Lecture 3

Scientific Programming: review of numpy, overview of scipy tools, and an intro to pandas

HW 2 Review

1a Change the color of the line

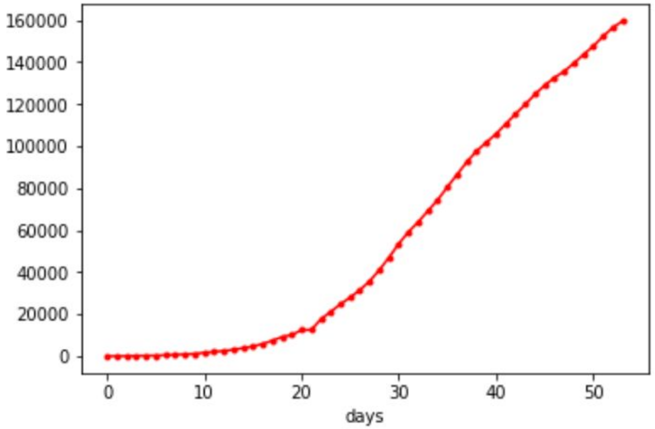
In [ ]:
```
# your code here
```

1a Change the color of the line

In [ ]:
```
# your code here
```

1a Change the color of the line

In [10]:
```
plt.plot(data_italy.values,'.-',color='r')
plt.xlabel('days')
plt.show()
```



1b Change the thickness of the line

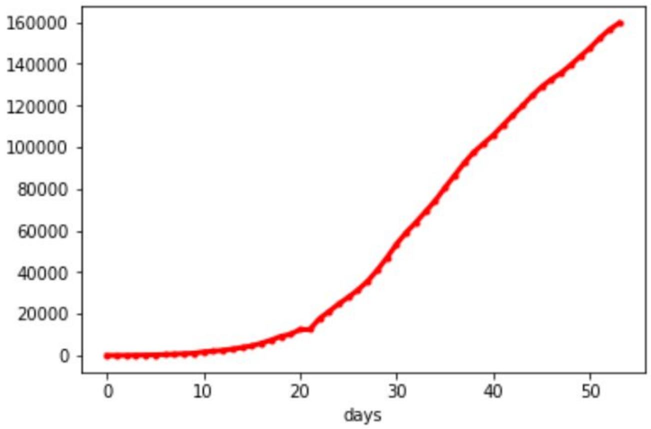
In [ ]:
```
# your code here
```

1b Change the thickness of the line

In [ ]:
```
# your code here
```

1b Change the thickness of the line

In [11]:
```
plt.plot(data_italy.values,'.-',color='r',linewidth=3)
plt.xlabel('days')
plt.show()
```



1c Add a label the y-axis

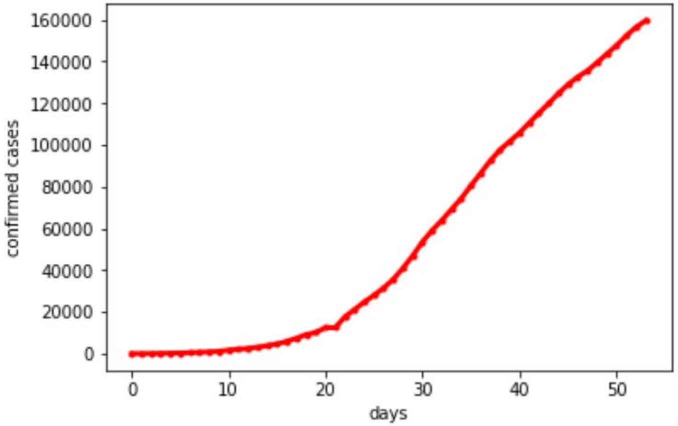
In [ ]:
```
# your code here
```

1c Add a label the y-axis

In [ ]:
```
# your code here
```

1c Add a label the y-axis

In [13]:
```
plt.plot(data_italy.values,'.-',color='r',linewidth=3)
plt.xlabel('days')
plt.ylabel('confirmed cases')
plt.show()
```

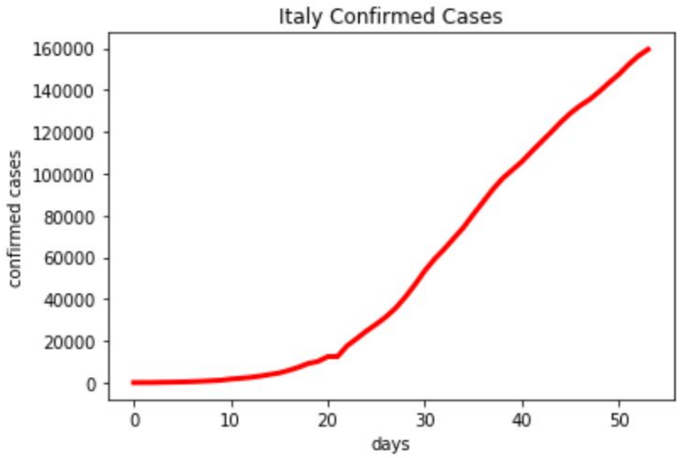


1d Add a title

In [ ]:
```
# your code here
```

1d Add a title

In [15]:
```
plt.plot(data_italy.values,'-',color='r',linewidth=3)
plt.xlabel('days')
plt.ylabel('confirmed cases')
plt.title('Italy Confirmed Cases')
plt.show()
```

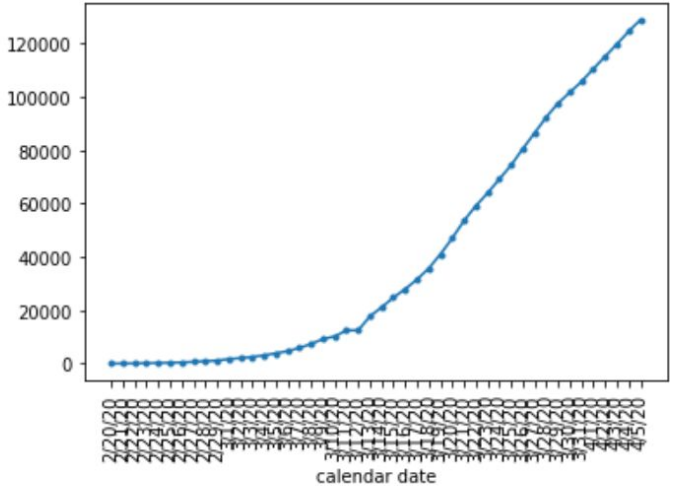


1e Resize the figure

Note that we can plot with the calendar date using `data_italy.index`

In [33]:

```
# resize this figure here


plt.plot(data_italy.index,data_italy.values,'.-')
plt.xticks(rotation=90) # rotate the xticks so text is not so tight
plt.xlabel('calendar date')
plt.show()
```

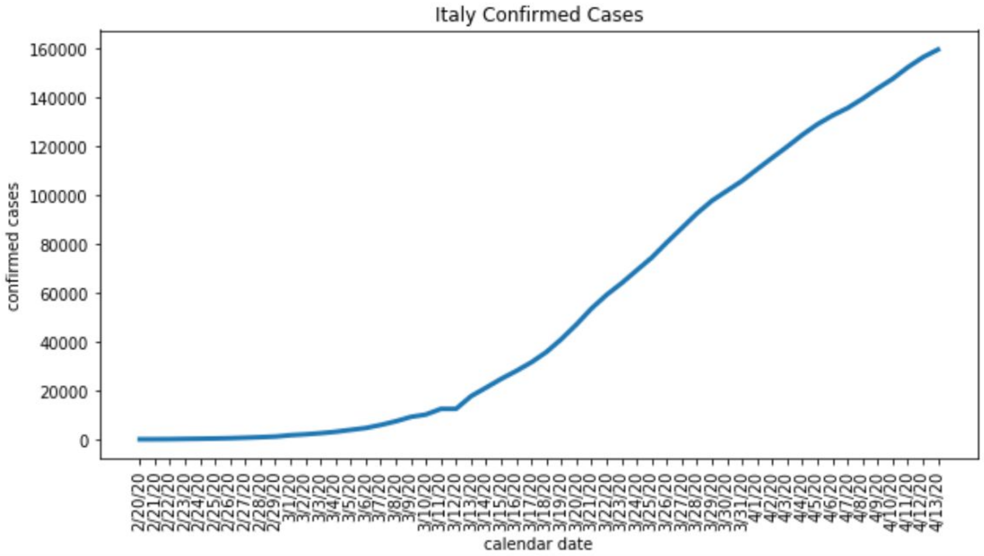


1e Resize the figure

Note that we can plot with the calendar date using `data_italy.index`

In [18]:

```
# resize this figure here
fig = plt.figure(figsize=(10,5))

plt.plot(data_italy.index,data_italy.values,'-',linewidth=3)
plt.xticks(rotation=90) # rotate the xticks so text is not so tight
plt.xlabel('calendar date')
plt.ylabel('confirmed cases')
plt.title('Italy Confirmed Cases')
plt.show()
```



