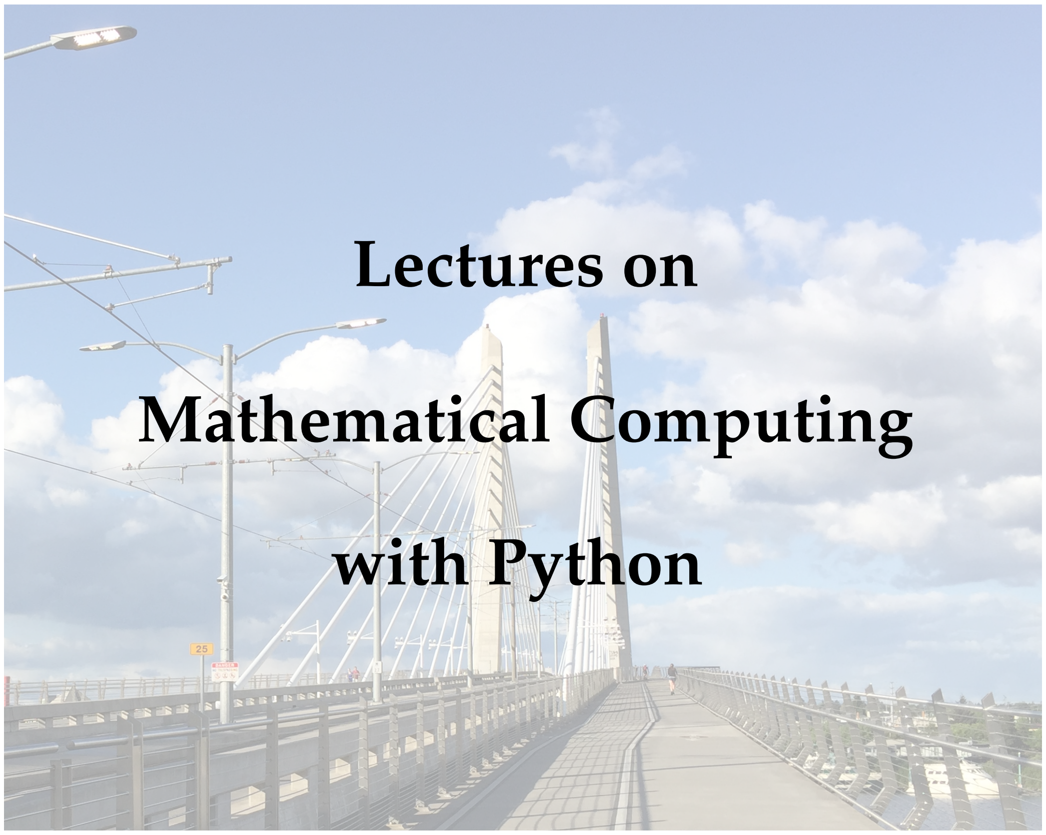


Lectures on Mathematical Computing with Python

Jay Gopalakrishnan

Portland State University

© 2020 Jay Gopalakrishnan

This work is licensed under a Creative Commons Attribution–ShareAlike 4.0 International License.

You are free to:

- Share: copy and redistribute the material in any medium or format, and
- Adapt: remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- Attribution: You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- ShareAlike: If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- No additional restrictions: You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Digital Object Identifier (DOI):

10.15760/pdxopen-28

(https://doi.org/10.15760/pdxopen-28)

Recommended citation:

Gopalakrishnan, J., *Lectures on Mathematical Computing with Python*, PDXOpen: Open Educational Resource 28, Portland State University Library, DOI: 10.15760/pdxopen-28, July 2020.

This corrected version is dated August 4, 2020.

Preface

These lectures were prepared for a class of (mostly) second year mathematics and statistics undergraduate students at Portland State University during Spring 2020. The term was unlike any other. The onslaught of COVID-19 moved the course meetings online, an emergency transition that few of us were prepared for. Many lectures reflect our preoccupations with the damage inflicted by the virus. I have not attempted to edit these out since I felt that a utilitarian course on computing need not be divested from the real world.

These materials offer class activities for studying basics of mathematical computing using the python programming language, with glimpses into modern topics in scientific computation and data science. The lectures attempt to illustrate computational thinking by examples. They do not attempt to introduce programming from the ground up, although students, by necessity, will learn programming skills from external materials. In my experience, students are able and eager to learn programming by themselves using the abundant free online resources that introduce python programming. In particular, my students and I found the two (free and online) books of Jake VanderPlas invaluable. Many sections of these two books, hyperlinked throughout these lectures, were assigned as required preparatory reading materials during the course (see List of Preparatory Materials).

Materials usually covered in a first undergraduate linear algebra course and in a one-variable differential calculus course form mathematical prerequisites for some lectures. Concepts like convergence may not be covered rigorously in such prerequisites, but I have not shied away from talking about them: I feel it is entirely appropriate that a first encounter with such concepts is via computation.

Each lecture has a date of preparation. It may help the reader understand the context in relation to current events and news headlines. The timestamp also serves as an indicator of the state of the modules in the ever-changing python ecosystem of modules for scientific computation. The specific version numbers of the modules used are listed overleaf. The codes may need tinkering with to ensure compatibility with future versions. The materials are best viewed as offering a starting point for your own adaptation.

If you are an instructor declaring these materials as a resource in your course syllabus, I would be happy to provide any needed solutions to exercises or datafiles. If you find errors please alert me. If you wish to contribute by updating or adding materials, please fork the public GitHub Repository where these materials reside and send me a pull request.

Jay Gopalakrishnan

(gjay@pdx.edu)

Software Requirements:

- Python >= 3.7
- Jupyter >= 1

Main modules used:

- cartopy==0.18.0b2.dev48+
- geopandas==0.7.0
- gitpython==3.1.0
- matplotlib==3.2.1
- numpy==1.18.2
- pandas==1.0.4
- scipy==1.4.1
- scikit-learn==0.23.1
- seaborn==0.10.0
- spacy==2.2.4

Other (optional) facilities used include line_profiler, memory_profiler, numexpr, pandas-datareader, and primesieve.

Table of Contents

List of Preparatory Materials for Each Activity

The activities in the table of contents are enumerated again below in a linear ordering with hyperlinks to external online preparatory materials for each.

Required Preparation	Activity
Watch the first few of the 44 Microsoft videos on python. Watch a 2014 video by SIAM: What is data science? Browse basic language facilities either from the official Python tutorial or from [JV-W].	01 Overview of tools
Read about the iPython shell facilities from the first chapter of [JV-H].	02 Interacting with python
Browse Git Handbook	03 Working with git
Using the Python tutorial or [JV-W] work with `if`, `while`, `for`,`range`, `print`, lists `[]`, tuples `()`, and list comprehension.	04 Conversion table Exercise: Power sum
Using Python tutorial or [JV-W], learn about functions, `def`, and `lambda`.	05 Approximating derivatives Project: Bisection
Using Python tutorial or [JV-W], learn about dictionaries `{}`, strings and file operations `open`, `readlines`	06 Genome of SARS-CoV-2 virus Project: Rise of CO_2 in the atmosphere
Learn about pytest, generator expressions, `yield`, line and cell magics	07 Fibonacci primes
Learn numpy basics from [JV-H], ufuncs, broadcasting indexing, masking	08 Numpy blitz Exercise: Argument passing Exercise: Piecewise functions

Required Preparation	Activity
Learn sorting, partitioning [JV-H], and quick ways to make matrices from [numpy.org].	Exercise: Row Swap Exercise: Averaging Matrix Exercise: Differentiation Matrix
Learn how to make simple plots using `matplotlib`. Read about aggregation and masking from [JV-H]	Exercise: Graphing functions Exercise: Pairwise differences Exercise: Hausdorff distance Exercise: k-nearest neighbors
Get an overview of scipy facilities. Online scipy lecture notes are very helpful. Familiarize yourselves with scipy's `sparse` and `integrate` modules.	09 SEIR model of infectious diseases Exercise: Predator-prey model
Learn numpy facilities for matrix factorizations, eigenvalues etc.	10 Singular value decomposition Exercise: Column space Exercise: Null space
Introduce yourselves to the data analysis module pandas.	Exercise: Pandas from dictionaries Exercise: Iris flower dataset
Reinforce your pandas skills.	11 Bikes on Tilikum Crossing Exercise: Stock prices Exercise: Passengers on the Titanic Project: Growth of COVID-19 cases in the west coast
Familiarize yourselves with geopandas, cartopy, and matplotlib.animation.	12 Visualizing geospatial data Exercise: Animate functions Project: World map of COVID-19 cases
Review scipy.sparse. Introduce yourselves to NetworkX.	13 Gambler's Ruin Exercise: Insurance Company Exercise: Probabilities on small graphs Exercise: Ehrenfest thought experiment

Required Preparation	Activity
	Project: Neighbor's color
Be acquainted with scipy.sparse's matrix format, specifically COO and CSR formats.	Exercise: Power method for large graphs Exercise: Google's toy graph
Read the good introduction to machine learning from [JV-H]	15 Supervised learning by regression Exercise: Atmospheric carbon dioxide
Read about unsupervised machine learning, focusing specifically on PCA. Also review the prior lecture on SVD.	16 Unsupervised learning by PCA Exercise: Ovarian cancer data Exercise: Eigenfaces
Learn about text features in machine learning from [JV-H].	17 Latent semantic analysis Exercise: Word vectors

I Overview of some tools

March 31, 2020

This lecture is an introductory overview to give you a sense of the broad utility of a few python tools you will encounter in later lectures. Each lecture or class activity is guided by a Jupyter Notebook (like this document), which combines executable code, mathematical formulae, and text notes. This overview notebook also serves to check and verify that you have a working installation of some of the python modules we will need to use later. We shall delve into basic programming using python (after this overview and a few further start-up notes) starting from a later lecture.

The ability to program, analyze and compute with data are life skills. They are useful well beyond your mathematics curriculum. To illustrate this claim, let us begin by considering the most pressing current issue in our minds as we begin these lectures: the progression of COVID-19 disease worldwide. The skills you will learn in depth later can be applied to understand many types of data, including the data on COVID-19 disease progression. In this overview, we shall use a few python tools to quickly obtain and visualize data on COVID-19 disease worldwide. The live data on COVID-19 (which is changing in as yet unknown ways) will also be used in several later activities.

Specifically, this notebook contains all the code needed to perform these tasks:

- download today's data on COVID-19 from a cloud repository,
- make a data frame object out of the data,
- use a geospatial module to put the data on a world map,
- download county maps from US Census Bureau, and
- visualize the COVID-19 data restricted to Oregon.

The material here is intended just to give you an overview of the various tools we will learn in depth later. There is no expectation that you can immediately digest the code here. The goal of this overview is merely to whet your appetite and motivate you to allocate time to learn the materials yet to come.

I.1 The modules you need

These are the python modules we shall use below.

- `matplotlib` (for various plotting & visualization tools in python)
- `descartes` (for specialized visualization of maps using matplotlib)
- `gitpython` (to work in python with Git repositories)
- `pandas` (to make data frame structures out of raw data)
- `geopandas` (for analysis of geospatial data)
- `urllib` (for fetching resources at an internet url)

Please install these modules if you do not have them already. (If you do not have these installed, attempting to run the next cell will give you an error.)

[1]:
```
import pandas as pd
import os
from git import Repo
import matplotlib.pyplot as plt
import geopandas as gpd
import urllib
import shutil
%matplotlib inline
```

I.2 Get the data

The Johns Hopkins University Center for Systems Science and Engineering has curated data on COVID-19 from multiple sources and provided it online as a "git" repository in a cloud server at https://github.com/CSSEGISandData/COVID-19. (We shall learn a bit more about git in a later lecture.) These days, as the disease progresses, new data is being pushed into this repository every day.

Git repositories in the cloud server can be *cloned* to get an identical local copy on our computers. Let us begin by cloning a copy of the Johns Hopkins COVID-19 data repository into a location in your computer. Please specify this location in your computer in the variable called `covidfolder` below. Once you have cloned the repository, the next time you run the same line of code, it does not clone it again. Instead, it only pulls updates from the cloud to sync your local copy with the remote original.

[2]:
```
# your local folder into which you want to download the covid data

covidfolder = '../../data_external/covid19'
```

Remember this location where you have stored the COVID-19 data. You will need to return to it when you use the data during activities in later days, including assignment projects.

[3]:
```
if os.path.isdir(covidfolder):    # if repo exists, pull newest data
    repo = Repo(covidfolder)
    repo.remotes.origin.pull()
else:                             # otherwise, clone from remote
    repo = Repo.clone_from('https://github.com/CSSEGISandData/COVID-19.
 ↪git',
                           covidfolder)
datadir = repo.working_dir + '/csse_covid_19_data/
 ↪csse_covid_19_daily_reports'
```

The folder `datadir` contains many files (all of which can be listed here using the command `os.listdir(datadir)` if needed). The filenames begin with a date like `03-27-2020` and ends in `.csv`. The ending suffix `csv` stands for "comma separated values", a common simple format for storing uncompressed data.

I.3 Examine the data for a specific date

The python module `pandas`, the workhorse for all data science tasks in python, can make a `DataFrame` object out of each such `.csv` files. You will learn more about pandas later in the course. For now, let us pick a recent date, say March 27, 2020, and examine the COVID-19 data for that date.

```
[4]: c = pd.read_csv(datadir+'/03-27-2020.csv')
```

The `DataFrame` object `c` has over 3000 rows. An examination of the first five rows already tells us a lot about the data layout:

```
[5]: c.head()
```

```
[5]:       FIPS     Admin2  Province_State  Country_Region          Last_Update ␣
   ↪\
0  45001.0  Abbeville  South Carolina              US  2020-03-27 22:14:55
1  22001.0     Acadia       Louisiana              US  2020-03-27 22:14:55
2  51001.0   Accomack        Virginia              US  2020-03-27 22:14:55
3  16001.0        Ada           Idaho              US  2020-03-27 22:14:55
4  19001.0      Adair            Iowa              US  2020-03-27 22:14:55

         Lat       Long_  Confirmed  Deaths  Recovered  Active  \
0  34.223334  -82.461707          4       0          0       0
1  30.295065  -92.414197          8       1          0       0
2  37.767072  -75.632346          2       0          0       0
3  43.452658 -116.241552         54       0          0       0
4  41.330756  -94.471059          1       0          0       0

                  Combined_Key
0  Abbeville, South Carolina, US
1          Acadia, Louisiana, US
2         Accomack, Virginia, US
3                Ada, Idaho, US
4               Adair, Iowa, US
```

Note that depending on how the output is rendered where you are reading this, the later columns may be line-wrapped or may be visible only after scrolling to the edges. This object `c`, whose `head` part is printed above, looks like a structured array. There are features corresponding to locations, as specified in latitude `Lat` and longitude `Long_`. The columns `Confirmed`, `Deaths`, and `Recovered` represents the number of confirmed cases, deaths, and recovered cases due to COVID-19 at a corresponding location.

