wonderful feature of the pandas DataFrame.
* For instance, we can use partial string indexing to find values from a particular year,

df['2017']

* Or we can do it from a particular month

df['2016-12']

* Or we can even slice on a range of dates For example, here we only want the values from December 2016 onwards.

df['2016-12':]

df['2016']

**Week 4: Beyond data manipulation**

**Basic Statistical Testing**

* We use statistics in a lot of different ways in data science, and on this lecture, I want to refresh your knowledge of hypothesis testing, which is a core data analysis activity behind experimentation.
* The goal of hypothesis testing is to determine if, for instance, the two different conditions we have in an experiment have resulted in different impacts

# Let's import our usual numpy and pandas libraries

import numpy as np

import pandas as pd

# Now let's bring in some new libraries from scipy

from scipy import stats

* Now, scipy is an interesting collection of libraries for data science and you'll use most or perhaps all of these libraries. It includes numpy and pandas, but also plotting libraries such as matplotlib, and a number of scientific library functions as well
* When we do hypothesis testing, we actually have two statements of interest:
  + the first is our actual explanation, which we call the alternative hypothesis,
  + and the second is that the explanation we have is not sufficient, and we call this the null hypothesis.
* Our actual testing method is to determine whether the null hypothesis is true or not.
* If we find that **there is a difference between groups**, then we can **reject** the null hypothesis and we **accept** our alternative.

# Let's see an example of this; we're going to use some grade data

df=pd.read\_csv ('datasets/grades.csv')

df.head()

* If we take a look at the data frame inside, we see we have six different assignments. Lets look at some summary statistics for this DataFrame

print("There are {} rows and {} columns".format(df.shape[0], df.shape[1]))

* For the purpose of this lecture, let's segment this population into two pieces.
* Let's say those who finish the first assignment by the end of December 2015, we'll call them early finishers, and those who finish it sometime after that, we'll call them late finishers.

early\_finishers=df[pd.to\_datetime(df['assignment1\_submission']) < '2016']

early\_finishers.head()

* First, the dataframe df and the early\_finishers share index values, so I really just want everything in the df which is not in early\_finishers

late\_finishers=df[~df.index.isin(early\_finishers.index)]

late\_finishers.head()

* There are lots of other ways to do this. For instance, you could just copy and paste the first projection and change the sign from less than to greater than or equal to. This is ok, but if you decide you want to change the date down the road you have to remember to change it in two places.
* You could also do a join of the dataframe df with early\_finishers - if you do a left join you only keep the items in the left dataframe, so this would have been a good answer.
* You also could have written a function that determines if someone is early or late, and then called .apply() on the dataframe and added a new column to the dataframe. This is a pretty reasonable answer as well.
* As you've seen, the pandas data frame object has a variety of statistical functions associated with it. If we call the mean function directly on the data frame, we see that each of the means for the assignments are calculated. Let's compare the means for our two populations

print(early\_finishers['assignment1\_grade'].mean())

print(late\_finishers['assignment1\_grade'].mean())

* Ok, these look pretty similar. But, are they the same? What do we mean by similar? This is where the students' t-test comes in. It allows us to form the alternative hypothesis ("These are different") as well as the null hypothesis ("These are the same") and then test that null hypothesis.
* When doing hypothesis testing, we have to choose a significance level as a threshold for how much of a chance we're willing to accept. This significance level is typically called alpha.
* For this example, let's use a threshold of 0.05 for our alpha or 5%. Now this is a commonly used number but it's really quite arbitrary.
* The SciPy library contains a number of different statistical tests and forms a basis for hypothesis testing in Python and we're going to use the ttest\_ind() function which does an independent t-test (meaning the populations are not related to one another). The result of ttest\_index() are the t-statistic and a p-value.
* It's this latter value, the probability, which is most important to us, as it indicates the chance (between 0 and 1) of our null hypothesis being True.