2a: Plot Italy and US data in two axes

Use subplots and assign separate colors to each country. Set the same limits on the y-axis for both subplots

In [ ]:
```
# use
fig,ax = plt.subplots(1,2,figsize=(15,5)) # or whatever dimensions you like
```

2a: Plot Italy and US data in two axes

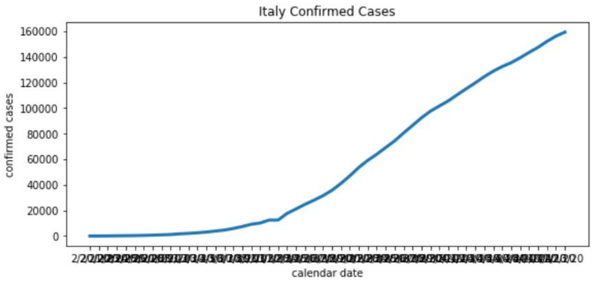
Use subplots and assign separate colors to each country. Set the same limits on the y-axis for both subplots

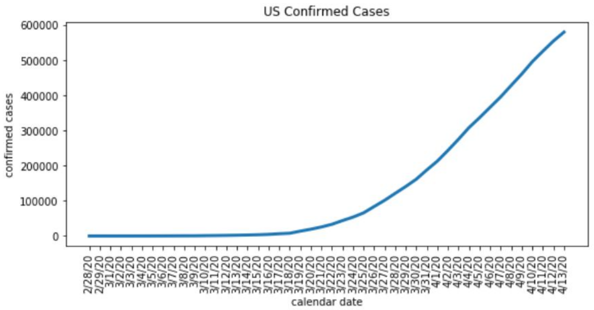
In [ ]:
```python
# use
fig,ax = plt.subplots(1,2,figsize=(15,5)) # or whatever dimensions you like
```

2a: Plot Italy and US data in two axes

Use subplots and assign separate colors to each country. Set the same limits on the y-axis for both subplots

In [34]:
```python
# use
fig,ax = plt.subplots(1,2,figsize=(20,4)) # or whatever dimensions you like

ax[0].plot(data_italy.index,data_italy.values,'-',linewidth=3)
ax[1].plot(data_us.index,data_us.values,'-',linewidth=3)

ax[0].set_title('Italy Confirmed Cases')
ax[1].set_title('US Confirmed Cases')

for a in ax:

    a.set_xlabel('calendar date')
    a.set_ylabel('confirmed cases')
    plt.xticks(rotation=90) # rotate the xticks so text is not so tight << this is tricky


plt.show()
```





2b: Plot Italy and US data in the same plot

Use the same colors as above, and include a legend

In [ ]:
```
# your code here
```

2b: Plot Italy and US data in the same plot

Use the same colors as above, and include a legend

In [ ]:

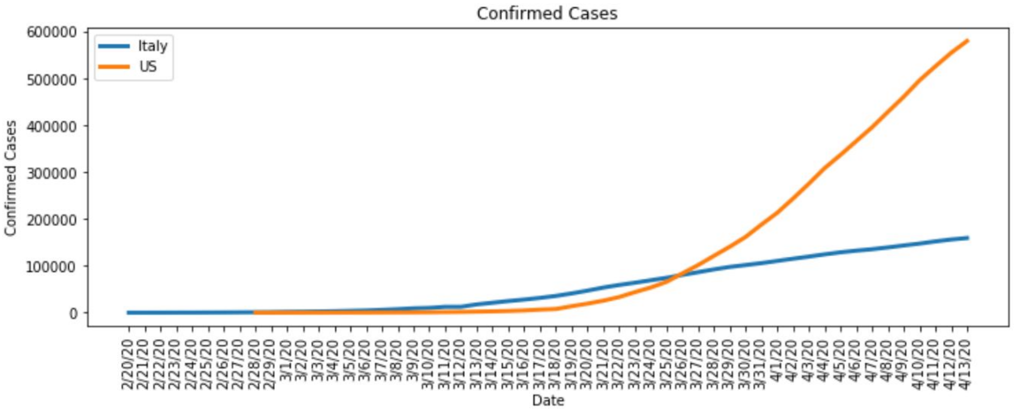
```python
# your code here
```


2b: Plot Italy and US data in the same plot

Use the same colors as above, and include a legend

In [42]:

```python
fig,ax = plt.subplots(1,1,figsize=(12,4)) # or whatever dimensions you like

ax.plot(data_italy.index,data_italy.values,'-',linewidth=3,label='Italy')
ax.plot(data_us.index,data_us.values,'-',linewidth=3,label='US')

ax.set_title('Confirmed Cases')
ax.set_xlabel('Date')
ax.set_ylabel('Confirmed Cases')
plt.xticks(rotation=90)
plt.legend()
plt.show()
```



2c Plot Italy and US data on log axis

Note that we only want the y-axis to be logarithmic

In [ ]:
```
# your code here
```

2c Plot Italy and US data on log axis

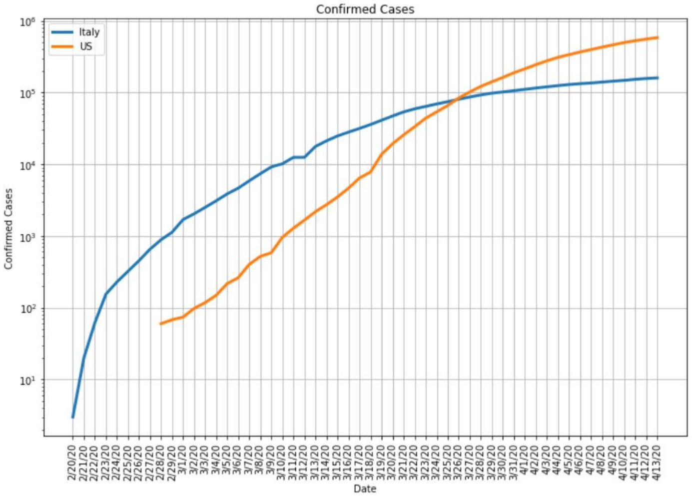
Note that we only want the y-axis to be logarithmic

In [ ]:
```python
# your code here
```

2c Plot Italy and US data on log axis

Note that we only want the y-axis to be logarithmic

In [48]:
```python
fig,ax = plt.subplots(1,1,figsize=(12,8)) # or whatever dimensions you like

ax.plot(data_italy.index,data_italy.values,'-',linewidth=3,label='Italy')
ax.plot(data_us.index,data_us.values,'-',linewidth=3,label='US')

ax.set_title('Confirmed Cases')
ax.set_xlabel('Date')
ax.set_ylabel('Confirmed Cases')
plt.xticks(rotation=90)
plt.yscale('log')
plt.grid(which='major')
plt.legend()
plt.show()
```