I.4 Put the data on a map

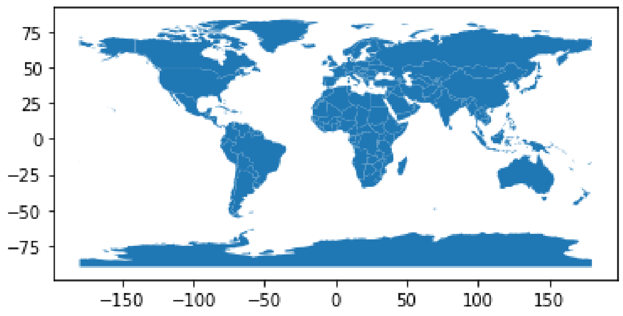
Data like that in `c` contains geospatial information. One way to visualize geospatial data is to indicate the quantity of interest on a map. We shall visualize the data in the `Confirmed` column by positioning a marker at its geographical location and make the marker size correspond to the number of confirmed cases at that position. The module `geopandas` (gpd) is well-suited for visualizing geospatial data. It is built on top of the pandas library. So it is easy to convert our pandas object c to a geopandas object.

```
[6]: # make a geometry object from Lat, Long
     geo = gpd.points_from_xy(c['Long_'], c['Lat'])
     # give the geometry to geopandas together with c
     gc = gpd.GeoDataFrame(c, geometry=geo)
     gc.head()
```

```
[6]:      FIPS     Admin2  Province_State Country_Region          Last_Update ␣
  ↪\
   0  45001.0  Abbeville  South Carolina             US  2020-03-27 22:14:55
   1  22001.0     Acadia       Louisiana             US  2020-03-27 22:14:55
   2  51001.0   Accomack        Virginia             US  2020-03-27 22:14:55
   3  16001.0        Ada           Idaho             US  2020-03-27 22:14:55
   4  19001.0      Adair            Iowa             US  2020-03-27 22:14:55

            Lat       Long_  Confirmed  Deaths  Recovered  Active  \
   0  34.223334  -82.461707          4       0          0       0
   1  30.295065  -92.414197          8       1          0       0
   2  37.767072  -75.632346          2       0          0       0
   3  43.452658 -116.241552         54       0          0       0
   4  41.330756  -94.471059          1       0          0       0

                      Combined_Key                    geometry
   0  Abbeville, South Carolina, US   POINT (-82.46171 34.22333)
   1          Acadia, Louisiana, US   POINT (-92.41420 30.29506)
   2         Accomack, Virginia, US   POINT (-75.63235 37.76707)
   3                 Ada, Idaho, US  POINT (-116.24155 43.45266)
   4                Adair, Iowa, US   POINT (-94.47106 41.33076)
```

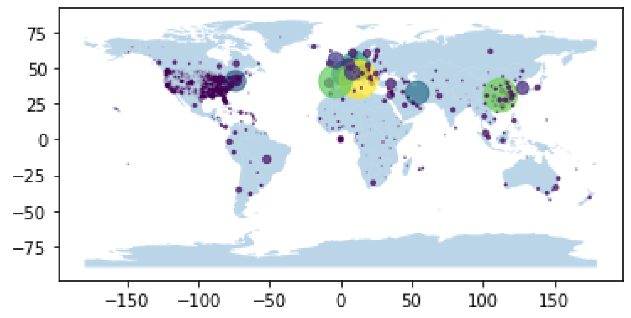
The only difference between gc and c is the last column, which contains the new geometry objects representing points on the globe. Next, in order to place markers at these points on a map of the world, we need to get a simple *low resolution* world map:

```
[7]: world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
     world.plot();
```



You can download and use maps with better resolution from Natural Earth, but that will be too far of a digression for this overview. On top of the above low resolution map, we can now put the markers whose sizes are proportional to the number of confirmed cases.

[8]:
```
base = world.plot(alpha=0.3)
msz = 500 * gc['Confirmed'] / gc['Confirmed'].max()
gc.plot(ax=base, column='Confirmed', markersize=msz, alpha=0.7);
```



These python tools have made it incredibly easy for us to immediately identify the COVID-19 trouble spots in the world. Moreover, these visualizations can be updated easily by re-running this code as data becomes available for other days.

I.5 Restricting to Oregon

Focusing on our part of the world, let us see how to restrict the COVID-19 data in the data frame `c` to Oregon.

[9]:
```
co = c[c['Province_State']=='Oregon']
```

The variable `co` now contains the data restricted to Oregon. However, we are now presented with a problem. To visualize the restricted data, we need a map of Oregon. The module `geopandas` does not carry any information about Oregon and its counties. However this information is available from the United States Census Bureau. (By the way, the 2020 census is happening now! Do not forget to respond to their survey. They are one of our authoritative sources of quality data.)

To visualize the COVID-19 information on a map of Oregon, we need to get the county boundary information from the census bureau. This illustrates a common situation that arises when trying to analyze data: it is often necessary to procure and merge data from multiple sources in order to understand a real-world phenomena.

A quick internet search reveals the census page with county information. The information is available in an online file `cb_2018_us_county_500k.zip` at the URL below. Python allows you to download this file using its `urllib` module without even needing to leave this notebook.

[10]:
```
# url of the data
census_url = 'https://www2.census.gov/geo/tiger/GENZ2018/shp/
↪cb_2018_us_county_500k.zip'
```

```
# location of your download
your_download_folder = '../../data_external'
if not os.path.isdir(your_download_folder):
    os.mkdir(your_download_folder)
us_county_file = your_download_folder + '/cb_2018_us_county_500k.zip'

# download if the file doesn't already exist
if not os.path.isfile(us_county_file):
    with urllib.request.urlopen(census_url) as response,␣
 ↪open(us_county_file, 'wb') as out_file:
        shutil.copyfileobj(response, out_file)
```

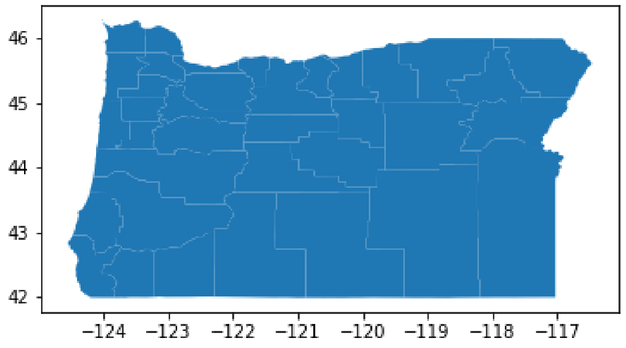
Now, your local computer has a zip file, which has among its contents, files with geometry information on the county boundaries, which can be read by geopandas. We let geopandas directly read in the zip file: it knows which information to extract from the zip archive to make a data frame with geometry.

[11]:
```
us_counties = gpd.read_file(f"zip://{us_county_file}")
us_counties.head()
```

[11]:
```
  STATEFP COUNTYFP  COUNTYNS        AFFGEOID  GEOID     NAME LSAD     ␣
 ↪ALAND  \
0      21      007  00516850  0500000US21007  21007  Ballard   06  ␣
 ↪639387454
1      21      017  00516855  0500000US21017  21017  Bourbon   06  ␣
 ↪750439351
2      21      031  00516862  0500000US21031  21031   Butler   06 ␣
 ↪1103571974
3      21      065  00516879  0500000US21065  21065   Estill   06  ␣
 ↪655509930
4      21      069  00516881  0500000US21069  21069  Fleming   06  ␣
 ↪902727151

     AWATER                                           geometry
0  69473325  POLYGON ((-89.18137 37.04630, -89.17938 37.053...
1   4829777  POLYGON ((-84.44266 38.28324, -84.44114 38.283...
2  13943044  POLYGON ((-86.94486 37.07341, -86.94346 37.074...
3   6516335  POLYGON ((-84.12662 37.64540, -84.12483 37.646...
4   7182793  POLYGON ((-83.98428 38.44549, -83.98246 38.450...
```

The object us_counties has information about all the counties. Now, we need to restrict this data to just that of Oregon. Looking at the columns, we find something called STATEFP. Searching through the government pages, we find that STATEFP refers to a 2-character state FIPS code. The FIPS code refers to Federal Information Processing Standard which was a "standard" at one time, then deemed obsolete, but still continues to be used today. All that aside, it suffices to note that Oregon's FIPS code is 41. Once we know this, python makes it is easy to restrict the data to Oregon:

```
[12]: ore = us_counties[us_counties['STATEFP']=='41']
      ore.plot();
```

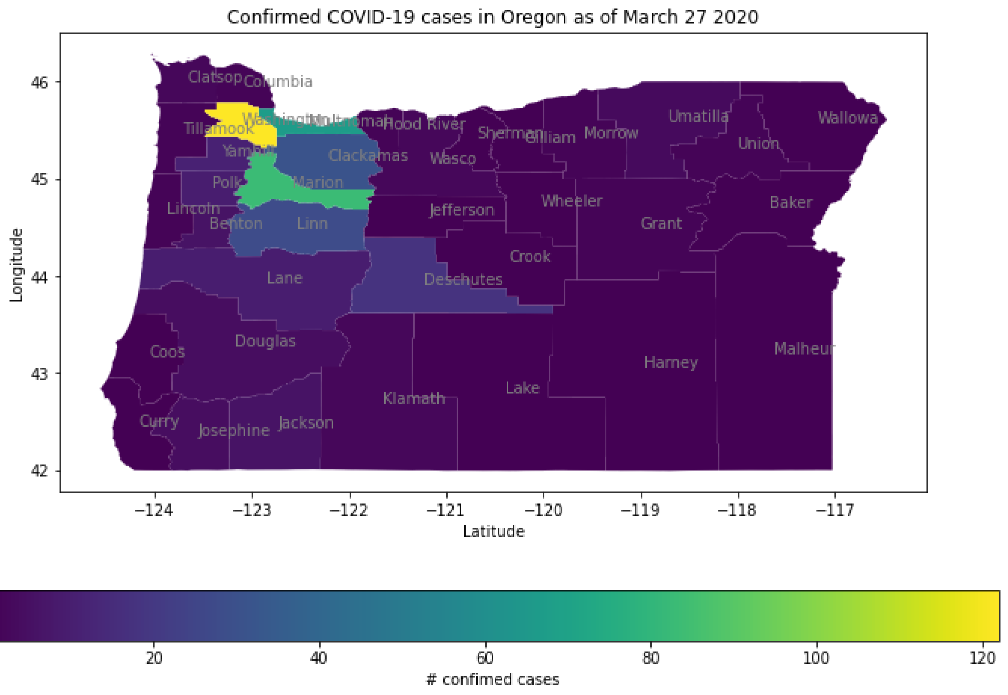


Now we have the Oregon data in two data frames, `ore` and `co`. We must combine the two data frames. This is again a situation so often encountered when dealing with real data that there is a facility for it in `pandas` called `merge`. Both data has FIPS codes: in `ore` you find it under column GEOID, and in `co` you find it called `FIPS`. The merged data frame is represented by the variable `orco` below:

```
[13]: ore = ore.astype({'GEOID': 'int64'}).rename(columns={'GEOID' : 'FIPS'})
      co = co.astype({'FIPS': 'int64'})
      orco = pd.merge(ore, co.iloc[:,:-1], on='FIPS')
```

The `orco` object now has both the geometry information as well as the COVID-19 information, making it extremely easy to visualize.

```
[14]: # plot coloring counties by number of confirmed cases

      fig, ax = plt.subplots(figsize=(12, 8))
      orco.plot(ax=ax, column='Confirmed', legend=True,
                legend_kwds={'label': '# confimed cases',
                             'orientation':'horizontal'})

      # label the counties

      for x, y, county in zip(orco['Long_'], orco['Lat'], orco['NAME']):
          ax.text(x, y, county, color='grey')

      ax.set_title('Confirmed COVID-19 cases in Oregon as of March 27 2020')
      ax.set_xlabel('Latitude'); ax.set_ylabel('Longitude');
```

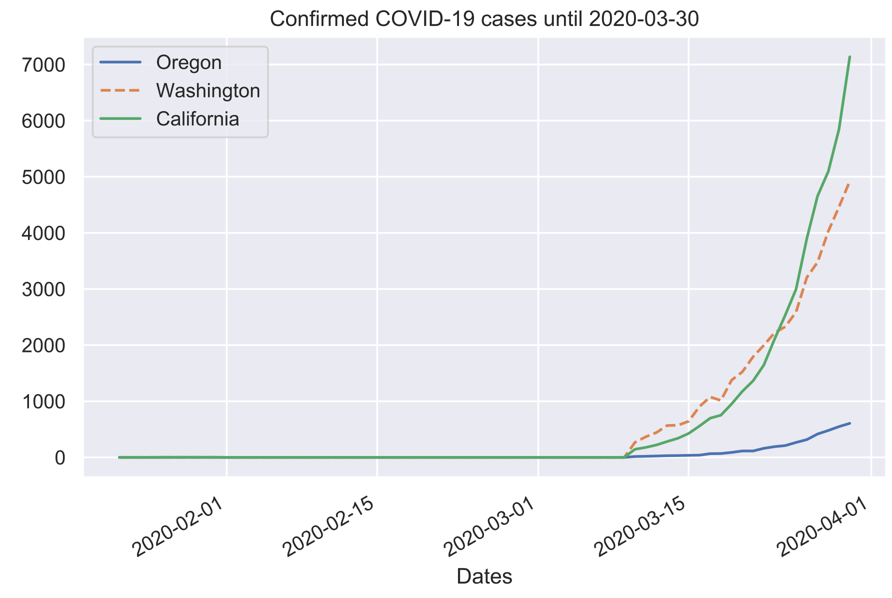


This is an example of a chloropleth map, a map where regions are colored or shaded in proportion to some data variable. It is an often-used data visualization tool.

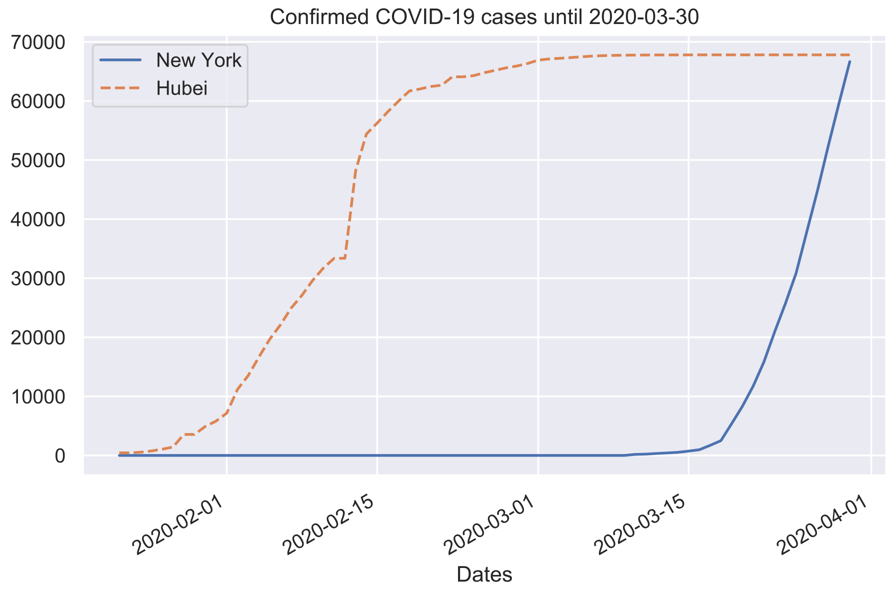
I.6 Ask the data

Different ways of displaying data often give different insights. There are many visualization tools in the python ecosystem and you will become more acquainted with these as we proceed.