# Let's bring in our ttest\_ind function

from scipy.stats import ttest\_ind

# Let's run this function with our two populations, looking at the assignment 1 grades

ttest\_ind(early\_finishers['assignment1\_grade'], late\_finishers['assignment1\_grade'])

* So here we see that the probability is 0.18, and this is above our alpha value of 0.05. This means that we cannot reject the null hypothesis. The null hypothesis was that the two populations are the same, and we don't have enough certainty in our evidence (because it is greater than alpha) to come to a conclusion to the contrary. This doesn't mean that we have proven the populations are the same.
* Why don't we check the other assignment grades?

print(ttest\_ind(early\_finishers['assignment2\_grade'], late\_finishers['assignment2\_grade']))

print(ttest\_ind(early\_finishers['assignment3\_grade'], late\_finishers['assignment3\_grade']))

print(ttest\_ind(early\_finishers['assignment4\_grade'], late\_finishers['assignment4\_grade']))

print(ttest\_ind(early\_finishers['assignment5\_grade'], late\_finishers['assignment5\_grade']))

print(ttest\_ind(early\_finishers['assignment6\_grade'], late\_finishers['assignment6\_grade']))

* Ok, so it looks like in this data we do not have enough evidence to suggest the populations differ with respect to grade. Let's take a look at those p-values for a moment though, because they are saying things that can inform experimental design down the road.
* For instance, one of the assignments, assignment 3, has a p-value around 0.1. This means that if we accepted a level of chance similarity of 11% this would have been considered statistically significant.
* As a research, this would suggest to me that there is something here worth considering following up on. For instance, if we had a small number of participants (we don't) or if there was something unique about this assignment as it relates to our experiment (whatever it was) then there may be follow-up experiments we could run.
* P-values have come under fire recently for being insufficient for telling us enough about the interactions which are happening, and two other techniques, confidence intervalues and bayesian analyses, are being used more regularly. One issue with p-values is that as you run more tests you are likely to get a value which is statistically significant just by chance.

# Let’s see a simulation of this. First, let’s create a data frame of 100 columns, each with 100 numbers

df1=pd.DataFrame([np.random.random(100) for x in range(100)])

df1.head()

# let's create a second dataframe

df2=pd.DataFrame([np.random.random(100) for x in range(100)])

* Are these two DataFrames the same? Maybe a better question is, for a given row inside of df1, is it the same as the row inside df2?
* Let's take a look. Let's say our critical value is 0.1, or and alpha of 10%. And we're going to compare each column in df1 to the same numbered column in df2. And we'll report when the p-value isn't less than 10%, which means that we have sufficient evidence to say that the columns are different.
* Let's write this in a function called test\_columns

def test\_columns(alpha=0.1):

# I want to keep track of how many differ

num\_diff=0

# And now we can just iterate over the columns

for col in df1.columns:

# we can run out ttest\_ind between the two dataframes

teststat,pval=ttest\_ind(df1[col],df2[col])

# and we check the pvalue versus the alpha

if pval<=alpha:

# And now we'll just print out if they are different and increment the num\_diff

print("Col {} is statistically significantly different at alpha={}, pval={}".format(col,alpha,pval))

num\_diff=num\_diff+1

# and let's print out some summary stats

print("Total number different was {}, which is {}%".format(num\_diff,float(num\_diff)/len(df1.columns)\*100))

# And now lets actually run this

test\_columns()

* Interesting, so we see that there are a bunch of columns that are different! In fact, that number looks a lot like the alpha value we chose. So what's going on - shouldn't all of the columns be the same?
* Remember that all the ttest does is check if two sets are similar given some level of confidence, in our case, 10%. The more random comparisons you do, the more will just happen to be the same by chance. In this example, we checked 100 columns, so we would expect there to be roughly 10 of them if our alpha was 0.1.