2d Make the same plot as above, but using a `for` loop

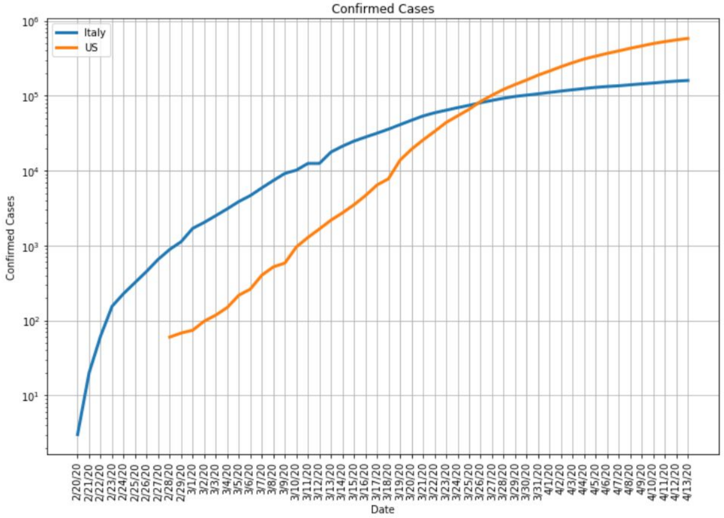
In [ ]:
```
# your code here
```

2d Make the same plot as above, but using a `for` loop

In [ ]:
```
# your code here
```

2d Make the same plot as above, but using a `for` loop

In [52]:
```
data_list = [data_italy,data_us]
data_names = ['Italy','US']

fig,ax = plt.subplots(1,1,figsize=(12,8)) # or whatever dimensions you like

for data,name in zip(data_list,data_names):
    ax.plot(data.index,data.values,'-',linewidth=3,label=name)

ax.set_title('Confirmed Cases')
ax.set_xlabel('Date')
ax.set_ylabel('Confirmed Cases')
plt.xticks(rotation=90)
plt.yscale('log')
plt.grid(which='major')
plt.legend()
plt.show()
```



Part 3 Plot Data for 5 countries

Use a for loop, and include x/y labels and a legend. Also save your figure as a jpg and share with friends. You are now a Python datascientist

In [ ]:

```
# save your figure
# fig.savefig('myplot.jpg')
```

Part 3 Plot Data for 5 countries

Use a for loop, and include x/y labels and a legend. Also save your figure as a jpg and share with friends. You are now a Python datascientist

In [ ]:

```
# save your figure
# fig.savefig('myplot.jpg')
```

Part 3 Plot Data for 5 countries

Use a for loop, and include x/y labels and a legend. Also save your figure as a jpg and share with friends. You are now a Python datascientist

In [67]:

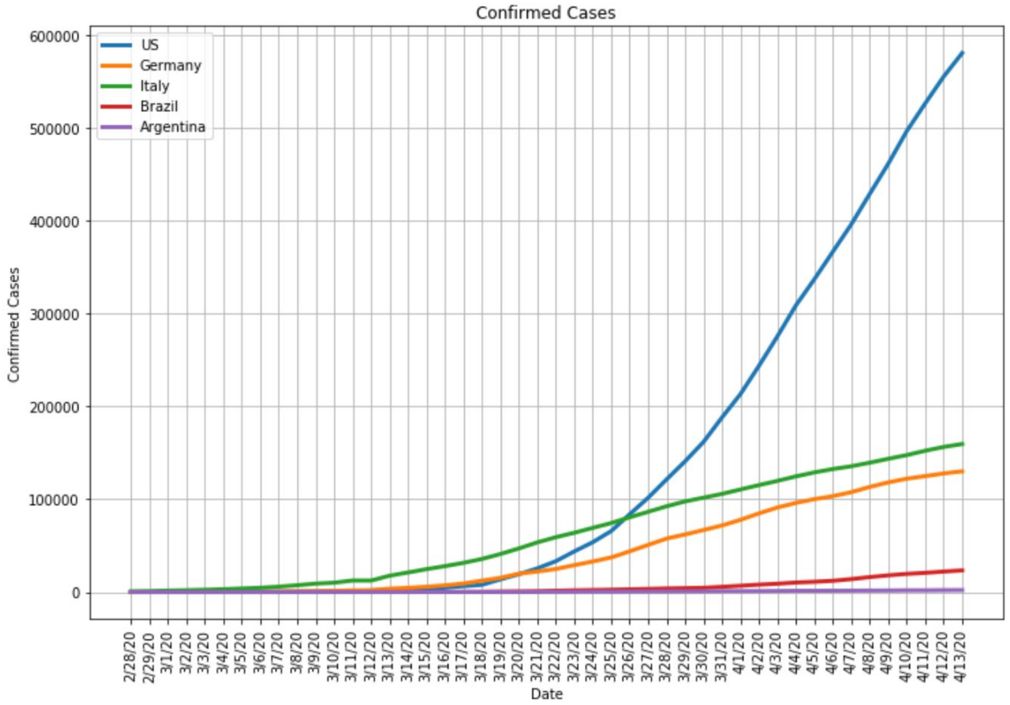
```
country_names = ['US', 'Germany','Italy','Brazil','Argentina']

fig,ax = plt.subplots(1,1,figsize=(12,8)) # or whatever dimensions you like

for name in country_names:

    df_t = df[df['Country/Region'] == name]
    df_t = df_t.set_index('Country/Region', drop = True) # set pandas dataframe index to country
    df_t = df_t.drop(columns=['Lat','Long','Province/State'])

    data = df_t.loc[name,'2/28/20':'4/13/20']



    ax.plot(data.index,data.values,'-',linewidth=3,label=name)

ax.set_title('Confirmed Cases')
ax.set_xlabel('Date')
ax.set_ylabel('Confirmed Cases')
plt.xticks(rotation=90)
#plt.yscale('log')
plt.grid(which='major')
plt.legend()
plt.show()

# save your figure
# fig.savefig('myplot.jpg')
```

Part 3 Plot Data for 5 countries

Use a for loop, and include x/y labels and a legend. Also save your figure as a jpg and share with friends. You are now a Python datascientist

In [ ]:

```
# save your figure
# fig.savefig('myplot.jpg')
```



Today’s lecture: review of numpy arrays, brief overview of scipy, intro to pandas

In the last class, we were introduced to numpy arrays.

To get back up to speed, let's try a few practice problems together.

Create a 1D numpy array "my_numbers" containing the numbers 1 through 10 in order.
What different ways are there to do this?

```
In [ ]: my_numbers = np.
```

A few options - np.array( ), np.arange( ), and np.linspace( )

***Please give a thumbs up when you've completed the problem*

Numpy arrays: indexing/slicing review + a few more tools

Create a 1D numpy array "my_numbers" containing the numbers 1 through 10 in order.
What different ways are there to do this?

```
In [23]: my_numbers = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
         print(my_numbers)

[ 1  2  3  4  5  6  7  8  9 10]

In [22]: my_numbers = np.arange(1,11,1)
         print(my_numbers)

[ 1  2  3  4  5  6  7  8  9 10]

In [20]: my_numbers = np.linspace(1, 10, 10, endpoint=True)
         print(my_numbers)

[  1.   2.   3.   4.   5.   6.   7.   8.   9.  10.]
```

Numpy arrays: indexing/slicing review + a few more tools

Now, using numpy’s reshape( ) method, make my_numbers into a 2 x 5 array

Numpy arrays: indexing/slicing review + a few more tools

Now, using numpy's reshape( ) method, make my_numbers into a 2 x 5 array

```
In [36]: my_numbers = my_numbers.reshape(2,5)
         print(my_numbers)

[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]

In [37]: my_numbers = np.reshape(my_numbers,(2,5))
         print(my_numbers)

[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
```

Using your new, reshaped my_numbers array, use array indexing to put out the second value of the second element (the number 8)

Numpy arrays: indexing/slicing review + a few more tools

Using your new, reshaped my_numbers array, use array indexing to print out the second value of the second element (the number 8)

```
In [40]: print(my_numbers[1,2])
Out[40]: 8
```

Using slicing techniques, print every alternating value from the first element of my_numbers

Numpy arrays: indexing/slicing review + a few more tools

Using your new, reshaped my_numbers array, use array indexing to print out the second value of the second element (the number 8)

```
In [40]: print(my_numbers[1,2])
Out[40]: 8
```

Using slicing techniques, print every alternating value from the first element of my_numbers

```
In [49]: print(my_numbers[0,::2])
         [1 3 5]
```

Numpy arrays: indexing using numpy's where() function

Another convenient way of pulling values from a numpy array is numpy's where() function. If we wanted to pull the indices of all values in my_numbers greater than 6, we could say: `np.where(my_numbers > 6)`

```
In [145]: np.where(my_numbers > 6)
Out[145]: (array([1, 1, 1, 1]), array([1, 2, 3, 4]))
```

This output is actually two arrays containing an **index** of matching rows and columns! e.g. `my_numbers[1,1]` = 7, `my_numbers[1,2]` = 8, and so on.

How would we create a new array called new_array that contains the actual values? Try now!