Meanwhile, you might have many questions whose answers already lie in the data we have downloaded. For example, you may wonder how Oregon is doing in terms of COVID-19 outbreak compared to the other two west coast states. Here is the answer extracted from the same data:



How does the progression of infections in New York compare with Hubei where the disease started? Again the answer based on the data we have up to today is easy to extract, and is displayed next.



Of course, the COVID-19 situation is evolving, so these figures are immediately outdated after today's class. This situation is evolving in as yet unknown ways. I am sure that you, like me, want to know more about how these plots will change in the next few months. You will be able to generate plots like this and learn many more data analysis skills from these lectures. As you amass more technical skills, let me encourage you to answer your own questions on COVID-19 by returning to this overview, pulling the most recent data, and modifying the code here to your needs. In fact, some later assignments will require you to work further with this Johns Hopkins COVID-19 worldwide dataset. Visualizing the COVID-19 data for any other state, or indeed, any other region in the world, is easily accomplished by some small modifications to the code of this lecture.

II
Interacting with Python

March 31, 2020

Python is a modern, general-purpose, object-oriented, high-level programming language with a clean and expressive syntax. The following features make for easy code development and debugging in python:

- *Python code is interpreted:* There is no need to compile the code. Your code is read by a python interpreter and made into executable instructions for your computer in real time.
- *Python is dynamically typed:* There is no need to declare the type of a variable or the type of an input to a function.
- *Python has automatic garbage collection or memory management:* There is no need to explicitly allocate memory for variables before you use them or deallocate them after use.

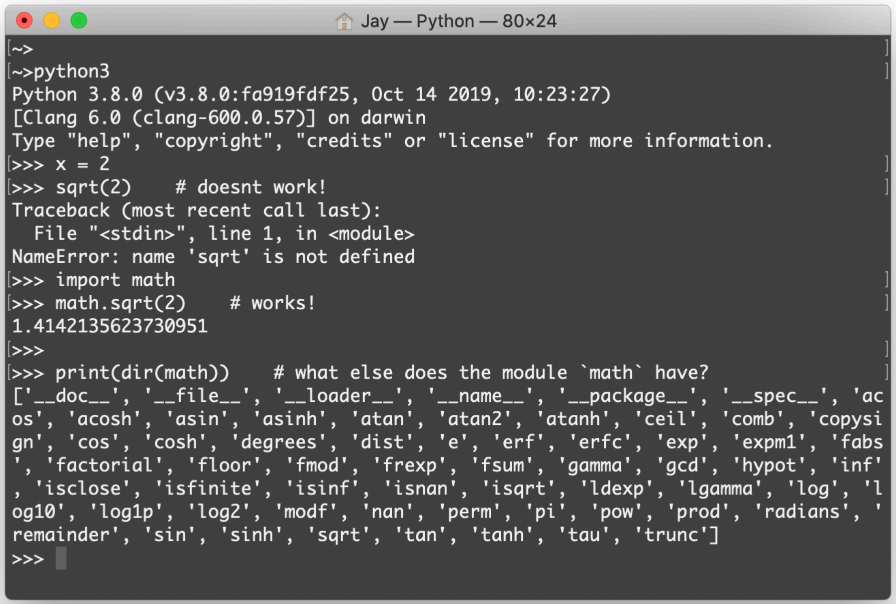
However, keep in mind that these features also make pure python code slower (than, say C) in repetitious loops because of repeated checking for the type of objects. Therefore many python modules (such as `numpy`, which we shall see in detail soon), have C or other compiled code, which is then wrapped in python to take advantage of python's usability without losing speed.

There are at least four ways to interact with your Python 3 installation.

1. Use a python shell
2. Use an iPython shell
3. Put code in a python file ending in `.py`
4. Write code + text in Jupyter notebook

II.1 Python shell

Type the python command you use in *your* system (`python` or `python3`) to get this shell. I will use `python3` since that is what my system requires, but please do make sure to replace it by `python` if that's what is needed on your system. Here is an image of the interactive python shell within a terminal.



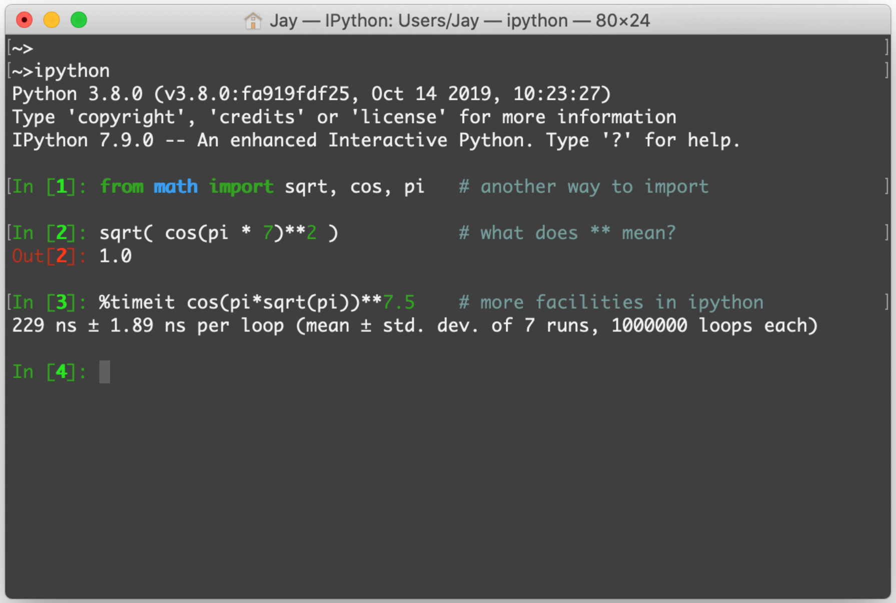
```
[~>
[~>python3
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> x = 2
[>>> sqrt(2)    # doesnt work!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
[>>> import math
[>>> math.sqrt(2)    # works!
1.4142135623730951
[>>>
[>>> print(dir(math))    # what else does the module `math` have?
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'ac
os', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysi
gn', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs
', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf'
, 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'l
og10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians', '
remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>> 
```

Note the following from the interactive session displayed in the figure above:

- Computing the square root of a number using `sqrt` is successful only after *importing* `math`. Most of the functionality in Python is provided by *modules*, like the `math` module. Some modules, like `math`, come with python, while others must be installed after python is installed.
- Strings that begin with `#` (like "`# works!`" in the figure) differentiate *comments* from code. This is the case in a python shell and also in the other forms of interacting with python discussed below.
- The `dir` command shows the facilities provided by a module. As you can see, the `math` module contains many functions in addition to `sqrt`.

II.2 iPython shell

A more powerful shell interactive environment is provided by the iPython shell (type in `ipython` or `ipython3` into your command prompt, or launch it from Anaconda navigator). The iPython shell has features like auto-completion, coloring, history of commands, automatic help by tacking on `?`, ability to interact with your operating system's commands, etc.



```
[~>
[~>ipython
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.9.0 -- An enhanced Interactive Python. Type '?' for help.

[In [1]: from math import sqrt, cos, pi    # another way to import

[In [2]: sqrt( cos(pi * 7)**2 )           # what does ** mean?
Out[2]: 1.0

[In [3]: %timeit cos(pi*sqrt(pi))**7.5    # more facilities in ipython
229 ns ± 1.89 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

In [4]: 
```

II.3 Jupyter Notebook

The Jupyter notebook is a web-browser based graphical environment consisting of cells, which can consist of code, or text. The text cells should contain text in markdown syntax, which allows you to type not just words in bold and italic, but also tables, mathematical formula using latex, etc. The code cells of Jupyter can contain code in various languages, but here we will exclusively focus on code cells with Python 3.

For example, this block of text that begins with this sentence marks the beginning of a jupyter notebook *cell* containing *markdown* content. If you are viewing this from jupyter, click on jupyter's top menu -> `Cell` -> `Cell Type` to see what is the type of the current cell, or to change the cell type. Computations must be done in a *code* cell, not a *markdown* cell. For example, to compute

$$\cos(\pi\sqrt{\pi})^7$$

we open a code cell next with the following two lines of python code:

```
[1]: from math import cos, sqrt, pi

     cos(pi*sqrt(pi))**7
```

```
[1]: 0.14008146171564725
```

This seamless integration of text and code makes Jupyter attractive for developing a reproducible environment for scientific computing.

II.4 Python file

Open your favorite text editor, type in some python code, and then save the file as `myfirstpy.py`. Here is a simple example of such a file.

```python
#------- myfirstpy.py ---------------------------------
from math import cos, sqrt, pi

print('Hello, I can compute! ')
x = 3
y = cos(pi*sqrt(pi)*x)**7
print('Starting from x =', x, 'we have computed y=', y)
#------------------------------------------------------
```

One executes such a python file by typing the following on the command line

```
python3 ../pyfiles/myfirstpy.py
```

Note that depending on your operating system, you may have to replace the above command by `python ..\pyfiles\myfirstpy.py` or similar variants.

You can also execute the python file in a platform-independent way from *within this Jupyter notebook* by loading the contents of the file into a cell. This is done using *line magic* command `%load ../pyfiles/myfirstpy.py`. Once you type in this command into a code cell and execute the cell, the contents of the file will be copied into the cell (and simultaneously, the `load` command will be commented out). Then, returning to the cell and executing the cell a *second time* runs the same code that was in the file.

[2]:
```python
# %load ../pyfiles/myfirstpy.py
from math import cos, sqrt, pi

print('Hello, I can compute! ')
x = 3
y = cos(pi*sqrt(pi)*x)**7
print('Starting from x =', x, 'we have computed y=', y)
```

```
Hello, I can compute!
Starting from x = 3 we have computed y= -0.013884089495354414
```

The above output cell should display the same output as what one would have obtained if we executed the python file on the command line.

For larger projects (including take-home assignments), you will need to create such python files with many lines of python code. Therefore it is essential that you know how to create and execute python files in your system.

III Working with git

April 2, 2020

Git a distributed version control system (and is a program often used independently of python). A version control system tracks the history of changes in projects with many files, including data files, and codes, which many people access simultaneously. Git facilitates identification of changes made, fetching revisions from a cloud repository in git format, and pushing revisions to the cloud.

GitHub is a cloud server that specializes in serving data in the form of `git` repositories. Many other such cloud services exists, such as Atlassian's *BitBucket*.

The notebooks that form these lectures are in a git repository served from GitHub. In this notebook, we describe how to access materials from this remote git repository. We will also use this opportunity to introduce some *object-oriented terminology* like classes, objects, constructor, data members, and methods, which are pervasive in python. Those already familiar with this terminology and GitHub may skip to the next activity.

III.1 Our materials in GitHub

Lecture notes, exercises, codes, and all accompanying materials can be found in the GitHub repository at https://github.com/jayggg/mth271content

One of the reasons we use git is that many continuously updated datasets, like the COVID-19 dataset, are served in git format. Another reason is that we may want to use current news and fresh data in our activities. Such activities may be prepared with very little lead time, so cloud git repositories are ideal for pushing in new materials as they get developed: once they are in the cloud, you have immediate access to them. After a lecture, the materials may be revised and updated per your feedback and these revisions will also be available for you from GitHub. Therefore, it is useful to be conversant with GitHub.

Let us spend a few minutes today on how to fetch materials from the git repository. In particular, executing this notebook will pull the updated data from GitHub and place it in a location you specify (below).

If you want to know more about git, there are many resources online, such as the Git Handbook. The most common way to fetch materials from a remote repository is using `git`'s command line tools, but for our purposes, the python code in this notebook will suffice.

III.2 Git `Repo` class in python

We shall use the python module `gitpython` to work with `git`. (We already used this module in the first overview lecture. The documentation of `gitpython` contains a lot of information on how to use its facilities. The main facility is the **class** called `Repo` which it uses to represent git repositories.

[1]:
```
from git import Repo
```

Python is an object-oriented language. Everything in the workspace is an **object.** An object is an instance of a class. The definition and features of the class `Repo` were imported into this workspace by the above line of code. A class has **members**, which could be **data members** or **attributes** (which themselves are objects residing in the class' memory layout), or function members, called **methods**, which provide functionalities of the class.

You can query the functionalities of `Repo` using `help`. Open a cell and type in

`help(Repo)`

You will see that the ouput contains the extensive documentation for objects of class `Repo`, including all its available methods.

Below, we will use the method called `clone_from`. Here is the class documentation for that method:

[2]:
```
help(Repo.clone_from)
```

```
Help on method clone_from in module git.repo.base:

clone_from(url, to_path, progress=None, env=None, multi_options=None, **kwargs) method of
builtins.type instance
    Create a clone from the given URL

    :param url: valid git url, see http://www.kernel.org/pub/software/scm/git/docs/git-
clone.html#URLS
    :param to_path: Path to which the repository should be cloned to
    :param progress: See 'git.remote.Remote.push'.
    :param env: Optional dictionary containing the desired environment variables.
        Note: Provided variables will be used to update the execution
        environment for `git`. If some variable is not specified in `env`
        and is defined in `os.environ`, value from `os.environ` will be used.
        If you want to unset some variable, consider providing empty string
        as its value.
    :param multi_options: See ``clone`` method
    :param kwargs: see the ``clone`` method
    :return: Repo instance pointing to the cloned directory
```

Classes have a special method called **constructor**, which you would find listed among its methods as `__init__`.

[3]:
```
help(Repo.__init__)
```

```
Help on function __init__ in module git.repo.base:

__init__(self, path=None, odbt=<class 'git.db.GitCmdObjectDB'>, search_parent_directories=False,
expand_vars=True)
    Create a new Repo instance
    :param path:
        the path to either the root git directory or the bare git repo::

            repo = Repo("/Users/mtrier/Development/git-python")
            repo = Repo("/Users/mtrier/Development/git-python.git")
            repo = Repo("~/Development/git-python.git")
            repo = Repo("$REPOSITORIES/Development/git-python.git")
            repo = Repo("C:\Users\mtrier\Development\git-python\.git")

        - In *Cygwin*, path may be a `'cygdrive/...'` prefixed path.
        - If it evaluates to false, :envvar:`GIT_DIR` is used, and if this also evals to false,
          the current-directory is used.
    :param odbt:
        Object DataBase type - a type which is constructed by providing
        the directory containing the database objects, i.e. .git/objects. It will
        be used to access all object data
    :param search_parent_directories:
        if True, all parent directories will be searched for a valid repo as well.

        Please note that this was the default behaviour in older versions of GitPython,
        which is considered a bug though.
    :raise InvalidGitRepositoryError:
    :raise NoSuchPathError:
    :return: git.Repo
```

The `__init__` method is called when you type in `Repo(...)` with the arguments allowed in `__init__`. Below, we will see how to initialize a `Repo` object using our github repository.