# We can test some other alpha values as well

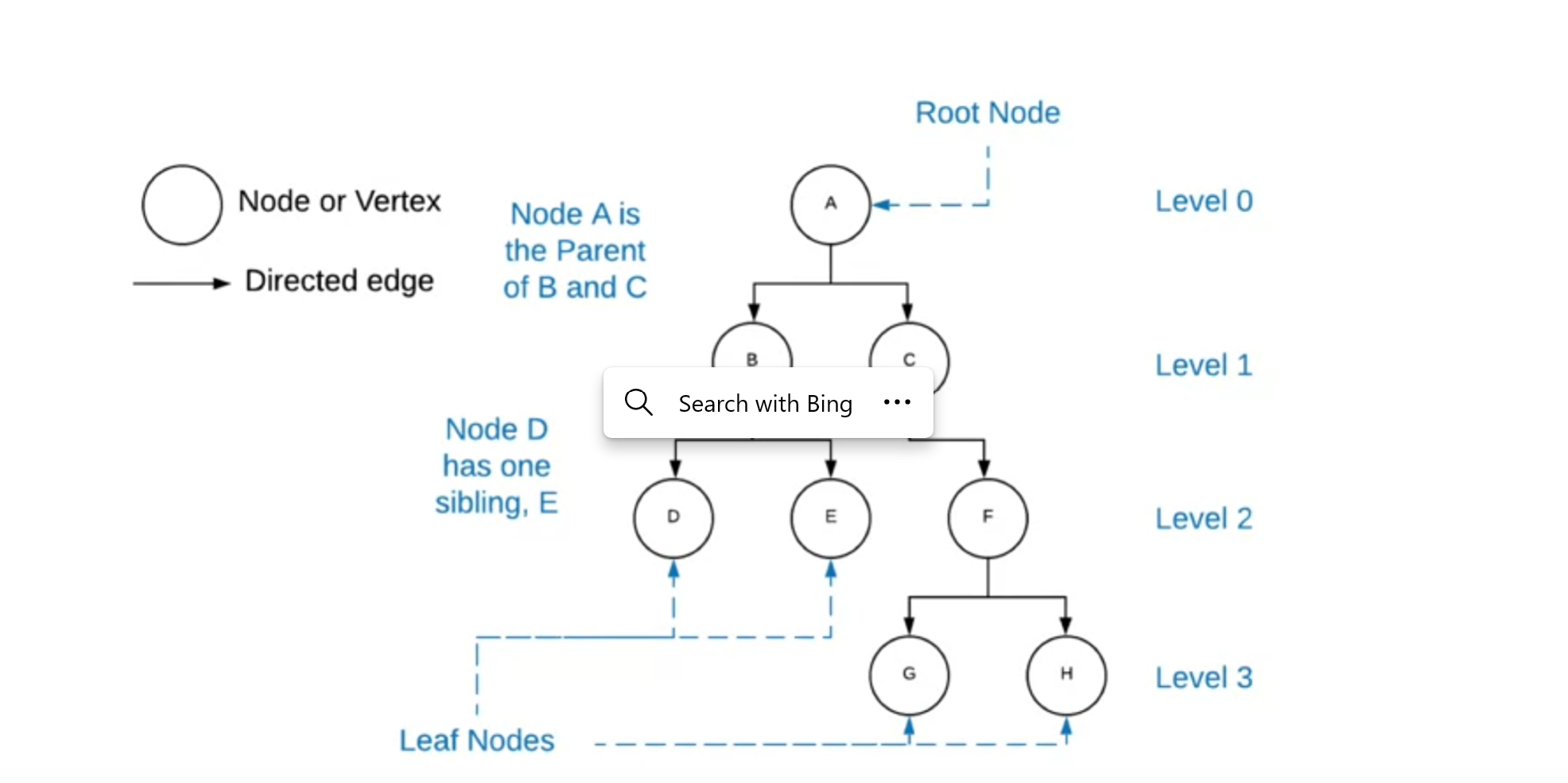
test\_columns(0.05)

* So, keep this in mind when you are doing statistical tests like the t-test which has a p-value. Understand that this p-value isn't magic, that it's a threshold for you when reporting results and trying to answer your hypothesis.
* What's a reasonable threshold? Depends on your question, and you need to engage domain experts to better understand what they would consider significant.
* Just for fun, lets recreate that second dataframe using a non-normal distribution, I'll arbitrarily chose chi squared

df2=pd.DataFrame([np.random.chisquare(df=1,size=100) for x in range(100)])

test\_columns()

**Other forms of structured data**



* We can think of networks as being made up of nodes and that these could represent anything, people, sports teams, planets.
* The nodes are connected through edges which may be directed or undirected.
* Sometimes network is referred to as a graph, and sometimes nodes are referred to as vertices.