Numpy arrays: indexing using numpy's where() function

Another convenient way of pulling values from a numpy array is numpy's where() function. If we wanted to pull the indices of all values in my_numbers greater than 6, we could say: `np.where(my_numbers > 6)`

```
In [145]: np.where(my_numbers > 6)

Out[145]: (array([1, 1, 1, 1]), array([1, 2, 3, 4]))
```

This output is actually two arrays containing an **index** of matching rows and columns! e.g. `my_numbers[1,1]` = 7, `my_numbers[1,2]` = 8, and so on.

How would we create a new array called new_array that contains the actual values? Try now!

```
In [149]: new_array = my_numbers[np.where(my_numbers > 6)]
          print(new_array)

          [ 7  8  9 10]
```

This output is actually two arrays containing an **index** of matching rows and columns! e.g. `my_numbers[1,1]` = 7, `my_numbers[1,2]` = 8, and so on.

How would we create a new array called new_array that contains the actual values? Try now!

Numpy arrays: indexing using numpy’s where() function

It's important to note that the **where** function has a lot more capability than just returning an index- it actually can manipulate elements using the same logic as

`[xv if c else yv for c, xv, yv in zip(condition, x, y)]`

when passed `np.where(condition, [x,y])`

```
In [166]: example_arr = np.arange(1,20,1)

          print('Before using np.where(): ','\n',example_arr)

          example_arr = np.where(example_arr < 10, example_arr, example_arr-10)

          print('After using np.where(), all indices where example_arr < 10 = False are subject to example_arr-10: ',
                '\n',example_arr)

Before using np.where():
 [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
After using np.where(), all indices where example_arr < 10 = False are subject to example_arr-10:
 [1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9]
```

Numpy arrays: indexing using boolean arrays

Finally, while it was touched on last lecture, we'd like to make the point that one can also easily index arrays by creating boolean/mask arrays using conditional statements. In this case, the functionality would look like:

new_array = old_array[old_array CONDITIONAL STATEMENT]

In the code below, we've creating an array **a** that's equal the the numbers between 1 and 20. Use a conditional statement to create a mask array with index values, and reassign a to only equaling its values that are greater than or equal to 12.

Numpy arrays: indexing using boolean arrays

Finally, while it was touched on last lecture, we'd like to make the point that one can also easily index arrays by creating boolean/mask arrays using conditional statements. In this case, the functionality would look like:

new_array = old_array[old_array CONDITIONAL STATEMENT]

In the code below, we've created an array **a** that's equal the the numbers between 1 and 20. Use a conditional statement to create a mask array with index values, and reassign a to only equaling its values that are greater than or equal to 12.

```
In [177]: a = np.arange(1,21,1)
          a = a[a>=12]
          print(a)

          [12 13 14 15 16 17 18 19 20]
```

Numpy arrays: random number generation

Numpy comes with a "random" module (np.random) that contains a number of functions for producing random numbers. Some examples include:

`np.random.random` (tuple indicating output dimensions, e.g. (2,3)) -> outputs random values from continuous distribuion over 0 to 1 in n-dimensions (specified by input tuple)

`np.random.randn` (dimension1, dimension2, dimension 3...) -> random values from 0 to 1 over normal distribution

`np.random.randint` (size, (lower bound, higher bound)) -> array of size x over specified lower/higher bounds

Check out further numpy rng features here:

https://docs.scipy.org/doc/numpy-1.15.1/reference/routines.random.html

Numpy arrays: random number generation

Using np.random.random, create two arrays of random numbers from a continuous distribution: array a (100x5), and array b (5x10)

Numpy arrays: random number generation

Using np.random.random, create two arrays of random numbers from a continuous distribution: array a (100x5), and array b (5x10)

In [154]:
```
#hint: don't forget that np.random.random takes in a tuple!
a = np.random.random((100,5))
b = np.random.random((5,10))
```

Next, using the **shape** attribute of numpy arrays, print the shape of a and the shape of b.

Numpy arrays: random number generation

Using np.random.random, create two arrays of random numbers from a continuous distribution: array a (100x5), and array b (5x10)

```
In [154]: #hint: don't forget that np.random.random takes in a tuple!
          a = np.random.random((100,5))
          b = np.random.random((5,10))
```

Next, using the **shape** attribute of numpy arrays, print the shape of a and the shape of b.

```
In [155]: #hint: attributes, unlike methods, don't require the use of parentheses!
          print(a.shape)
          print(b.shape)

          (100, 5)
          (5, 10)
```

Numpy arrays: operators and a few more useful attributes

Rather than having a "len" like lists, numpy arrays have two attributes than can help users keep track of dimensions/elements: `size` and `shape`. `size` refers to the **number of elements** within an array, while `shape` returns an array's dimensions.

Below, print the size and shape of a and b. What do you get?

Numpy arrays: operators and a few more useful attributes

Rather than having a "len" like lists, numpy arrays have two attributes than can help users keep track of dimensions/elements: `size` and `shape`. `size` refers to the **number of elements** within an array, while `shape` returns an array's dimensions.

Below, print the size and shape of a and b. What do you get?

```
In [192]: print(a.size) #a size
          print(a.shape) #a shape
          print(b.size) #b size
          print(b.shape) #b shape

500
(100, 5)
300
(2, 3, 5, 10)
```

Numpy arrays: operators and a few more useful attributes

In the last lecture, Jacob covered some operators (+, -, etc.) that can be used on numpy arrays. We just wanted to make the point that matrix multiplication can be quickly accomplished using the @ . Below, create a matrix c that is the product of a and b, and print the shapes of a, b, and c to demonstrate matrix multiplication.

Numpy arrays: operators and a few more useful attributes

In the last lecture, Jacob covered some operators (+, -, etc.) that can be used on numpy arrays. We just wanted to make the point that matrix multiplication can be quickly accomplished using the @ . Below, create a matrix c that is the product of a and b, and print the shapes of a, b, and c to demonstrate matrix multiplication.

In [186]:
```
print('a: ',a.shape)
print('b: ',b.shape)
c = a @ b
print('c: ',c.shape)
```

```
a:  (100, 5)
b:  (5, 10)
c:  (100, 10)
```

Numpy arrays: operators and a few more useful attributes

Arrays also have lots of useful methods associated with them that perform various functions, including:

`np.min()` <- returns minimum value of array

`np.max()` <-returns max value of array

`np.mean()` <-returns mean value of array

`np.astype()` <- casts all data types contained in array to specified data type, eg. =astype(int)

For a complete list of attributes and methods associated with numpy arrays, check out this documentation:

https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html#array-methods

Using this information, find the minimum value of a.

Numpy arrays: operators and a few more useful attributes

Arrays also have lots of useful methods associated with them that perform various functions, including:

`np.min()` <- returns minimum value of array

`np.max()` <-returns max value of array

`np.mean()` <-returns mean value of array

`np.astype()` <- casts all data types contained in array to specified data type, eg. =astype(int)

For a complete list of attributes and methods associated with numpy arrays, check out this documentation:

https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html#array-methods

```
In [196]: print(a.min())

4.582238062977861 4e-05
```

Numpy arrays: a note on axes in numpy arrays

We've just told you about some of the neat methods you can use with numpy arrays; however, we haven't covered situations in which you might want to find the max or min of a **single column** or **row** of an array (we refer to these as **axes**).

In this case, you'll want to specify *within* the method the axis that you would like to operate on. Let's take our 100x5 array "a" as an example. Let's say that we actually want to take the mean across the **first value** of each "column", and return a vector of length 5 containing each of those values. In this case, we would need to specify axis = 0 *as input to the function*, like so:

```
np.mean(ARRAY,axis=0)
```

Conversely, specifying "axis = 1" would return a vector of length 100 with the mean value of each "row". Below, create an array "a_mean" that's equal to the mean of array "a" across axis 0.

Numpy arrays: a note on axes in numpy arrays

We've just told you about some of the neat methods you can use with numpy arrays; however, we haven't covered situations in which you might want to find the max or min of a **single column** or **row** of an array (we refer to these as **axes**).