III.3 Your local copy of the repository

Next, each of you need to specify a location on *your* computer where you want the course materials to reside. This location can be specified as a string, where subfolders are delineated by forward slash. Please revise the string below to suit your needs.

[4]:
```
coursefolder = '/Users/Jay/tmpdir/'
```

Python provides a module `os` to perform operating system dependent tasks in a portable (platform-independent) way. If you did not give the *full* name of the folder, `os` can attempt to produce it as follows:

[5]:
```
import os
os.path.abspath(coursefolder)
```

[5]:
```
'/Users/Jay/tmpdir'
```

Please double-check that the output is what you expected on your operating system: if not, please go back and revise `coursefolder` before proceeding. (Windows users should see forward slashes converted to double backslashes, while mac and linux users will usually retain the forward slashes.)

We proceed to download the course materials from GitHub. These materials will be stored in a *subfolder* of `coursefolder` called `mth271content`, which is the name of the git repository.

[6]:
```
repodir = os.path.join(os.path.abspath(coursefolder), 'mth271content')
repodir   # full path name of the subfolder
```

[6]:
```
'/Users/Jay/tmpdir/mth271content'
```

Again, the value of the string variable `repodir` output above describes the location on your computer where your copy of the course materials from GitHub will reside.

III.4 Two cases

Now there are two cases to consider:

1. Are you downloading the remote git repository for the first time?
2. Or, are you returning to the remote repository to update the materials?

In Case 1, you want to **clone** the repository. This will create a local copy (on your computer) of the remote cloud repository.

In Case 2, you want to **pull** updates (only) from the repository, i.e., only changes in the remote cloud that you don't have in your existing local copy.

To decide which case you are in, I will assume the following. If the folder whose name is the value of the string `repodir` *already exists*, then I will assume you are in Case 2. Otherwise, you are in Case 1. To find out if a folder exists, we can use another facility from `os`:

[7]:
```
os.path.isdir(repodir)
```

[7]:
```
True
```

The output above should be `False` if you are running this notebook for the first time, per my assumption above. When you run it after you have executed this notebook successfully at least once, you would already have cloned the repository, so the folder will exist.

III.5 Clone or pull

The code below uses the conditionals `if` and `else` (included in the prerequisite reading for this lecture) to check if the folder exists: If it does not exist, a new local copy of the GitHub repository is cloned into your local hard drive. If it exists, then only the differences (or updates) between your local copy and the remote repository are fetched, so that your local copy is up to date with the remote.

[8]:
```
if os.path.isdir(repodir):       # if repo exists, pull newest data
    repo = Repo(repodir)
    repo.remotes.origin.pull()
else:                            # otherwise, clone from remote
    repo = Repo.clone_from('https://github.com/jayggg/mth271content',
                           repodir)
```

- Here `repo` is an **object** of **class** `Repo`.
- `Repo(repodir)` invokes the constructor, namely the `__init__` method.

- `Repo.clone_from(...)` calls the `clone_from(...)` method.

Now you have the updated course materials in your computer in a local folder. The object `repo` stores information about this folder, which you gave to the constructor in the string variable `repodir`, in a **data member** called `working_dir`. You can access any data members of an object in memory, and you do so just like you access a method, using a dot `.` followed by the member name. Here is an example:

[9]:
```
repo.working_dir
```

[9]:
```
'/Users/Jay/tmpdir/mth271content'
```

Note how the `Repo` object was either initialized with `repodir` (if that folder exists) or set to clone a remote repository at a URL.

III.6 Updated and future materials

The following instructions are for those of you who want to keep tracking the git repository closely in the future. Suppose you want to update your local folder with new materials from GitHub. But at the same time, you want to experiment and modify the notebooks as you like. This can create conflicting versions, which we should know how to handle.

Consider the situation where I have pushed changes to a file into the remote git repository that you want your local folder to reflect. But you have been working with the same file locally and have made changes to it - perhaps you have put a note to yourself to look something up, or perhaps you have found a better explanation, or better code, than what I gave. You want to keep your changes.

You should know that once you modify a file that is *tracked* by git as a local copy of a remote file, and you ask git to update, *git will refuse to overwrite your changes.* Because the remote version of the file and the local version of the file are now in *conflict*, a simple git pull command will fail. Git provides constructs to help resolve such conflicts, but let's try to keep things simple today. The following method is a solution that doubles the number of files, but has the advantage of simplicity:

Go to the `repodir` location in your computer. Copy the `jupyter` subfolder as, say `jupyterCopy`. Overwrite the copy of this notebook (called `03_Working_with_git.ipynb`) in the `jupyterCopy` folder with this file, which you saved after making your changes to variables like `coursefolder` above. Note that `jupyerCopy` is untracked by git: there is no remote folder in the cloud repository with that name. So any changes you make in `jupyterCopy` will be left untouched by git. So you can freely change any jupyter notebooks within this folder. The next time you run this file from `jupyterCopy` it will pull updates from the remote repository into the original `jupyter` folder. This way you get your updates from the cloud in `jupyter` and at the same time get to retain your modifications in `jupyterCopy`.

Alternately, if you like working on the command line, instead of running this notebook, you can run the python file update_course.py on the command line. You should move this file outside of the repository and save it after changing the value of the string `coursefolder` to your specific local folder name.

IV Conversion table

April 2, 2020

This elementary activity is intended to check and consolidate your understanding of very basic python language features. It is modeled after a similar activity in [HPL] and involves a simple temperature conversion formula. You may have seen kitchen cheat sheets (or have one yourself) like the following:

	Fahrenheit	Celsius
cool oven	200 F	90 C
very slow oven	250 F	120 C
slow oven	300-325 F	150-160 C
moderately slow oven	325-350 F	160-180 C
moderate oven	350-375 F	180-190 C
moderately hot oven	375-400 F	190-200 C
hot oven	400-450 F	200-230 C
very hot oven	450-500 F	230-260 C

This is modeled after a conversion table at the website Cooking Conversions for Old Time Recipes, which many found particularly useful for translating old recipes from Europe. Of course, the “old continent” has already moved on to the newer, more rational, metric system, so all European recipes, old and new, are bound to have temperatures in Celsius (C). Even if recipes don’t peak your interest, do know that every scientist must learn to work with the metric system.

Celsius values can be converted to the Fahrenheit system by the formula

$$F = \frac{9}{5}C + 32.$$

The task in this activity is to print a table of F and C values per this formula. While accomplishing this task, you will recall basic python language features, like `while` loop, `for` loop, `range`, `print`, list and tuples, `zip`, and list comprehension.

IV.1 Using the `while` loop

We start by making a table of F and C values, starting from 0 C to 250 C, using the `while` loop.

```
[1]: print('F      C')
```

```
C = 0
while C <= 250:
    F = 9 * C / 5 + 32
    print(F, C)
    C += 10
```

```
F     C
32.0 0
50.0 10
68.0 20
86.0 30
104.0 40
122.0 50
140.0 60
158.0 70
176.0 80
194.0 90
212.0 100
230.0 110
248.0 120
266.0 130
284.0 140
302.0 150
320.0 160
338.0 170
356.0 180
374.0 190
392.0 200
410.0 210
428.0 220
446.0 230
464.0 240
482.0 250
```

This cell shows how to add, multiply, assign, increment, print, and run a while loop. Such basic language features are introduced very well in the prerequisite reading for this lecture, the official python tutorial's section titled "An informal introduction to Python." (Note that all pointers to prerequisite reading materials are listed together just after the table of contents in the beginning.)

IV.2 Adjusting the printed output

Examining the output above, we note that it is not perfectly aligned like a printed table. Here is how we can use `print`'s features to format or align them to our tastes.

Formatting options like `%10.3f` can be used for alignment. It's easy to describe this by an example:

```
%10.3f: print 3 decimals, field width 10
 %9.2e: print 2 decimals, field width 9, scientific notation
```

Type `help(print)` to recall these and other options. Below, we use a fixed width of 4 to format F and C values.

[2]:
```
print('   F    C')

C = 0
```

```
while C <= 250:
    F = 9 * C / 5 + 32
    print('%4.0f %4.0f' % (F, C))
    C += 10
```

```
   F    C
  32    0
  50   10
  68   20
  86   30
 104   40
 122   50
 140   60
 158   70
 176   80
 194   90
 212  100
 230  110
 248  120
 266  130
 284  140
 302  150
 320  160
 338  170
 356  180
 374  190
 392  200
 410  210
 428  220
 446  230
 464  240
 482  250
```

IV.3 Do the same using `for` loop

In addition to the `while` loop construct, python also has a `for` loop, which is often safer from an accidental bug sending the system into an infinite loop. Also recall the very useful `range` construct. The loop statement

```
for i in range(4):
```

runs over `i=0,1,2,3` implicitly using `range`'s default starting value `0` and the default stepping value `1`. For our temperature conversion task, we step by 10 degrees instead of the default value of 1:

[3]:
```
print('   F    C')
for C in range(0, 250, 10):
    F = 9 * C / 5 + 32
    print('%4.0f %4.0f' % (F, C))
```

```
   F    C
  32    0
  50   10
  68   20
  86   30
 104   40
 122   50
 140   60
 158   70
 176   80
```

```
194   90
212  100
230  110
248  120
266  130
284  140
302  150
320  160
338  170
356  180
374  190
392  200
410  210
428  220
446  230
464  240
```

IV.4 Is there a temperature whose F and C values are equal?

As you can see from the above values, for a 10 degree increase in the C column, we see a corresponding 18 degree increase in the F column. Due to the these different rates of increase, we should see the values coincide by going to lower C values. Focusing on lower C values, let us run the `for` loop again:

```
[4]: print('   F    C')
     for C in range(-50, 50, 5):
         F = 9 * C / 5 + 32
         print('%4.0f %4.0f' % (F, C))
```

```
   F    C
 -58  -50
 -49  -45
 -40  -40
 -31  -35
 -22  -30
 -13  -25
  -4  -20
   5  -15
  14  -10
  23   -5
  32    0
  41    5
  50   10
  59   15
  68   20
  77   25
  86   30
  95   35
 104   40
 113   45
```

As you see from the output above, at -40 degrees, the Fahrenheit scale and the Celsius scale coincide. If you have lived in Minnesota, you probably know how -40 feels like, and you likely already know the fact we just discovered above (it's common for Minnesotans to throw around this tidbit while commiserating in the cold).

IV.5 Store in a list

If we want to use the above-printed tables later, we would have to run a loop again. Our conversion problem is so small that there is no cost to run as many loops as we like, but in many practical problems, loops contains expensive computations. So one often wants to store the quantities computed in the loop in order to reuse them later. Lists are good constructs for this.

First we should note that python has *lists* and also *tuples*. Only the former can be modified after creation. Here is an example of a list:

```
[5]: Cs = [0, 10]      # create list using []
     Cs.append(20)    # modify by appending an entry
     Cs
```

```
[5]: [0, 10, 20]
```

And here is an example of a tuple:

```
[6]: Cs = (0, 10)      # create a tuple using ()
```

You access a tuple element just like a list element, so `Cs[0]` will give the first element whether or not `Cs` is a list or a tuple. But the statement `Cs[0] = -10` that changes an element of the container will work only if `Cs` is a list. We say that a list is *mutable*, while a tuple is *immutable*. Tuples are generally faster than lists, but lists are more flexible than tuples.

Here is an example of how to store the computed C and F values within a loop into lists.

```
[7]: Cs = []    # empty list
     Fs = []

     for C in range(0, 250, 25):
         Cs.append(C)
         Fs.append(9 * C / 5 + 32)
```

The lists `Cs` and `Fs` can be accessed later:

```
[8]: print(Cs)
```

```
[0, 25, 50, 75, 100, 125, 150, 175, 200, 225]
```

```
[9]: print(Fs)
```

```
[32.0, 77.0, 122.0, 167.0, 212.0, 257.0, 302.0, 347.0, 392.0, 437.0]
```

This is not as pretty an output as before. But we can easily run a loop and print the stored values in any format we like. This is a good opportunity to show off a *pythonic* feature `zip` that allows you to traverse two lists *simultaneously:*

[10]:
```
print('   F   C')
for C, F in zip(Cs, Fs):
    print('%4.0f %4.0f' % (F, C))
```

```
   F    C
  32    0
  77   25
 122   50
 167   75
 212  100
 257  125
 302  150
 347  175
 392  200
 437  225
```

IV.6 List comprehension

An alternate and very interesting way to make lists in python is by the *list comprehension* feature. Codes with list comprehension read almost like English. Let's illustrate this by creating the list of F values from the existing list `Cs` of C values. Instead of making `Fs` in a loop as above, in a *list comprehension*, we just say that each value of the list `Fs` is obtained applying a formula *for each `C` in a list `Cs`:*

[11]:
```
Fs = [9 * C / 5 + 32   for C in Cs]
```

Note how this makes for compact code without sacrificing readability: constructs like this are why your hear so much praise for python's expressiveness. For mathematicians, the list comprehension syntax is also reminiscent of the set notation in mathematics: the set (list) Fs is described in mathematical notation by

$$\mathsf{Fs} = \left\{ \frac{9}{5}C + 32 : \ C \in \mathsf{Cs} \right\}.$$

Note how similar it is to the list comprehension code. (Feel free to check that the `Fs` computed by the above one-liner is the same as the `Fs` we computed previously within a loop.)

V

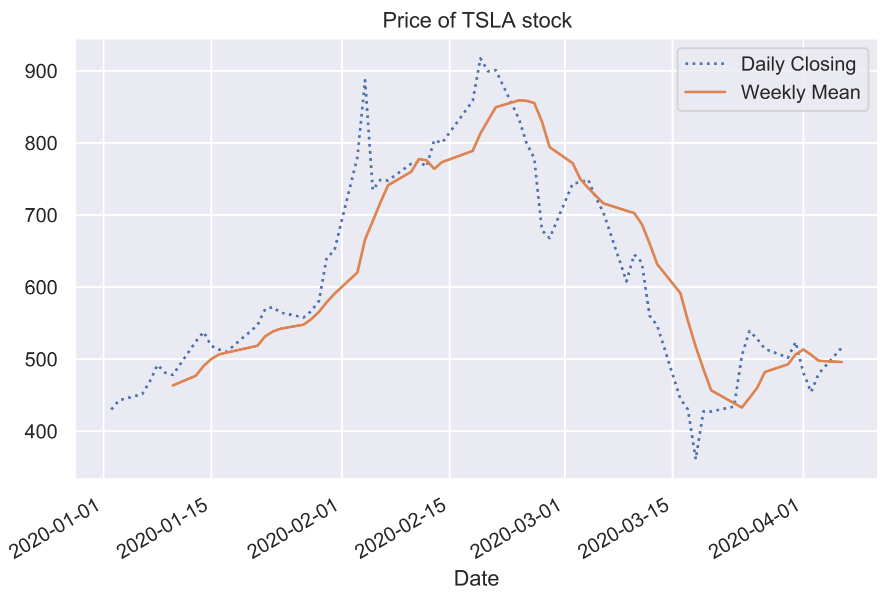
Approximating the derivative

April 7, 2020

In calculus, you learnt about the derivative and its central role in modeling processes where a rate of change is important. How do we compute the derivate on a computer?