In this case, you'll want to specify *within* the method the axis that you would like to operate on. Let's take our 100x5 array "a" as an example. Let's say that we actually want to take the mean across the **first value** of each "column", and return a vector of length 5 containing each of those values. In this case, we would need to specify axis = 0 *as input to the function*, like so:

```
np.mean(ARRAY,axis=0)
```

Conversely, specifying "axis = 1" would return a vector of length 100 with the mean value of each "row". Below, create an array "a_mean" that's equal to the mean of array "a" across axis 0.

In [201]:
```
a_mean = np.mean(a,axis=0)
print(a_mean)
```

```
[0.5035254  0.44271019 0.51215743 0.50210546 0.49073827]
```

Numpy arrays: it doesn't end here!!

In the last few slides, we've given you an overview of

a.) numpy array objects

b.) indexing and slicing arrays

c.) **some of** the methods/attributes associated with numpy arrays

But this is absolutely not comprehensive!! We encourage you to check out the documentation on numpy array objects here: https://docs.scipy.org/doc/numpy-1.15.1/reference/arrays.html

And keep in mind - a *huge* part of being "good at Python" is being good at googling and reading documentation. Some of the homework this time around may reference attributes/methods of arrays that we didn't introduce in lecture :D :D :D


3:59

Let’s talk a little bit about SciPy.

From the documentation:

“SciPy is a collection of mathematical algorithms and convenience functions built on the NumPy extension of Python. It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data.”

The most important thing to know off the bat:

SciPy is unlike packages you’ve been exposed to thus far, as it’s organized into *subpackages*.

You don’t need to worry about the minutiae of what this means, beyond the fact that you should never really plan on typing the command “~~import scipy as~~” (I just crossed it out there so that you don’t get used to it)

SciPy subpackages

Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions

SciPy sub-packages need to be imported separately, for example:

```
>>> from scipy import linalg, optimize
```

docs.scipy.org

docs.SciPy.org is the SciPy (and numpy) bible

SciPy.org

SciPy.org

Documentation

Documentation for the core SciPy Stack projects:

- NumPy
- SciPy
- Matplotlib
- IPython
- SymPy
- pandas

The Getting started page contains links to several good tutorials dealing with the SciPy stack.

docs.scipy.org

For this reason, we're not going to be doing much demo-ing of SciPy in class - these are well-documented, plug-and-play functions!

But also the SciPy Cookbook is...also...the SciPy bible

SciPy Cookbook
latest

Search docs

- Input & Output
- Interfacing With Other Languages
- Interpolation
- Linear Algebra
- Matplotlib
- Mayavi
- Numpy
- Optimization and fitting
- Ordinary differential equations
- Other examples
- Performance
- Root finding
- Scientific GUIs
- Scientific Scripts
- Signal processing
- Outdated

Support Read the Docs!

Read the Docs v: latest

Contents » SciPy Cookbook

SciPy Cookbook

This is the "SciPy Cookbook" — a collection of various user-contributed recipes, which once lived under `wiki.scipy.org`. If you have a nice notebook you'd like to add here, or you'd like to make some other edits, please see the SciPy-CookBook repository.

Input & Output

Data Acquisition with NIDAQmx | Data acquisition with PyUL | Fortran I/O Formats | Input and output | LAS reader | Reading SPE file from CCD camera | Reading mat files | hdf5 in Matlab

Interfacing With Other Languages

C Extensions for Using NumPy Arrays | C extensions | Ctypes | F2py | Inline Weave With Basic Array Conversion (no Blitz) | SWIG Numpy examples | SWIG and Numpy | SWIG memory deallocation | f2py and numpy

Interpolation

Interpolation | Using radial basis functions for smoothing/interpolation

Linear Algebra

Rank and nullspace of a matrix

Matplotlib / 3D Plotting

Matplotlib VTK integration | Matplotlib: mplot3d

https://scipy-cookbook.readthedocs.io/

In your homework, you'll be using a few SciPy submodules to perform several analysis-relevant tasks, including using the **signal** package to create and use a filter.

fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions

In this case, you'll be importing it as though it was an independent module! E.g.

```
from scipy import stats
```

Aaannnnd that was fast! We’re about to move on to pandas!

This is a transition slide.

Hooray, everyone! We made it to **pandas**!

New logo ->

pandas

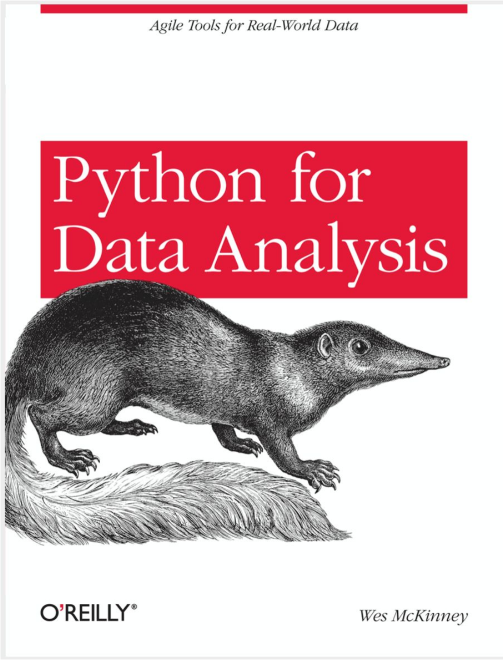
Old logo ft. bear ->

Pandas

Red panda ->

Hooray, everyone! We made it to **pandas**!

pandas (name comes from “panel data”, an econ term) was developed by BDFL (benevolent dictator for life) Wes McKinney. He wrote the truly excellent book “Python for Data Analysis”, which we **highly** recommend you find a copy of



pandas objects: DataFrames and Series

You've already gained some exposure to pandas in HW2. If you worked through some of the problems, you may have noticed that Jacob initially uploaded his data into a new type of object called a **DataFrame**.

The **DataFrame**, along with the **Series**, are the two core objects that you'll encounter in pandas. Let's focus on Series first. *Series are the building block of pandas*. They're basically just 1D arrays that are **labeled and indexed**, and can hold 0 or more values of *any single data type.* If we wanted to be crass, we could just go ahead and refer to them as a column in an excel spreadsheet.

The primary benefit to a series is the fact that it is indexed - otherwise, they work very similarly to numpy arrays! All of the functions and methods that you've learned from arrays are applicable here.

When should you use a series? Well, one thing to keep in mind is that numpy is designed to be a fast way of handling large, multidimensional arrays for scientific computing (and plays very nicely SciPy tools). So Series really come into play when you're working with 1D data that benefits from an **index**. Series also play a huge role in **DataFrames,** which we'll talk about in a second.

pandas objects: creating a Series

First, let's get a quick handle on using pandas! In your notebook, go ahead and import pandas as pd.

How do we make a series? There are many ways!

Below, find the list "names_list", consisting of five random names.

```
In [208]: names_list = ['Aragorn', 'Legolas', 'Gimli', 'Galadriel', 'Eowyn']
```

Use panda's Series() method, with names_list as input, to create a series assigned to variable "names_series"
Print output.

pandas objects: creating a Series from a list (example)

First, let's get a quick handle on using pandas! In your notebook, go ahead and import pandas as pd.

How do we make a series? There are many ways!

Below, find the list "names_list", consisting of five random names.

```
In [208]: names_list = ['Aragorn', 'Legolas', 'Gimli', 'Galadriel', 'Eowyn']
```

Use panda's Series() method, with names_list as input, to create a series assigned to variable "names_series" Print output.

```
In [211]: names_series = pd.Series(names_list)
          print(names_series)

0      Aragorn
1      Legolas
2        Gimli
3    Galadriel
4        Eowyn
dtype: object
```

pandas objects: creating a Series from a list (example)

First, let's get a quick handle on using pandas! In your notebook, go ahead and import pandas as pd.

How do we make a series? There are many ways!

Below, find the list "names_list", consisting of five random names.

```
In [208]: names_list = ['Aragorn', 'Legolas', 'Gimli', 'Galadriel', 'Eowyn']
```

Use panda's Series() method, with names_list as input, to create a series assigned to variable "names_series" Print output.

```
In [211]: names_series = pd.Series(names_list)
          print(names_series)

0      Aragorn
1      Legolas
2        Gimli
3    Galadriel
4        Eowyn
dtype: object
```

Obviously, we could also pass values to Series directly, a la `pd.Series(['Aragorn','Legolas',... etc.])` – you can also pass numpy arrays **or even dicts** into the Series function!

pandas objects: creating a Series from a list (example)

```
In [211]: names_series = pd.Series(names_list)
          print(names_series)

0      Aragorn
1      Legolas
2        Gimli
3    Galadriel
4        Eowyn
dtype: object
```

Looking at our printed series, it's pretty clear that this no longer looks like a list. There are values all along the left side!! This is the series **index**. Because we didn't specify anything in particular, pandas automatically created an index for us beginning with 0 and continuing through the length of the list we passed it; however, we have control over the index! We can specify values *within* the input we provide the Series() function, like this:

```
names_series = pd.Series(names_list, index = [5,6,7,8,9])
```

OR If we create a series from a **dictionary**, the index will automatically contain key values from the dict.

pandas objects: DataFrames

Now, to our real pandas bread and butter - the **DataFrame**

For all intents and purposes, you can think of a DataFrame as a table containing an array of entries, each of which corresponds to a *row (index)* and a *column*. Once again, if we wanted to be crass, we could use an excel spreadsheet with column headings as a sort of rudimentary example of how to think of DataFrames (but DataFrames have **so. much. more. flexible. functionality.**)

Why would we use a DataFrame instead of a numpy array? This is actually a pretty difficult question when it's framed (lol) like this, but some short answers are:

1.) DataFrames can contain numpy arrays (technically the columns are stored as numpy arrays, and pandas functions are actually wrapper functions around the numpy functions that you've seen)
2.) DataFrames allow for extremely intuitive, easy organization and grouping of data
3.) Do keep in mind that numpy arrays are built to handle large, multidimensional arrays quickly. DataFrames will be slower than arrays in many functions! It's all about choosing the correct DataType for what you're trying to do.

pandas objects: DataFrames

So how do we make a DataFrame? There are actually myriad ways!! This is **one of many** - and not the most efficient!

Create a dictionary lotr_data, with the key 'Beings' assigned a list containing the strings 'human','elf','dwarf','elf', and 'human', and the key 'Age', assigned a list containing the values 87, 2931, 139, 7000, and 24

pandas objects: DataFrames

So how do we make a DataFrame? There are actually myriad ways!! This is **one of many** - and not the most efficient!

Create a dictionary lotr_data, with the key 'Beings'assigned a list containing the strings 'human','elf','dwarf','elf', and 'human', and the key 'Age', assigned a list containing the values 87, 2931, 139, 7000, and 24

```
In [227]: lotr_data = {
              'Beings' : ['human','elf','dwarf','elf','human'],
              'Age' : [87, 2931, 139, 7000, 24]
          }
```

Now, create a dataframe lotr_df by passing lotr_data into the pandas function DataFrame()

pandas objects: DataFrames

So how do we make a DataFrame? There are actually myriad ways!! This is **one of many** - and not the most efficient!

Create a dictionary lotr_data, with the key 'Beings'assigned a list containing the strings 'human','elf','dwarf','elf', and 'human', and the key 'Age', assigned a list containing the values 87, 2931, 139, 7000, and 24

```
In [227]: lotr_data = {
              'Beings' : ['human','elf','dwarf','elf','human'],
              'Age' : [87, 2931, 139, 7000, 24]
          }
```

Now, create a dataframe lotr_df by passing lotr_data into the pandas function DataFrame()

```
In [228]: lotr_df = pd.DataFrame(lotr_data)
          print(lotr_df)

  Beings   Age
0  human    87
1    elf  2931
2  dwarf   139
3    elf  7000
4  human    24
```

pandas objects: DataFrames

We've been using Python's built-in print function throughout this class to look at data - but DataFrames are unique in that they are actually nicer to look at as output! Try using the .head() method associated with DataFrames - head() will normally return the first five rows of a DataFrame, but can take any number as input. In this case, just show the first 3 rows:

pandas objects: DataFrames

We've been using Python's built-in print function throughout this class to look at data - but DataFrames are unique in that they are actually nicer to look at as output! Try using the .head() method associated with DataFrames - head() will normally return the first five rows of a DataFrame, but can take any number as input. In this case, just show the first 3 rows:

```
In [231]: lotr_df.head(3)
```

Out[231]:

	Beings	Age
0	human	87
1	elf	2931
2	dwarf	139

Let's say we want to view just the 'Beings' column of our DataFrame - this can be accomplished via the simple command `DataFrame['COLUMN NAME']`. Try looking at just Beings below.

pandas objects: DataFrames

We've been using Python's built-in print function throughout this class to look at data - but DataFrames are unique in that they are actually nicer to look at as output! Try using the .head() method associated with DataFrames - head() will normally return the first five rows of a DataFrame, but can take any number as input. In this case, just show the first 3 rows:

```
In [231]: lotr_df.head(3)
Out[231]:
```

	Beings	Age
0	human	87
1	elf	2931
2	dwarf	139

Let's say we want to view just the 'Beings' column of our DataFrame - this can be accomplished via the simple command `DataFrame['COLUMN NAME']`. Try looking at just Beings below.

```
In [232]: lotr_df['Beings']
Out[232]: 0    human
          1      elf
          2    dwarf
          3      elf
          4    human
          Name: Beings, dtype: object
```

pandas objects: DataFrames

Interestingly, the columns of a DataFrame are actually also **attributes**, meaning they can be accessed using DataFrame.COLUMNS notation. Try pulling out the Age column this way below.

pandas objects: DataFrames

Interestingly, the columns of a DataFrame are actually also **attributes**, meaning they can be accessed using DataFrame.COLUMNS notation. Try pulling out the Age column this way below.

```
In [240]:  1  lotr_df.Age

Out[240]: 0      87
          1    2931
          2     139
          3    7000
          4      24
```

pandas objects: DataFrames

Egads! It looks like we've forgotten to insert our character names from earlier into our DataFrame! Thankfully, using the same indexing that we see above, this is quite easy to accomplish in a DataFrame - simply create a *new* column with the command DataFrame['NEW COLUMN NAME'] = _____ Below, insert your names_series Series into lotr_df

pandas objects: DataFrames

Egads! It looks like we've forgotten to insert our character names from earlier into our DataFrame! Thankfully, using the same indexing that we see above, this is quite easy to accomplish in a DataFrame - simply create a *new* column with the command DataFrame['NEW COLUMN NAME'] = ____ Below, insert your names_series Series into lotr_df

In [235]:
```
lotr_df['Character Name'] = names_series
lotr_df
```

Out[235]:

	Beings	Age	Character Name
0	human	87	Aragorn
1	elf	2931	Legolas
2	dwarf	139	Gimli
3	elf	7000	Galadriel
4	human	24	Eowyn

pandas objects: DataFrames

This is all well and good, but let's say we get our hands on a *slightly* more complete dataset, and want to import it. pandas has built in functions to read/import **many** different data types, including (but not limited to) numpy arrays, .xlsx, and .csv files.

Use the pandas read_csv function, which will take a csv file at a given directory and import it into a DataFrame, to import the lotr_char_age.csv file that you should have put into the same folder as this notebook at the beginning of class. Re-assign your lotr_df DataFrame to the output of this function.

pandas objects: DataFrames

This is all well and good, but let's say we get our hands on a *slightly* more complete dataset, and want to import it. pandas has built in functions to read/import **many** different data types, including (but not limited to) numpy arrays, .xlsx, and .csv files.

Use the pandas read_csv function, which will take a csv file at a given directory and import it into a DataFrame, to import the lotr_char_age.csv file that you should have put into the same folder as this notebook at the beginning of class. Re-assign your lotr_df DataFrame to the output of this function.

In [237]:

```
lotr_df = pd.read_csv('lotr_char_age.csv')
lotr_df
```

Out[237]:

	Character Name	Beings	Age
0	Aragorn	human	87
1	Legolas	elf	2931
2	Gimli	dwarf	139
3	Galadriel	elf	7000
4	Eowyn	human	24
5	Frodo	hobbit	51
6	Arwen	elf	2778
7	Boromir	human	41

pandas objects: DataFrames

What a lovely DataFrame we've created! It doesn't have a *lot* more information, but it should be enough for us to quickly go over how to pull the data your want out of your DataFrame.