Recall what you did in your first calculus course to compute the derivative. You memorized derivatives of simple functions like $\cos x, \sin x, \exp x, x^n$ etc. Then you learnt rules like product rule, quotient rule, chain rule etc. In the end you could systematically compute derivatives of complicated functions by reducing it to simpler components and applying the rules. We could teach the computer the same rules and come up with an algorithm for computing derivatives. This is the idea behind automatic differentiation. Python modules like `sympy` can compute derivatives symbolically in this fashion. However, this approach has its limits.

In the real world, we often encounter complicated functions, such as functions that cannot be represented in terms of simple component functions, or functions whose values you can only query from some proprietary company code, or functions whose values are based off a table, like for instance this function.



This function represents Tesla's stock prices this year until yesterday (which I got, in case you are curious, using just a few lines of python code). The function is complicated (not to mention depressing - it reflects the market downturn due to the pandemic). But its rate of change drives some investment decisions. Instead of the oscillatory daily stock values, analysts often look at the rate of change of trend lines (like the rolling weekly means above), a function certainly not expressible in terms of a few simple functions like sines or cosines.

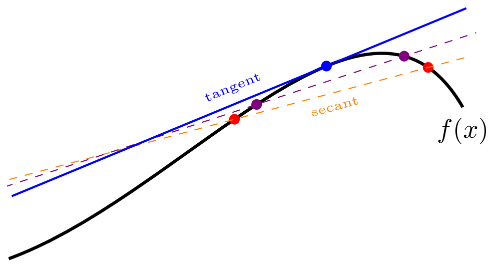
In this activity, we look at computing a numerical approximation to the derivative using something you learnt in calculus.

V.1 Numerical differentiation

Suppose f is a function of a single real variable x. Its derivative at any point x is the slope of the tangent of its graph at x. This slope, as you no doubt recall from calculus, can be numerically approximated by the slope of a secant line:

$$f'(x) \approx \frac{f(x+h/2) - f(x-h/2)}{h}$$

Below is a plot of the tangent line of some function f at x, whose slope is $f'(x)$, together with the secant line whose slope is the approximation on the right hand side above. Clearly as the spacing h decreases, the secant line becomes a better and better approximation to the tangent line.



The right hand side formula

$$\frac{f(x+h/2) - f(x-h/2)}{h}$$

can be implemented in python as long as we can compute the values $f(x+h/2)$ and $f(x-h/2)$. As $h \to 0$, we should a good obtain approximation to $f'(x)$.

V.2 Second derivative

We take one further step and approximate the second derivative by

$$\begin{aligned} f''(x) &\approx \frac{f'(x+h/2) - f'(x-h/2)}{h} \\ &\approx \frac{\left(\frac{f(x+h/2+h/2)-f(x+h/2-h/2)}{h}\right) - \left(\frac{f(x-h/2)-f(x-h/2-h/2)}{h}\right)}{h} \\ &\approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \end{aligned}$$

This is the **Central Difference Formula** for the second derivative.

The first task in this activity is to write a function to compute the above-stated second derivative approximation,

$$\frac{f(x-h) - 2f(x) + f(x+h)}{h^2}$$

given any function f of a single variable x. The parameter h should also be input, but can take a default value of 10^{-6}.

The prerequisite reading for this activity included python functions, keyword arguments, positional arguments, and lambda functions. Let's apply all of these concepts while computing the derivative approximation. Note that python allows you to pass functions themselves as arguments to other functions. Therefore, without knowing what specific function f to apply the central difference formula, we can write a generic function `D2` for implementing the formula for any f.

[1]:
```
def D2(f, x, h=1E-6):
    return (f(x-h) - 2*f(x) + f(x+h)) / (h*h)
```

Let's apply the formula to some nice function, say the sine function.

[2]:
```
from math import sin

D2(sin, 0.2)
```

[2]:
```
-0.19864665468105613
```

Of course we know that second derivative of $\sin(x)$ is negative of itself, so a quick test of correctness is to compare the above value to that of $-\sin(0.2)$.

[3]:
```
-sin(0.2)
```

[3]:
```
-0.19866933079506122
```

How do we apply `D2` to, say, $\sin(2x)$? One way is to define a function returning $\sin(2 * x)$ and then pass it to `D2`, as follows.

[4]:
```
def g(x):
    return sin(2*x)

D2(g, 0.2)
```

[4]:
```
-1.5576429035490946
```

An alternate way is using a `lambda` function. This gives a one-liner without damaging code readability.

[5]:
```
D2(lambda x: sin(2*x), 0.2)   # central diff approximation
```

```
[5]: -1.5576429035490946
```

Of course, in either case the computed value approximates the actual value of $\sin''(2x) = -4\sin(2x)$, thus verifying our code.

```
[6]: -4*sin(2* 0.2)              # actual 2nd derivative value
```

```
[6]: -1.557673369234602
```

V.3 Error

The error in the approximation formula we just implemented is

$$\varepsilon(x) = f''(x) - \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}$$

Although we can't know the error $\varepsilon(x)$ without knowing the true value $f''(x)$, calculus gives you all the tools to bound this error.

Substituting the Taylor expansions

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f''''(x) + \cdots$$

and

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f''''(x) + \cdots$$

into the definition of $\varepsilon(x)$, we find that the after several cancellations, the dominant term is $O(h^2)$ as $h \to 0$.

This means that if h is halved, the error should decrease by a factor of 4. Let us take a look at the error in the derivative approximations applied to a simple function

$$f(x) = x^{-6}$$

at, say $x = 1$. I am sure you can compute the exact derivative using your calculus knowledge. In the code below, we subtract this exact derivative from the computed derivative approximation to obtain the error.

```
[7]: print(' h      D2 Result   Error')
     for k in range(4,8):
         h = 2**(-k)
         d2g = D2(lambda x: x**-6, 1, h=h)
         e = d2g - 42
         print('%.0e  %.5f  %7.6f' %(h, d2g, e))
```

```
 h      D2 Result  Error
6e-02  42.99863  0.998629
3e-02  42.24698  0.246977
2e-02  42.06158  0.061579
8e-03  42.01538  0.015384
```

Clearly, we observe that the error decreases by a factor of 4 when h is halved. This is in accordance with what we expected from the Taylor expansion analysis above.

V.4 Limitations

A serious limitation of numerical differentiation formulas like this can be seen when we take values of h really close to 0. Although the limiting process in calculus relies on h going to 0, your computer is not equipped to deal with very small numbers. This creates issues. Instead of halving h, let us aggressively reduce h by a factor of 10, going down to 10^{-13} and look at the results.

[8]:
```
for k in range(1,14):
    h = 10**(-k)
    d2g = D2(lambda x: x**-6,1, h)
    print('%.0e %18.5f' %(h, d2g))
```

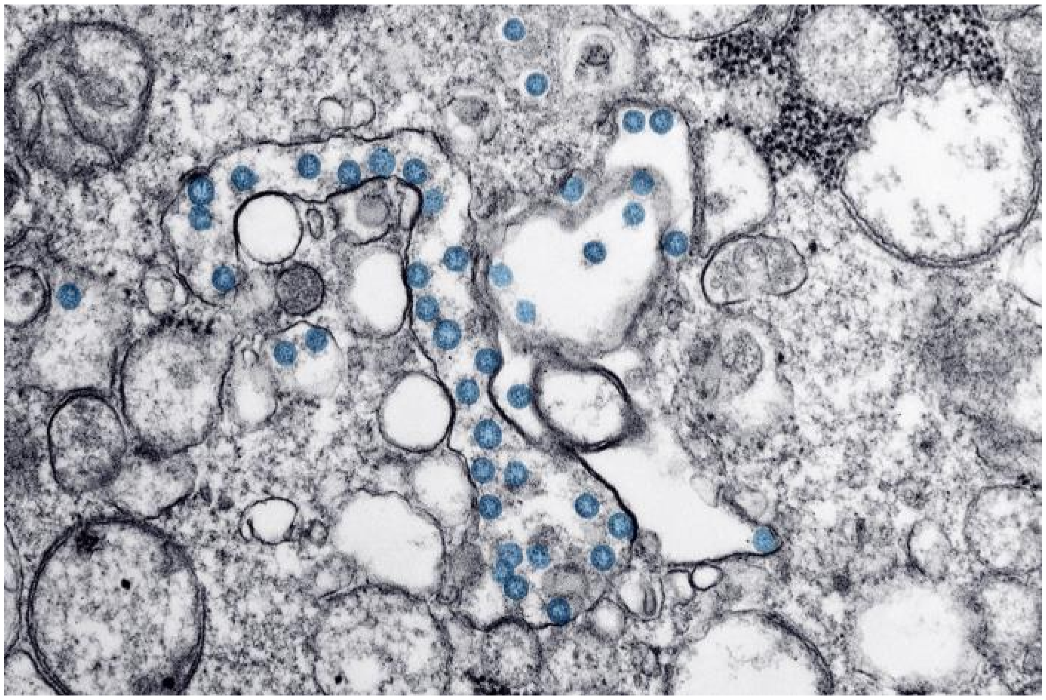
```
1e-01           44.61504
1e-02           42.02521
1e-03           42.00025
1e-04           42.00000
1e-05           41.99999
1e-06           42.00074
1e-07           41.94423
1e-08           47.73959
1e-09         -666.13381
1e-10            0.00000
1e-11            0.00000
1e-12   -666133814.77509
1e-13  66613381477.50939
```

Although a mathematical argument led us to expect better approximations as $h \to 0$, we find that the results from our computer for $h < 10^{-8}$ are totally wrong! The problem is that computers cannot do exact arithmetic: the infinite real number system is replaced by a finite set of numbers allowed in the so-called IEEE standard. This causes errors, called *round-off errors* that are different from the approximation error $\varepsilon(x)$ we discussed. Specifically, what happened was that for small h we subtracted very closeby numbers, creating round-off errors; we then multiplied by a big number $(1/h^2)$ amplifying these round-off errors. We shall not deal in depth with round-off errors in this course, but it pays to be wary of them.

VI Genome of SARS-CoV-2

April 7, 2020

Since most data come in files and streams, a data scientist must be able to effectively work with them. Python provides many facilities to make this easy. In this class activity, we will review some of python's file, string, and dictionary facilities by examining a file containing the genetic code of the virus that has been disrupting our lives this term. Here is a transmission electron micrograph showing the virus (a public domain image from the CDC, credited to H. A. Bullock and A. Tamin).



The genetic code of each living organism is a long sequence of simple molecules called **nucleotides** or **bases**. Although many nucleotides exist in nature, only 4 nucleotides, labeled A, C, G, and T, have been found in DNA. They are abbreviations of Adenine, Cytosine, Guanine, and Thymine. Although it is difficult to put viruses in the category of living organisms, they also have genetic codes made up of nucleotides.

VI.1 Get the genome

The NCBI (National Center for Biotechnology Information) has recently started maintaining a data hub for genetic sequences related to the virus causing COVID-19. Recall that the name of the virus is SARS-CoV-2 (which is different from the name of the disease, COVID-19), or "Severe Acute Respiratory Syndrome Coronavirus 2" in full. Searching the NCBI website with the proper virus name will help you locate many publicly available data sets.

Let's download NCBI's Reference Sequence NC_045512 giving the complete genome extracted from a sample of SARS-CoV-2 from the Wuhan seafood market, called the Wuhan-Hu-1 isolate. Here is a code using `urllib` that will attempt to directly download from the url specified below. It is unclear if this url would serve as a stable permanent link. In the event you have problems executing the next cell, please just head over to the webpage for NC_045512, click on "FASTA" (a data format) and then click on "Send to" a file. Then save the file in the same relative location mentioned below in `f` within the folder where we have been putting all the data files in this course.

[1]:
```
# NCBI  url:

url = 'https://www.ncbi.nlm.nih.gov/sviewer/viewer.cgi?tool=portal&' + \
    'save=file&log$=seqview&db=nuccore&report=fasta&id=1798174254&' + \
    'extrafeat=null&conwithfeat=on&hide-cdd=on'

# your local downloaded file:

f = '../../data_external/SARS-CoV-2-Wuhan-NC_045512.2.fasta'
```

[2]:
```
import os
import urllib
import shutil

if not os.path.isdir('../../data_external/'):
    os.mkdir('../../data_external/')

r = urllib.request.urlopen(url)
fo = open(f, 'wb')
shutil.copyfileobj(r, fo)
fo.close()
```

As mentioned in the page describing the data, this file gives the RNA of the virus.

[3]:
```
lines = open(f, 'r').readlines()
```

The file has been opened in read-only mode. The variable `lines` contains a list of all the lines of the file. Here are the first five lines:

[4]:
```
lines[0:5]
```

[4]:
```
['>NC_045512.2 Severe acute respiratory syndrome coronavirus 2 isolate␣
 ↪Wuhan-
Hu-1, complete genome\n',
␣
 ↪'ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTGTTCTCTAAA\n',
␣
 ↪'CGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACTCACGCAGTATAATTAATAAC\n',
␣
 ↪'TAATTACTGTCGTTGACAGGACACGAGTAACTCGTCTATCTTCTGCAGGCTGCTTACGGTTTCGTCCGTG\n',
␣
 ↪'TTGCAGCCGATCATCAGCACATCTAGGTTTCGTCCGGGTGTGACCGAAAGGTAAGATGGAGAGCCTTGTC\n']
```

The first line is a description of the data. The long genetic code is broken up into the following lines. We need to strip end-of-line characters from each such line to re-assemble the RNA string. Here is a way to strip off the end-of-line character:

```
[5]: lines[1].strip()
```

```
[5]: 'ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTGTTCTCTAAA'
```

Let's do so for every line starting ignoring the first. Since `lines` is a list object, ignoring the first element of the list is done by `lines[1:]`. (If you don't know this already, you must review the list access constructs.) The following code uses the string operation `join` to put together the lines into one long string. This is the RNA of the virus.

```
[6]: rna = ''.join([line.strip() for line in lines[1:]])
```

The first thousand characters and the last thousand characters of the RNA of the coronavirus are printed below:

```
[7]: rna[:1000]
```

```
[7]: 'ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTGTTCTCTAAACGAACTTTA
AAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACTCACGCAGTATAATTAATAACTAATTACTGTCGTTGACAG
GACACGAGTAACTCGTCTATCTTCTGCAGGCTGCTTACGGTTTCGTCCGTGTTGCAGCCGATCATCAGCACATCTAGGTT
TCGTCCGGGTGTGACCGAAAGGTAAGATGGAGAGCCTTGTCCCTGGTTTCAACGAGAAAACACACGTCCAACTCAGTTTG
CCTGTTTTACAGGTTCGCGACGTGCTCGTACGTGGCTTTGGAGACTCCGTGGAGGAGGTCTTATCAGAGGCACGTCAACA
TCTTAAAGATGGCACTTGTGGCTTAGTAGAAGTTGAAAAAGGCGTTTTGCCTCAACTTGAACAGCCCTATGTGTTCATCA
AACGTTCGGATGCTCGAACTGCACCTCATGGTCATGTTATGGTTGAGCTGGTAGCAGAACTCGAAGGCATTCAGTACGGT
CGTAGTGGTGAGACACTTGGTGTCCTTGTCCCTCATGTGGGCGAAATACCAGTGGCTTACCGCAAGGTTCTTCTTCGTAA
GAACGGTAATAAAGGAGCTGGTGGCCATAGTTACGGCGCCGATCTAAAGTCATTTGACTTAGGCGACGAGCTTGGCACTG
ATCCTTATGAAGATTTTCAAGAAAACTGGAACACTAAACATAGCAGTGGTGTTACCCGTGAACTCATGCGTGAGCTTAAC
GGAGGGGCATACACTCGCTATGTCGATAACAACTTCTGTGGCCCTGATGGCTACCCTCTTGAGTGCATTAAAGACCTTCT
AGCACGTGCTGGTAAAGCTTCATGCACTTTGTCCGAACAACTGGACTTTATTGACACTAAGAGGGGTGTATACTGCTGCC
GTGAACATGAGCATGAAATTGCTTGGTACACGGAACGTTCT'
```

```
[8]: rna[-1000:]
```

```
[8]: 'GCTGGCAATGGCGGTGATGCTGCTCTTGCTTTGCTGCTGCTTGACAGATTGAACCAGCTTGAGAGCAAAATGTCTGGTA
AAGGCCAACAACAACAAGGCCAAACTGTCACTAAGAAATCTGCTGCTGAGGCTTCTAAGAAGCCTCGGCAAAAACGTACT
GCCACTAAAGCATACAATGTAACACAAGCTTTCGGCAGACGTGGTCCAGAACAAACCCAAGGAAATTTTGGGGACCAGGA
```

```
ACTAATCAGACAAGGAACTGATTACAAACATTGGCCGCAAATTGCACAATTTGCCCCCAGCGCTTCAGCGTTCTTCGGAA
TGTCGCGCATTGGCATGGAAGTCACACCTTCGGGAACGTGGTTGACCTACACAGGTGCCATCAAATTGGATGACAAAGAT
CCAAATTTCAAAGATCAAGTCATTTTGCTGAATAAGCATATTGACGCATACAAAACATTCCCACCAACAGAGCCTAAAAA
GGACAAAAAGAAGAAGGCTGATGAAACTCAAGCCTTACCGCAGAGACAGAAGAAACAGCAAACTGTGACTCTTCTTCCTG
CTGCAGATTTGGATGATTTCTCCAAACAATTGCAACAATCCATGAGCAGTGCTGACTCAACTCAGGCCTAAACTCATGCA
GACCACACAAGGCAGATGGGCTATATAAACGTTTTCGCTTTTCCGTTTACGATATATAGTCTACTCTTGTGCAGAATGAA
TTCTCGTAACTACATAGCACAAGTAGATGTAGTTAACTTTAATCTCACATAGCAATCTTTAATCAGTGTGTAACATTAGG
GAGGACTTGAAAGAGCCACCACATTTTCACCGAGGCCACGCGGAGTACGATCGAGTGTACAGTGAACAATGCTAGGGAGA
GCTGCCTATATGGAAGAGCCCTAATGTGTAAAATTAATTTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGG
AGAATGACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
```

Here is the total length of the RNA:

[9]:
```
len(rna)
```

[9]:
```
29903
```

While the human genome is over 3 billion in length, the genome of this virus does not even reach the length of 30000.

VI.2 Finding a protein

When describing RNA, the T (Thymine) is often replaced by U (Uracil). This is done for example in an interesting New York Times article that came out last Friday. The article explains how this RNA code makes infected host cells produce a variety of proteins. Scientists have a good understanding of what some of these proteins do, but not all.

Here is a quote from the article on a protein it nicknamed **Virus Liberator. ORF7a**

```
When new viruses try to escape a cell, the cell can snare them with
proteins called tetherin. Some research suggests that ORF7a cuts
down an infected cell's supply of tetherin, allowing more of the
viruses to escape. Researchers have also found that the protein can
trigger infected cells to commit suicide - which contributes to the
damage Covid-19 causes to the lungs.
```

The article then gives the **ORF7a** sequence, which I have copied and pasted into the next cell, adding some string breaks. Note how the article has used lower case characters and the character u instead of T.

[10]:
```
orf7a = \
'augaaaauuauucuuuucuuggcacugauaacacucgcuacuugugagcuuuaucacuaccaag' + \
'aguguguuagagguacaacaguacuuuuaaaagaaccuugcucuucuggaacauacgagggcaa' + \
'uucaccauuucauccucuagcugauaacaaauuugcacugacuugcuuuagcacucaauuugcu' + \
'uuugcuuguccugacggcguaaaacacgucuaucaguuacgugccagaucaguuucaccuaaac' + \
'uguucaucagacaagaggaaguucaagaacuuuacucuccaauuuuucuuauuguugcggcaau' + \
'aguguuuauaacacuuugcuucacacucaaaagaaagacagaaugauugaacuuucauuaauug' + \
'acuucuauuugugcuuuuuagccuuucugcuauuccuuguuuuaauuaugcuuauuaucuuuug' + \
'guucucacuugaacugcaagaucauaaugaaacuugucacgccuaaacgaac'
```

The next task in this class activity is to find if this sequence occurs in the RNA we just downloaded, and if it does, where it occurs. To this end, we first make the replacements required to read the string in terms of A, T, G, and C.

```
[11]: s=orf7a.replace('u', 'T').replace('a', 'A').replace('g', 'G').replace('c',␣
        ↪'C')
      s
```

```
[11]: 'ATGAAAATTATTCTTTTCTTGGCACTGATAACACTCGCTACTTGTGAGCTTTATCACTACCAAGAGTGTGTTAGAGGTA
      CAACAGTACTTTTAAAAGAACCTTGCTCTTCTGGAACATACGAGGGCAATTCACCATTTCATCCTCTAGCTGATAACAAA
      TTTGCACTGACTTGCTTTAGCACTCAATTTGCTTTTGCTTGTCCTGACGGCGTAAAACACGTCTATCAGTTACGTGCCAG
      ATCAGTTTCACCTAAACTGTTCATCAGACAAGAGGAAGTTCAAGAACTTTACTCTCCAATTTTTCTTATTGTTGCGGCAA
      TAGTGTTTATAACACTTTGCTTCACACTCAAAAGAAAGACAGAATGATTGAACTTTCATTAATTGACTTCTATTTGTGCT
      TTTTAGCCTTTCTGCTATTCCTTGTTTTAATTATGCTTATTATCTTTTGGTTCTCACTTGAACTGCAAGATCATAATGAA
      ACTTGTCACGCCTAAACGAAC'
```

The next step is now a triviality in view of python's exceptional string handling mechanisms:

```
[12]: s in rna
```

```
[12]: True
```

We may also easily find the location of the ORF7a sequence and read off the entire string beginning with the sequence.

```
[13]: rna.find(s)
```

```
[13]: 27393
```

```
[14]: rna[27393:]
```

```
[14]: 'ATGAAAATTATTCTTTTCTTGGCACTGATAACACTCGCTACTTGTGAGCTTTATCACTACCAAGAGTGTGTTAGAGGTA
      CAACAGTACTTTTAAAAGAACCTTGCTCTTCTGGAACATACGAGGGCAATTCACCATTTCATCCTCTAGCTGATAACAAA
      TTTGCACTGACTTGCTTTAGCACTCAATTTGCTTTTGCTTGTCCTGACGGCGTAAAACACGTCTATCAGTTACGTGCCAG
      ATCAGTTTCACCTAAACTGTTCATCAGACAAGAGGAAGTTCAAGAACTTTACTCTCCAATTTTTCTTATTGTTGCGGCAA
      TAGTGTTTATAACACTTTGCTTCACACTCAAAAGAAAGACAGAATGATTGAACTTTCATTAATTGACTTCTATTTGTGCT
      TTTTAGCCTTTCTGCTATTCCTTGTTTTAATTATGCTTATTATCTTTTGGTTCTCACTTGAACTGCAAGATCATAATGAA
      ACTTGTCACGCCTAAACGAACATGAAATTTCTTGTTTTCTTAGGAATCATCACAACTGTAGCTGCATTTCACCAAGAATG
      TAGTTTACAGTCATGTACTCAACATCAACCATATGTAGTTGATGACCCGTGTCCTATTCACTTCTATTCTAAATGGTATA
      TTAGAGTAGGAGCTAGAAAATCAGCACCTTTAATTGAATTGTGCGTGGATGAGGCTGGTTCTAAATCACCCATTCAGTAC
      ATCGATATCGGTAATTATACAGTTTCCTGTTTACCTTTTACAATTAATTGCCAGGAACCTAAATTGGGTAGTCTTGTAGT
      GCGTTGTTCGTTCTATGAAGACTTTTTAGAGTATCATGACGTTCGTGTTGTTTTAGATTTCATCTAAACGAACAAACTAA
      AATGTCTGATAATGGACCCCAAAATCAGCGAAATGCACCCCGCATTACGTTTGGTGGACCCTCAGATTCAACTGGCAGTA
      ACCAGAATGGAGAACGCAGTGGGGCGCGATCAAAACAACGTCGGCCCCAAGGTTTACCCAATAATACTGCGTCTTGGTTC
      ACCGCTCTCACTCAACATGGCAAGGAAGACCTTAAATTCCCTCGAGGACAAGGCGTTCCAATTAACACCAATAGCAGTCC
      AGATGACCAAATTGGCTACTACCGAAGAGCTACCAGACGAATTCGTGGTGGTGACGGTAAAATGAAAGATCTCAGTCCAA
      GATGGTATTTCTACTACCTAGGAACTGGGCCAGAAGCTGGACTTCCCTATGGTGCTAACAAAGACGGCATCATATGGGTT
```

```
GCAACTGAGGGAGCCTTGAATACACCAAAAGATCACATTGGCACCCGCAATCCTGCTAACAATGCTGCAATCGTGCTACA
ACTTCCTCAAGGAACAACATTGCCAAAAGGCTTCTACGCAGAAGGGAGCAGAGGCGGCAGTCAAGCCTCTTCTCGTTCCT
CATCACGTAGTCGCAACAGTTCAAGAAATTCAACTCCAGGCAGCAGTAGGGGAACTTCTCCTGCTAGAATGGCTGGCAAT
GGCGGTGATGCTGCTCTTGCTTTGCTGCTGCTTGACAGATTGAACCAGCTTGAGAGCAAAATGTCTGGTAAAGGCCAACA
ACAACAAGGCCAAACTGTCACTAAGAAATCTGCTGCTGAGGCTTCTAAGAAGCCTCGGCAAAAACGTACTGCCACTAAAG
CATACAATGTAACACAAGCTTTCGGCAGACGTGGTCCAGAACAAACCCAAGGAAATTTTGGGGACCAGGAACTAATCAGA
CAAGGAACTGATTACAAACATTGGCCGCAAATTGCACAATTTGCCCCCAGCGCTTCAGCGTTCTTCGGAATGTCGCGCAT
TGGCATGGAAGTCACACCTTCGGGAACGTGGTTGACCTACACAGGTGCCATCAAATTGGATGACAAAGATCCAAATTTCA
AAGATCAAGTCATTTTGCTGAATAAGCATATTGACGCATACAAAACATTCCCACCAACAGAGCCTAAAAAGGACAAAAAG
AAGAAGGCTGATGAAACTCAAGCCTTACCGCAGAGACAGAAGAAACAGCAAACTGTGACTCTTCTTCCTGCTGCAGATTT
GGATGATTTCTCCAAACAATTGCAACAATCCATGAGCAGTGCTGACTCAACTCAGGCCTAAACTCATGCAGACCACACAA
GGCAGATGGGCTATATAAACGTTTTCGCTTTTCCGTTTACGATATATAGTCTACTCTTGTGCAGAATGAATTCTCGTAAC
TACATAGCACAAGTAGATGTAGTTAACTTTAATCTCACATAGCAATCTTTAATCAGTGTGTAACATTAGGGAGGACTTGA
AAGAGCCACCACATTTTCACCGAGGCCACGCGGAGTACGATCGAGTGTACAGTGAACAATGCTAGGGAGAGCTGCCTATA
TGGAAGAGCCCTAATGTGTAAAATTAATTTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGGAGAATGACAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
```

VI.3 Nucleotide frequencies

The **frequency** of a base or a nucleotide in a genetic code is the number of times it occurs divided by the length of the code. The varying frequency of different nucleotides, called the **nucleotide bias** varies between organisms and is known to have biological implications. Biologists also often talk of the *GC content*, the percentage of nitrogeneous bases (G and C) in an RNA or DNA to get insights into its stability.

The next task in this activity is to make a *python dictionary*, called `freq`, whose keys are the nucleotide characters and whose values are the number of times it occurs in the virus RNA. Once you have made it, `freq['A']`, for example, should output the frequency of nucleotide `A`.

```
[15]: freq = {b: rna.count(b)/len(rna)   for b in 'ATGC'}
```

```
[16]: freq
```

```
[16]: {'A': 0.29943483931378123,
       'T': 0.32083737417650404,
       'G': 0.19606728421897468,
       'C': 0.18366050229074005}
```

VI.4 A Washington sample

A more recent dataset at NCBI, apparently just submitted for peer-review on April 3, claims to contain the genome of a virus sample from our neighboring state of Washington. You can find it labeled there as the data set MT293201. Let us take a look. (Again, if the url below fails, please head over the NCBI webpage, find and download the corresponding data file for this sample, again in FASTA format, and save it using the name `f2` below.)

```
[17]: url2 = 'https://www.ncbi.nlm.nih.gov/sviewer/viewer.cgi?'          + \
          'tool=portal&save=file&log$=seqview&db=nuccore&report=fasta&' + \
          'id=1828694245&extrafeat=null&conwithfeat=on&hide-cdd=on'
      f2 = '../../data_external/SARS-CoV-2-Washington_MT293201.1.fasta'
```

```
[18]: r2 = urllib.request.urlopen(url2)
      fo2 = open(f2, 'wb')
      shutil.copyfileobj(r2, fo2)
```

You might have already heard in the news that there are multiple strains of the virus around the globe. Let's investigate this genetic code a bit closer.