We've already learned that it's easy to pull out all of the contents of a column - but this technique can also be used with **conditionals**. First, try seeing what happens if you run a conditional asking for the Age column where age < 1000?

pandas objects: DataFrames

What a lovely DataFrame we've created! It doesn't have a *lot* more information, but it should be enough for us to quickly go over how to pull the data your want out of your DataFrame.

We've already learned that it's easy to pull out all of the contents of a column - but this technique can also be used with **conditionals**. First, try seeing what happens if you run a conditional asking for the Age column where age < 1000?

In [242]:
```
lotr_df['Age'] < 1000
```

Out[242]:
```
0      True
1     False
2      True
3     False
4      True
5      True
6     False
7      True
8      True
9      True
10     True
11     True
12     True
13    False
14     True
15     True
16    False
Name: Age, dtype: bool
```

pandas objects: DataFrames

We've seen booleans before, and covered how to use them in numpy arrays earlier in this lecture! Based on what we learned then, and knowing how to pull column data from DataFrames, can you think of a method to return all values of lotr_df that are associated with an age <1000?

pandas objects: DataFrames

We've seen booleans before, and covered how to use them in numpy arrays earlier in this lecture! Based on what we learned then, and knowing how to pull column data from DataFrames, can you think of a method to return all values of lotr_df that are associated with an age <1000?

In [247]:
```
lotr_df[lotr_df['Age'] < 1000]
```
Out[247]:

	Character Name	Beings	Age
0	Aragorn	human	87
2	Gimli	dwarf	139
4	Eowyn	human	24
5	Frodo	hobbit	51
7	Boromir	human	41
8	Merry	hobbit	37
9	Pippin	hobbit	29
10	Sam	hobbit	36
11	Eomer	human	28
12	Theoden	human	71
14	Bilbo	hobbit	129

pandas objects: DataFrames

Great!

We can even slice columns once we've pulled them from our DataFrame. Let's say we only want to see the first 3 entries from what we pulled out in the last window. We'd use the notation DataFrame['COLUMN NAME'][INDEX OR SLICE]. Give it a try now!

pandas objects: DataFrames

Great!

We can even slice columns once we've pulled them from our DataFrame. Let's say we only want to see the first 3 entries from what we pulled out in the last window. We'd use the notation DataFrame['COLUMN NAME'][INDEX OR SLICE]. Give it a try now!

```
In [248]:  1  lotr_df[lotr_df['Age'] < 1000][:3]
Out[248]:
```

	Character Name	Beings	Age
0	Aragorn	human	87
2	Gimli	dwarf	139
4	Eowyn	human	24

pandas objects: DataFrames

Everything we've done so far has been focused on pulling data out from DataFrame based on its column. But, when we introduced DataFrames, we also made a point that their indexing is valuable! How can we pull out rows based on index?

This is were DataFrames differ significantly from arrays - in order to access data in particular rows, DataFrames require users to use .iloc[] and .loc[] methods.

iloc, or index-based selection, treats your DataFrame like a giant matrix, and pulls out data based on its location (corresponding to the value you provided as input). loc, on the other hand, takes into account the *labels of the data* in a DataFrame. In the case of lotr_df, index values are the same as their numerical location; however, you might see how we need to use loc if we instead want to specify 'Beings' == 'hobbit'.

Let's try iloc first - works quite similarly to how you've gotten used to indexing/slicing numpy arrays, except that it takes in inputs in the order of **rows, columns** - e.g., DataFrame.iloc[2:4, 0] will return values from the 0th column, rows 2 to 4.

Below, pull out the 2nd column, rows 1, 3, 5, and 7 of lotr_df

pandas objects: DataFrames

Below, pull out the 2nd column, rows 1, 3, 5, and 7 of lotr_df

In [252]:
```
lotr_df.iloc[[1,3,5,7],2]
lotr_df.iloc[1:8:2,2]
```

Out[252]:
```
1    2931
3    7000
5      51
7      41
Name: Age, dtype: int64
```

pandas objects: DataFrames

Great! Let's move on to loc - in this case, we can start to use labels and conditionals! For example, what happens if we specify that we want only rows of lotr_df where `'Beings' == 'elf'` **and** `age < 4000` ? We can use the "&" operator!!

In [255]:

```
#This is an example - feel free to run the code to see output
lotr_df.loc[(lotr_df['Beings']=='elf') & (lotr_df['Age'] < 4000)]
```

Out[255]:

	Character Name	Beings	Age
1	Legolas	elf	2931
6	Arwen	elf	2778

Your turn! Pull out all hobbits under the age of 50 from lotr_df **and** assign these values to a new DataFrame called young_hobbits

pandas objects: DataFrames

Great! Let's move on to loc - in this case, we can start to use labels and conditionals! For example, what happens if we specify that we want only rows of lotr_df where `'Beings' == 'elf'` **and** `age < 4000` ? We can use the "&" operator!!

In [255]:

```
#This is an example - feel free to run the code to see output
lotr_df.loc[(lotr_df['Beings']=='elf') & (lotr_df['Age'] < 4000)]
```

Out[255]:

	Character Name	Beings	Age
1	Legolas	elf	2931
6	Arwen	elf	2778

Your turn! Pull out all hobbits under the age of 50 from lotr_df **and** assign these values to a new DataFrame called young_hobbits

In [258]:

```
young_hobbits = lotr_df.loc[(lotr_df['Beings']=='hobbit') & (lotr_df['Age'] < 50)]
young_hobbits
```

Out[258]:

	Character Name	Beings	Age
8	Merry	hobbit	37
9	Pippin	hobbit	29
10	Sam	hobbit	36

pandas objects: DataFrames

As you can see, if we had a **very** large data set, this would be an invaluable tool for segmenting our data. Next, we'll be covering one or two more powerful tools associated with DataFrames before moving onto a brief overview of plotting with seaborn and wrapping up!

But first, oh no! The Tolkien fan who made our dataset didn't know that it can be hard to work with spaces in code - to make things easier for future users, let's see if we can rename our 'Character Name' column to 'Character_Name'. This can easily be accomplished using the rename method, which takes in dict-like input that can be specified to **either** columns or the index. In this case, the format would look like:

```
DataFrame.rename( {
    ‘EXISTING COLUMN': 'NEW NAME', etc.
        }, axis = ‘columns’)
```

pandas objects: DataFrames

Rename our 'Character Name' column to 'Character_Name':

pandas objects: DataFrames

Rename our 'Character Name' column to 'Character_Name':

In [268]:
```
lotr_df = lotr_df.rename({'Character Name':'Character_Name'},axis='columns')
lotr_df.head()
```

Out[268]:

	Character_Name	Beings	Age
0	Aragorn	human	87
1	Legolas	elf	2931
2	Gimli	dwarf	139
3	Galadriel	elf	7000
4	Eowyn	human	24

pandas objects: DataFrames - groupby()

Now that we've got our column names in order, let's take a look at one of the most powerful tools we have with DataFrames : the groupby() function.

Simply put, `groupby(['CATEGORY'])` is a method of grouping categories, and applying a function to each group. Let's just test it out to see what's going on! Create a DataFrame called "being_stats" that is lotr_df grouped by beings, and run the "count()" method on it:

In [282]:
```
being_stats = lotr_df.groupby(['Beings'])
being_stats.count()
```

Out[282]:

Beings	Character_Name	Age
dwarf	1	1
dwarf	1	1
elf	5	5
hobbit	5	5
human	5	5

What the above output is telling us is that each label under "Beings" is a unique value in the lotr_df['Beings'] column. For elves, there are 5 entered character names, and 5 entered ages.

But how strange! It looks like we're seeing...two dwarfs (dwarves, I know...)?? Can anybody think of a reason for this?

pandas objects: DataFrames - groupby()

We can take a closer look by using the unique() function - run `lotr_df['Beings'].unique()` . What do you see?

pandas objects: DataFrames - unique()

We can take a closer look by using the unique() function - run `lotr_df['Beings'].unique()` . What do you see?

```
In [278]:  1 lotr_df['Beings'].unique()

Out[278]: array(['human', 'elf', 'dwarf', 'hobbit', 'dwarf '], dtype=object)
```

pandas objects: DataFrames - replace()

This is all part of cleaning data, folks! People make mistakes!