Is this the same genetic code as from the Wuhan sample? Repeating the previous procedure on this new file, we now make a string object that contains the RNA from the Washington sample. We shall call it `rna2` below.

```
[19]: lines = open(f2, 'r').readlines()
      rna2 = ''.join([line.strip() for line in lines[1:]])
```

We should note that not all data sets uses just ATGC. There is a standard notation that extends the four letters, e.g., N is used to indicate *any* nucleotide. So, it might be a good idea to answer this question first: what are the distinct characters in the new `rna2`? There can be very simply done in python if you use the `set` data structure, which removes duplicates.

```
[20]: set(rna2)
```

```
[20]: {'A', 'C', 'G', 'T'}
```

The next natural question might be this. Are the lengths of `rna` and `rna2` the same?

```
[21]: len(rna2), len(rna)
```

```
[21]: (29846, 29903)
```

We could also look at the first and last 30 characters and check if they are the same, like so:

```
[22]: rna2[:30], rna2[-30:]
```

```
[22]: ('AACCTTTAAACTTTCGATCTCTTGTAGATC', 'TTTAATAGCTTCTTAGGAGAATGACAAAAA')
```

```
[23]: rna[:30], rna[-30:]
```

```
[23]: ('ATTAAAGGTTTATACCTTCCCAGGTAACAA', 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA')
```

Clearly, `rna` and `rna2` are different strings.

Compare their nucleotide frequencies

```
[24]: freq2 = {b: rna2.count(b)/len(rna2)   for b in 'ATGC'}
```

```
[25]: freq2
```

```
[25]: {'A': 0.29866648797158746,
       'T': 0.3214166052402332,
       'G': 0.1963077129263553,
       'C': 0.18360919386182403}
```

Although the Washington genome is not identical to the Wuhan one, their nucleotide frequencies are very close to the Wuhan one, reproduced here:

```
[26]: freq
```

```
[26]: {'A': 0.29943483931378123,
       'T': 0.32083737417650404,
       'G': 0.19606728421897468,
       'C': 0.18366050229074005}
```

Does it contain `ORF7a`?

```
[27]: s in rna2
```

```
[27]: True
```

```
[28]: rna2.find(s)
```

```
[28]: 27364
```

Thus, we located the same ORF7a instruction in this virus at a different location. Although the genetic code from the Washington sample and the Wuhan sample are different, they can make the same protein ORF7a and their nucleotide frequencies are very close.

This activity provided you with just a glimpse into the large field of bioinformatics, which studies, among other things, patterns of nucleotide arrangements. If you are interested in this field, you should take a look at Biopython, a bioinformatics python package.

VII Fibonacci primes

April 9, 2020

Fibonacci numbers appear in so many unexpected places that I am sure you have already seen them. They are elements of the Fibonacci sequence F_n defined by

$$\begin{aligned} &F_0 = 0, \qquad F_1 = 1, \\ &F_n = F_{n-1} + F_{n-2}, \qquad \text{for } n > 1. \end{aligned}$$

Obviously, this recursive formula gives infinitely many Fibonacci numbers. We also know that there are infinitely many prime numbers: the ancient Greeks knew it (actually proved it) in 300 BC!

But, to this day, we still do not know *if there are infinitely many prime numbers in the Fibonacci sequence.* These numbers form the set of **Fibonacci primes.** Its (in)finiteness is one of the still unsolved problems in mathematics.

In this activity, you will compute a few initial Fibonacci primes, while reviewing some python features along the way, such as generator expressions, `yield`, `next`, `all`, line magics, modules, and test functions. Packages we shall come across include `memory_profiler`, `primesieve`, and `pytest`.

VII.1 Generator expressions

Representing sequences is one of the elementary tasks any programming language should be able to do well. Python lists can certainly be used for this. For example, the following list comprehension gives elements of the sequence

$$n^i, \qquad n = 0, 1, 2, \ldots, N-1$$

succinctly:

[1]:
```
i=2; N=10

L = [n**i for n in range(1, N)]
```

If you *change the brackets to parentheses*, then instead of a list comprehension, you get a different object called *generator expression*.

[2]:
```
G = (n**i for n in range(1, N))
```

Both `L` and `G` are examples of **iterators**, an abstraction of a sequence of things with the ability to tell, given an element, what is the *next* element of the sequence. Since both `L` and

G are iterators, you will generally not see a difference in the results if you run a loop to print their values, or if you use them within a list comprehension.

```
[3]: [l for l in L]
```

```
[3]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[4]: [g for g in G]
```

```
[4]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

However, if you run the last statement again, what happens?

```
[5]: [g for g in G]
```

```
[5]: []
```

The difference between the generator expression G and the list L is that a generator expression does not actually compute the values until they are needed. Once an element of the sequence is computed, the next time, the generator can only compute the next element in the sequence. If the end of a finite sequence was already reached in a previous use of the generator, then there are no more elements of the sequence to compute. This is why we got the empty output above.

VII.2 Generator functions

Just as list comprehensions can be viewed as abbreviations of loops, generator expressions can also be viewed so using the `yield` statement. The statement

```
G = (n**i for n in range(1, N))
```

is an abbreviation of the following function with a loop where you find `yield` in the location where you might have expected a `return` statement.

```
[6]: def GG():
         for n in range(1, N):
             yield n**i
```

```
[7]: G2 = GG()
     print(*G2)  # see that you get the same values as before
```

```
1 4 9 16 25 36 49 64 81
```

The `yield` statement tells python that this function does not just return a value, but rather a value that is an element of a sequence, or an *iterator*. Internally, in order for something to be an iterator in python, it must have a well-defined `__next__()` method: even though you did not explicitly define anything called `__next__` when you defined `GG`, python seeing `yield` defines one for you behind the scenes.

Recall that you have seen another method whose name also began with two underscores, the special `__init__` method, which allows you to construct a object using the name of the class followed by parentheses. The `__next__` method is also a "special" method in that it allows you to call `next` on the iterator to get its next value, like so:

```
[8]: G2 = GG()

     # get the first 3 values of the sequence using next:

     next(G2), next(G2), next(G2)
```

```
[8]: (1, 4, 9)
```

```
[9]: print(*G2)    # print the remaining values of the sequence
```

```
16 25 36 49 64 81
```

As you can see, a generator "remembers" where it left off in a prior iteration.

VII.3 Disposable generators or reusable lists?

It might seem that generators are dangerous disposable objects that are somehow inferior to resuable lists which have all the same properties. Here is an example that checks that thinking:

```
[10]: i = -20
      N = 10**8
```

To compute the sum

$$\sum_{n=1}^{10^8} \frac{1}{n^{20}},$$

would you use the following list comprehension?

`sum([n**i for n in range(1, N)])`

If you do, you would need to store the created list in memory. If you install the `memory_profiler` and use it as described in the prerequisite reading material from [JV-H], then you can see memory usage easily. If you don't have a GB of RAM free, be warned that running this list comprehension (mentioned above, and in the cell after next) might crash your computer.

```
[11]: %load_ext memory_profiler
```

```
[12]: %memit sum([n**i for n in range(1, N)])
```

```
peak memory: 3884.82 MiB, increment: 3842.59 MiB
```

Per official standards, memory should be reported in mebibytes (MiB), a power of two that is close to 10^3 ("mebi" is made up of words "mega" and "binary"), although the commercial world continues to use 10-based MB, GB, etc. The "increment" reported in the above output in MiB is the difference between the peak memory and the memory used just before `memit` was called: that gives the memory used by the statement.

Clearly we should not need so much memory for such a simple task. A better solution is offered by the generator expression. It has no memory issues since it doesn't store all the elements of the sequence at once. Moreover, we can decide to stop iterating when the numbers in the sequence get below machine precision, thus getting to the sum faster.

[13]:
```
G3 = (n**i for n in range(1, N))

s = 0

for g in G3:
    s += g
    if g < 1e-15:
        break

print(s)
```

```
1.0000009539620338
```

VII.4 Infinite sequences

By now you are wondering, if we can work with a sequence of 10^8 entries, then why can we not work with an infinite sequence. Yes, python makes it easy for you to make an infinite sequence construct:

[14]:
```
def natural_numbers():
    n = 0
    while True:
        yield n
        n += 1
```

[15]:
```
for n in natural_numbers():
    print(n)
    if n >= 5:  break    # don't go into infinite loop!
```

```
0
1
2
3
4
5
```

In fact the function `count` in module `itertools` does just this. Python does assume that you are smart enough to use these without sending yourself into infinite loops. If you want to stay safe, then avoid using `while True`, replacing it with `while n < max` where `max` is some maximum number, a sentinel, that you never plan to exceed.

VII.5 Fibonacci generator

To generate F_n satisfying

$$F_0 = 0,\ F_1 = 1, \quad \forall n > 1 : F_n = F_{n-1} + F_{n-2},$$

we use a generator that keeps in memory two prior elements of the sequence, as follows.

[16]:
```
def fibonacci(max):
    f, fnext = 0, 1
    while f < max:
        yield f
        f, fnext = fnext, f + fnext
```

[17]:
```
Fn = fibonacci(10000)
print(*Fn)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

Note that we have used python's **tuple swap idiom** in the definition of `fibonacci` above. To understand it, note the evaluation order of expressions

```
expr3, expr4 = expr1, expr2
```

per the official documentation. The tuple swap idiom is an example (yet another) of how python achieves brevity without compromising elegance or readability.

VII.6 Prime number generator

Let's make a generator for the infinite prime number sequence. This classic example is beautifully discussed in [JV-H], which I suggest you read, if you have not already. Here is a standard method to generate the set P of all primes less than some N. Suppose at any stage of the generator algorithm, a subset $P = \{2, 3, \ldots, q\}$ of primes up to and including q have been found. The prime number generator should find the next prime number by checking if any element of P divides n for a number n greater than q: if the remainder in the division of n by p is nonzero for all $p \in P$, then n is the next prime.

For example, at some stage, suppose P is this:

[18]:
```
P = [2, 3]
```

Then, the next number $n = 4$ has remainders $4\%p$ given by

[19]:
```
[4 % p for p in P]
```

[19]:
```
[0, 1]
```

Clearly not `all` of the remainders are nonzero:

```
[20]: all([4 % p for p in P])
```

```
[20]: False
```

Hence the generator would conclude that the number 4 is not a prime, and proceed to the next case $n = 5$, which it would conclude is a prime because:

```
[21]: all([5 % p for p in P])
```

```
[21]: True
```

This is implemented below.

```
[22]: def prime_numbers(N):
          primes = []
          q = 1
          for n in range(q+1, N):
              if all(n % p > 0 for p in primes):
                  primes.append(n)
                  q = n
                  yield n
```

```
[23]: list(prime_numbers(70))
```

```
[23]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
```

VII.7 First few Fibonacci primes

Now we can generate all primes less than any number N and all Fibonacci numbers less than N. Listing Fibonacci primes less than N then becomes possible by simply intersecting the two sets. Python does have a `set` data structure which comes with a handy intersection method, so the code is trivial:

```
[24]: def fibonacci_primes(N):
          F = set(fibonacci(N))
          P = set(prime_numbers(N))
          print('Intersecting', len(P), 'primes with', len(F), 'fibonaccis.')
          return P.intersection(F)
      fibonacci_primes(100000)
```

```
Intersecting 9592 primes with 25 fibonaccis.
```

```
[24]: {2, 3, 5, 13, 89, 233, 1597, 28657}
```

VII.8 Verification

Verification refers to the process of cross checking that a program behaves as expected in a few chosen cases. It is the developer's responsibility to ensure that a code is correct.

Part of this responsibility involves designing and adding **test functions** that verify that the code produces the correct output in a few cases where correct output is known. Complex codebases with multiple files often have a suite of test functions. After a development team member changes one file, if the test suite does not pass all tests, it is a sign that the change has broken the code functionality due to some unanticipated repercussions of the change in other parts of the code.

How do you know that the result of your `fibonacci_primes` is correct? We could design checks, say by verifying that our prime number routine is correct for the first few primes, plus a similar check for Fibonacci numbers. Or, we could look up the known Fibonacci prime numbers and see if we have got the first few correctly. Design of such tests is the process of verification. While there no standard method for it, one often used principle is to *code with all cases in mind and test using known cases.*

Let us use the Online Encylopedia of Integer Sequences (OEIS) which reports the currently known *values of* n for which F_n is prime. It is listed there as sequence A001605. Here are the first 10 elements of this sequence:

[25]:
```
nFP = [3, 4, 5, 7, 11, 13, 17, 23, 29, 43]
```

Based on this we can write a *test function.* A test function has a name that begins with `test` and does not take any argument. In the test function below you see a statement of the form `assert Proposition, Error`, which will raise an `AssertionError` and print `Error` if `Proposition` evaluates to `False` (and if `True`, then assert lets execution continues to the next line). The test checks if our list of initial Fibonacci primes coincides with the one implied by `nFP` above.

[26]:
```
def test_fibonacci_prime():
    N = 10000
    F = list(fibonacci(N))
    nFP = [3, 4, 5, 7, 11, 13, 17, 23, 29, 43]

    our_list = fibonacci_primes(N)
    known_list = set([F[n] for n in nFP if n < len(F)])

    assert len(known_list.difference(our_list))==0, 'We have a bug!'
    print('Passed test!')
```

[27]:
```
test_fibonacci_prime()
```

```
Intersecting 1229 primes with 20 fibonaccis.
Passed test!
```

One of the python modules that facilitates automated testing in python codes is the `pytest` module. If you run `pytest` within a folder (directory), it will run all test functions it finds in all files of the form `test_*.py` or `*_test.py` in the current directory and its subdirectories. Please install `pytest` in the usual way you install any other python package.

To illustrate its use, let us make up a simple project structure as follows. This also serves as your introduction to **modules** in python. Please create a folder `fibonacci_primes` and files `my_simple_primes.py`, `fibonacci.py` and `test_fibonacci_primes.py` within the folder as shown below:

```
fibonacci_primes        <- create this folder within ../pyfiles
 |-- fibonacci.py         <- define functions fibonacci & fibonacci_primes
 |-- my_simple_primes.py     <- copy prime_numbers function definition here
 |-- test_fibonacci_primes.py   <- copy test_fibonacci_prime function here
```

Individual files may be thought of as python modules and you can import from them the way you have been importing from external packages. Since `fibonacci.py` uses `prime_numbers` which is now in a different file `my_simple_primes.py`, you should add the line

```
    from my_simple_primes import prime_numbers
```

at the top of `fibonacci.py`. Additionally, since `test_fibonacci_primes.py` uses the functions `fibonacci` and `fibonacci_primes`, in order for the test function to find these function definitions, you should include the line

```
    from fibonacci import fibonacci, fibonacci_primes
```

at the top of `test_fibonacci_primes.py`.

Now you have a project called `fibonacci_primes` on which you can apply automated testing programs like `pytest`, as shown in the next cell. Note that a test function *will run silently* if all tests pass (without printing any outputs).

[28]:
```
!pytest ../pyfiles/fibonacci_primes
```

```
================================ test session starts ================================
platform darwin -- Python 3.8.0, pytest-5.3.2, py-1.8.0, pluggy-0.13.1
rootdir: /Users/jay/Dropbox/Jay/teaching/2019-20/MTH271/mth271content
collected 1 item

../pyfiles/fibonacci_primes/test_fibonacci_prime.py .

[100%]

================================= 1 passed in 0.07s =================================
```

To see how the output would be like if a test fails, you might want to run this again after deliberately creating a bug: for example, set the initializer for Fibonacci recurrence to 2 instead of 1 in the file `fibonacci.py` and then return to run the above pytest to see what happens.