In this case, we want to go in and replace the error, 'dwarf ', with 'dwarf'. To do this, let's use the replace() function associated with DataFrames. replace() can take in dict-like formats just like rename() - it will look like this:

```
DataFrame.replace({'COLUMN NAME' : VALUE IN COLUMN }, VALUE TO BE INSERTED )
```

Try replacing ‘dwarf ‘ with the correct value - can you check the ‘dwarf’ entries by using .loc() ?

pandas objects: DataFrames - replace()

This is all part of cleaning data, folks! People make mistakes!

In this case, we want to go in and replace the error, 'dwarf ', with 'dwarf'. To do this, let's use the replace() function associated with DataFrames. replace() can take in dict-like formats just like rename() - it will look like this:

```
DataFrame.replace({'COLUMN NAME' : VALUE IN COLUMN }, VALUE TO BE INSERTED )
```

Try replacing ‘dwarf ‘ with the correct value - can you check the ‘dwarf’ entries by using .loc() ?

```
In [285]: lotr_df = lotr_df.replace({'Beings':'dwarf '},'dwarf')
```

```
In [289]: lotr_df.loc[lotr_df['Beings'] == 'dwarf']
```

Out[289]:

	Character_Name	Beings	Age
2	Gimli	dwarf	139
15	Gloin	dwarf	253

pandas objects: DataFrames - back to groupby()

Great work!!! Finally, let's return to groupby(). Now that we know that we've corrected the error in the DataFrame, see if you can use groupby() and the .mean() functions to pull out the mean age for each type of being.

pandas objects: DataFrames - back to groupby()

Great work!!! Finally, let's return to groupby(). Now that we know that we've corrected the error in the DataFrame, see if you can use groupby() and the .mean() functions to pull out the mean age for each type of being.

```
In [290]: lotr_df.groupby(['Beings'])['Age'].mean()
```

```
Out[290]: Beings
dwarf        196.0
elf         5145.4
hobbit        56.4
human         50.2
Name: Age, dtype: float64
```

pandas objects: DataFrames - back to groupby()

Great work!!! Finally, let's return to groupby(). Now that we know that we've corrected the error in the DataFrame, see if you can use groupby() and the .mean() functions to pull out the mean age for each type of being.

```
In [290]: lotr_df.groupby(['Beings'])['Age'].mean()
```

```
Out[290]: Beings
dwarf       196.0
elf        5145.4
hobbit       56.4
human        50.2
Name: Age, dtype: float64
```

I feel like I give this warning for everything I do - but what we've done here is give you a **very** brief rundown of all the functionality you can achieve using pandas DataFrames… but we've only covered a fraction to give you an intuition!! I *implore* you to go online, do some more exercises, and check out the documentation!

One last, quick thing: seaborn!

This one is fast and fun!!

Now that you're a pro at working with DataFrames, it's time to introduce you to the beauty that is plotting with seaborn.

In Jacob's lecture, you learned about the powerful, versatile **matplotlib**. Seaborn is *built on top of matplotlib,* but creates absolutely beautiful plots from DataFrames with minimal effort to the user.

Check out the gallery: https://seaborn.pydata.org/examples/index.html

All you need is a well-formatted DataFrame!!

One last, quick thing: seaborn!

Just like SciPy, what we're going to do with seaborn is turn you loose with the documentation - I think you'll find it to be beautifully intuitive and easy to follow!

But for kicks, why don't we try *one little* example?

Go ahead and run the "boxplot" code

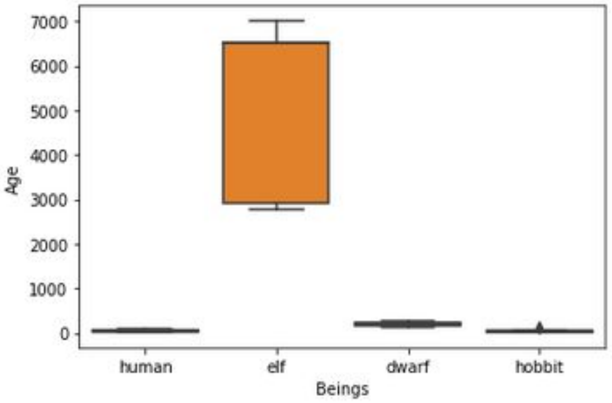
One last, quick thing: seaborn!

Just like SciPy, what we’re going to do with seaborn is turn you loose with the documentation - I think you’ll find it to be beautifully intuitive and easy to follow!

But for kicks, why don’t we try *one little* example?

Go ahead and run the “boxplot” code

```
In [305]:
# Simple boxplot code!!
#go ahead and run this window, just to see what happens
sns.boxplot(x='Beings',y='Age',data=lotr_df)

Out[305]: <matplotlib.axes._subplots.AxesSubplot at 0x1a19ae7c10>
```



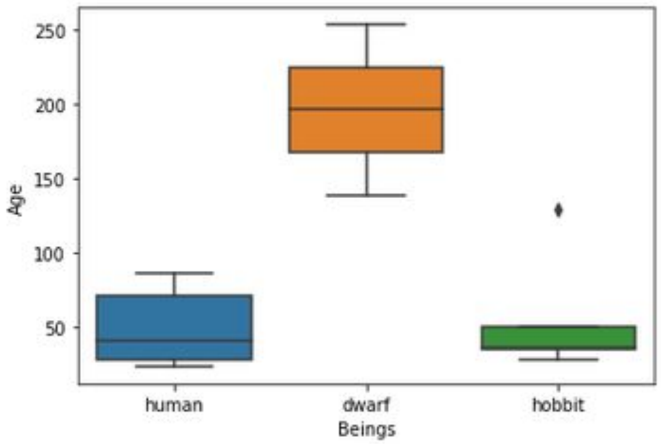
One last, quick thing: seaborn!

I promised beauty...but there are a lot of things happening in that plot that aren't great. But based on how easy it is to change things around in seaborn...in the cell below, change data to equal lotr_df **without** elves (don't forget loc combined with conditional statements!)

One last, quick thing: seaborn!

I promised beauty...but there are a lot of things happening in that plot that aren't great. But based on how easy it is to change things around in seaborn...in the cell below, change data to equal lotr_df **without** elves (don't forget loc combined with conditional statements!)

```
In [306]:  1  sns.boxplot(x='Beings',y='Age',data=lotr_df.loc[lotr_df['Beings']!='elf'])
Out[306]: <matplotlib.axes._subplots.AxesSubplot at 0x1a19c117d0>
```

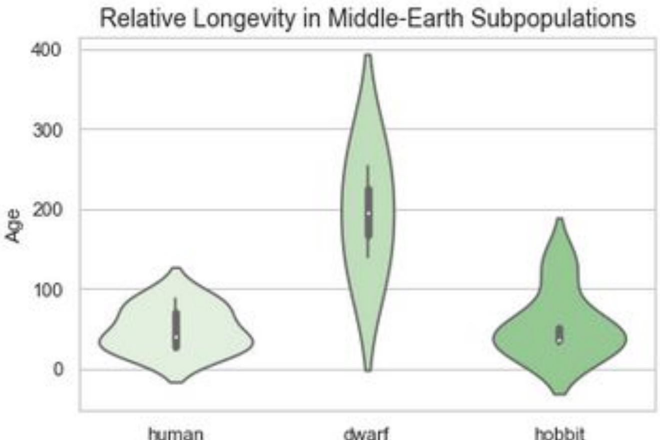


One last, quick thing: seaborn!

Okay, looking better! A few more improvements…
let's use seaborn's set() function to set the palette to something calming, like 'Greens'...
and the background to something weird, like 'whitegrid'...
And let's go ahead and use violinplot, not boxplot, just to be fancy...
And while we're at it, let's set our plot to an object "g" so that we can title it easily.

In [319]:
```
sns.set(palette = 'Greens',style='whitegrid')

g = sns.violinplot(x='Beings',y='Age',data=lotr_df.loc[lotr_df['Beings']!='elf'])

g.set_title('Relative Longevity in Middle-Earth Subpopulations',fontsize=14)

g.set_xlabel('Subpopulation')# <- decided 'Beings' looked silly, so replaced with 'Subpopulation'

```

Out[319]:
```
Text(0.5, 0, 'Subpopulation')
```



That’s all, folks!

The fun thing about seaborn is that it’s primarily based around the “groupby” function we just learned about...it’s just that that’s all under the hood! And I know it’s fun :) So don’t worry! There will be a very short seaborn problem on HW3!