VII.9 There must be a module for it!

While coding up the prime number generator, did you get that nagging question in your mind, the one that we all get when coding up a basic algorithmic task in python? *May be there is already a module for this?*

Yes, indeed, the few lines we implemented above to get the prime numbers actually form an ancient algorithm, called the Sieve of Eratosthenes, which is implemented in many places. An example is a python binding for a C library called `primesieve`. (You might need to install python-primesieve and primesieve depending on your system.) After you install it, the following two lines will give you the same prime number list.

```
[29]: from primesieve import primes     # do after you have installed primesieve
      list(primes(70))
```

```
[29]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
```

This package is many times faster than our simple code with a python loop above, as you can verify using another `%magic`. (Recall that iPython/Jupyter facilties like `%timeit` and `%memit` are called line magics. You can read more about line and cell magics in your [JV-H].)

```
[30]: %timeit primes(1000)
```

```
3.15 µs ± 53.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
[31]: %timeit list(prime_numbers(1000))
```

```
1.2 ms ± 37 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

VII.10 The Fibonaccis among primes (or vice versa)?

Finding bigger and bigger primes is kind of like finding rare bit coins. Indeed, the difficulty of factoring the product of two large prime numbers is the basis of several encryption techniques. There is a world-wide effort to find more and more primes. For example, GIMPS, the Great Internet Mersenne Prime Search, discovered the largest (currently) known prime number, $2^{82,589,933} - 1$, having 24,862,048 digits (on December 7, 2018, using a computer in Ocala, Florida, whose owner decided to download the free GIMPS software to stress test his computer).

Our simple function `fibonacci_primes` was not designed to go above a finite maximal `N`, so of course, it can make no contribution to answering the unsolved question on finiteness of the set of Fibonacci primes. To write a code to find larger and larger Fibonacci primes, one might consider two options:

1. Look for prime numbers within the set of Fibonacci numbers.

OR

2. Look for Fibonacci numbers within the set of prime numbers.

The few Fibonacci numbers we saw above looked quite sparse so Option 1 might look good, but it would require us to test whether a number is prime or not, which as we saw involves quite a bit of effort as the numbers get larger.

Option 2 could work as a good strategy, especially when more and more primes are discovered, provided we know how to test if a number is in the Fibonacci sequence. Using some completely elementary mathematics (and without having to use any fancy theorems you haven't yet studied) you can prove the following. (Do be warned that proving this is not a 5-minute exercise; if you can do it in 5 minutes, I'd love to hear!)

Theorem 1. A number F is a Fibonacci number if and only if $5F^2 + 4$ or $5F^2 - 4$ is a perfect square.

Let's close by implementing this check for a number to be a perfect square using `math.sqrt` and let's use it together with `primesieve` to get a Fibonacci prime.

[32]:
```
import primesieve, math

def is_square(n):
    s = int(math.sqrt(n))
    return s*s == n

it = primesieve.Iterator()
it.skipto(2**28-1)
p = it.next_prime()

while p < 2**30-1:
    if is_square(5*p*p+4) or is_square(5*p*p-4):
        print('¡¡ Got one !! ', p, 'is a Fibonacci prime!')
    p = it.next_prime()
```

```
¡¡ Got one !!  433494437 is a Fibonacci prime!
```

Do feel free to experiment increasing the `2**30` limit above. But may be it is now time to manage your expectations a bit. A few decades ago, $2^{31} - 1$ was the largest integer representable on most computers. Now that 64-bit computing is common, we can go up to $2^{63} - 1$ (and a bit more with unsigned integers). To go beyond, not only will we need much faster programs, but also specialized software to represent larger integers and do arithmetic with them.

VIII Numpy blitz

April 14, 2020

Numpy arrays are more efficient than lists because all elements of numpy arrays are of the same pre-determined type. Numpy also provides efficient ufuncs (universal functions) which are vectorized functions that loop over array elements with loops pre-compiled in C. Numpy also exhibits some syntactic features that a mathematician may consider a nuisance, but knowing how to work with numpy is key for almost all scientific computation in python. It takes some practice to be comfortable with numpy and practice is what we are aiming for in this activity. This activity is structured as a list of questions. You will be in a position to appreciate these questions (and their answers) after going through this lecture's prerequistes on numpy yourself.

```
[1]: import numpy as np
     import math
```

VIII.0.1 Are lists and numpy arrays different?

```
[2]: A = [0.1, 1.3, 0.4, 0.5]      # list
     a = np.array(A)               # numpy array

     type(a), type(A)
```

```
[2]: (numpy.ndarray, list)
```

Here is how you find out the common data type of elements of a numpy array (and there is no such analogue for list, since list elements can be of different types).

```
[3]: a.dtype           # a's common element type (A.dtype is undefined!)
```

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

VIII.0.2 What is the difference between `2*a` and `2*A`?

```
[4]: 2*a
```

```
[4]: array([0.2, 2.6, 0.8, 1. ])
```

```
[5]: 2*A
```

```
[5]: [0.1, 1.3, 0.4, 0.5, 0.1, 1.3, 0.4, 0.5]
```

VIII.0.3 How best to compute $\sin(x)e^{-x}$ for many x?

Here is one option:

```
[sin(x[i]) * exp(-x[i])  for i in range(n)]
```

And here is another:

```
np.sin(x) * np.exp(-x)
```

Which is better?

```
[6]: n = 100000
     x = np.linspace(0, 2*np.pi, n)
```

```
[7]: # list comprehension
     %timeit y = [math.sin(x[i]) * math.exp(-x[i]) for i in range(n)]

     # use numpy ufuncs
     %timeit y = np.sin(x) * np.exp(-x)
```

```
64.7 ms ± 246 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
1.79 ms ± 202 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The functions `np.sin` and `np.exp` are examples of numpy's universal functions (ufuncs) that act directly on arrays. While `np.sin` are unary ufunc, there are many binary ufuncs like `np.add`: when you write `x+y`, you are actually calling the binary ufunc `np.add(x, y)`. Ufuncs are *vectorized* functions.

VIII.1 What is vectorization?

Vectorization refers to one or both of the following, depending on context:

1. A convenience feature: Apply an operation to all elements of a collection at once.
2. A performance feature: Use hardware instruction sets to execute single instruction on multiple data (SIMD).

A numpy ufunc like `np.sin` is vectorized in both the above senses: (1) you can apply it directly to an array thus avoiding python loops (unlike `math.sin` which can be applied to just a single value), and (2) numpy's C implementation of `np.sin` uses some SIMD instruction sets, allowing you to get automatic speed up when running on hardware that supports the instructions. If the latter still sounds mysterious, here is more explanation than you probably need: most chips today come with a SIMD instruction to process 4 numbers (float64) at once (and fancier chips can do more), so a loop over an array of N floats can finish in N/4 iterations if you utilize that instruction.

To examine the difference between the convenience feature and the performance feature, consider the following function, which is written to apply to just one number:

[8]:
```
def f(v):                       # apply f to one scalar value v
    return math.sin(v) * math.exp(-v)
```

To gain (1), the convenience feature, there are at least two options other than using ufuncs:

a) Use `map`

A function `f` acting on a scalar value can be made into a function that acts on a vector of values using the functional programming tool `map(f, x)`, which returns an iterator that applies f to every element of x.

[9]:
```
vectorizedf = map(f, x)         # apply same f to a vector of values x
```

b) Use numpy's `vectorize`

[10]:
```
F = np.vectorize(f)             # F can be applied to a array x
```

Both options (a) and (b) provide the convenience feature (1), letting you avoid python loops, and allows you to write expressive short codes.

However, neither option (a) nor (b) gives you the full performance of numpy's ufuncs.

[11]:
```
# use map
%timeit y = list(map(f, x))

# use numpy's vectorize
%timeit y = F(x)

# use numpy's ufunc
%timeit y = np.sin(x) * np.exp(-x)
```

```
48.2 ms ± 2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
36 ms ± 312 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
1.63 ms ± 9.14 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

VIII.1.1 Is `range` as efficient as `np.arange`?

[12]:
```
%timeit for x in range(1000000): x**3
%timeit for x in np.arange(1000000): x**3
```

```
356 ms ± 3.24 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
333 ms ± 14.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

There was a time (in Python 2) when `range` was not as efficient, but those times have passed.

VIII.1.2 Have you really understood indexing and slicing?

- A slice `a[b:e:s]` of `a` refers to the array of elements from the **b**eginning index `b` (included) till **e**nding index `e` (excluded), **s**tepping `s` elements.

- The defaults `b=0`, `e=len(a)`, and `s=1` may be omitted in a slice specification.
- Negative indices count from the end of the array: `a[-1]` is the last element of `a` and `a[-k] = a[len(a)-k]`.
- Positive indices count from the begining as usual.

[13]:
```
a = np.random.randint(0,9,5)
a
```

[13]:
```
array([2, 0, 5, 7, 3])
```

If you have understood these, then you should be able to say what the expected results are from each of the following statements.

```
a[::]
a[-1]
a[len(a)-1]
a[-3:]
a[-4:-1:2]
slice = range(-4,-1,2)
a[-4:-1:2], a[slice]
```

Verify your answers:

[14]:
```
a[::]
```

[14]:
```
array([2, 0, 5, 7, 3])
```

[15]:
```
a[-3:]
```

[15]:
```
array([5, 7, 3])
```

[16]:
```
a[-1], a[len(a)-1]
```

[16]:
```
(3, 3)
```

[17]:
```
a[-4:-1:2]
```

[17]:
```
array([0, 7])
```

[18]:
```
slice = range(-4,-1,2)  # Think of b:e:s specification as a range.
a[-4:-1:2], a[slice]     # In older versions, a[slice] may not work
                         # but will work with slice=arange(-4,-1,2).
```

[18]:
```
(array([0, 7]), array([0, 7]))
```

VIII.1.3 Do you really know what = does?

[19]:
```
a = np.array([1,2,3])
b = np.array([3,4,5,6])
```

After assigning a to b by =, what happens when you change an element of a?

[20]:
```
a = b
a[0] = 1
a
```

[20]:
```
array([1, 4, 5, 6])
```

We certainly expected the 3 to become 1 in a. Did you also expect the following?

[21]:
```
b
```

[21]:
```
array([1, 4, 5, 6])
```

If this surprises you, pay close attention to the next question.

VIII.1.4 What is a python variable anyway?

In most languages, each variable has its own memory address. For example consider this simple C++ code (ignore it if you don't know C++).

```
#include <vector>
std::vector<int> a{1,2,3}, b{3,4,5,6};
// Objects a and b each have their own memory addresses.
// Assignment a=b copies contents of b's memory into a.
a = b;
// a's memory address has not changed, but its contents have.
```

If you have programmed in C or C++, you might have gotten used to variables being permanently linked to their memory locations.

In contrast, python variables are just names. In python, variables like a and b are *names* which are not associated to fixed memory addresses. Names can be *bound* to one object in memory, and later to another. Multiple names can be bound to the same object (sometimes known as *aliasing* in other languages). The upshot of this is that in python, the assignment "=" changes names, but need not copy memory contents.

[22]:
```
a = np.array([1,2,3])       # This is Object1 and "a" is a name for it.
b = np.array([3,4,5,6])     # This is Object2 and "b" is a name for it.
```

We can double check that these objects occupy different memory locations using python's id function.

[23]:
```
id(a), id(b)
```

```
[23]: (4529008192, 4529242560)
```

Consider what happens when you say a=b:

```
[24]: a = b  # a is no longer a name for Object1, it is now a name for Object2.
```

```
[25]: id(a), id(b)
```

```
[25]: (4529242560, 4529242560)
```

Names a and b are now both bound to the same "Object2". (And we no longer have a way to access "Object1"!) Now, when we change a, or when we change b, we are really changing the same object.

VIII.1.5 What if I really want to copy data?

```
[26]: a = np.array([1,2,3])     # Object1
      b = np.array([3,4,5,6])   # Object2
      a = b.copy()              # Copies Object2, and binds a to the copy
      a[0] = 2                  # Only the copied (new) object is changed
```

```
[27]: a, b
```

```
[27]: (array([2, 4, 5, 6]), array([3, 4, 5, 6]))
```

VIII.1.6 Does numpy have matrices?

Of course, numpy is all about vectors and matrices (and even higher-order tensors). Two-dimensional data, or tabular data, or matrix data of the form

$$\begin{bmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

can be represented in python - either as list of lists - or as a numpy array.

The numpy array is more efficient than list of lists and has constructs for some matrix operations. (Note that you might find a numpy.matrix class, distinct from the array class, in some older codes, but be warned that it is deprecated. Due to problems arising from mixing matrix and array objects in python codes, we will *not* use the deprecated matrix class in this course. You should not use it in work you turn in.)

```
[28]: Amat = [[1,2],
              [3,4]]
      Amat
```

```
[28]: [[1, 2], [3, 4]]
```

```
[29]: amat = np.array(Amat)
      amat
```

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

```
[30]: type(A), type(a)
```

```
[30]: (list, numpy.ndarray)
```

Note that 2D and 1D numpy arrays are of the same type called `numpy.ndarray`.

VIII.1.7 Multiply a list or a matrix?

What is the difference between `2*Amat` and `2*amat`, for the objects `Amat` (list of lists) and `amat` (numpy array) just made above?

```
[31]: 2*Amat
```

```
[31]: [[1, 2], [3, 4], [1, 2], [3, 4]]
```

```
[32]: 2*amat
```

```
[32]: array([[2, 4],
             [6, 8]])
```

VIII.1.8 How do I matrix multiply?

```
[33]: amat
```

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

```
[34]: amat * amat
```

```
[34]: array([[ 1,  4],
             [ 9, 16]])
```

Look at the output: *is this really matrix multiplication?!* This is one thing that drives mathematicians crazy when they look at `numpy` for the first time. Mathematicians want the multiplication operator `*` to mean matrix multiplication when it is applied to numpy arrays. Unfortunately python's default `*` does **element-by-element multiplication**, not matrix multiplication. Since the first proposal for numpy, decades ago, many battles have been waged to resolve this embarrassment.

Finally, a few years ago, there came some good news. Since Python 3.5, the `@` symbol was dedicated to mean the matrix multiplication operator. You can read more about it at PEP 465.

```
[35]: import sys
      print(sys.version)   # check if you have version >= 3.5 before trying @
```

```
3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)]
```

```
[36]: amat @ amat
```

```
[36]: array([[ 7, 10],
             [15, 22]])
```

Naturally, many of us needed matrix multiplication before the `@` came along, so as you must have guessed, there is another way to do matrix multiplication:

```
[37]: np.dot(amat, amat)   # dot(A,B) = matrix A multiplied by matrix B
```

```
[37]: array([[ 7, 10],
             [15, 22]])
```

```
[38]: amat.dot(amat)
```

```
[38]: array([[ 7, 10],
             [15, 22]])
```

I think you will agree with me that this is not as neat as `@`.

You should know that the embarrassment continues in matrix powers. If you thought `amat ** 2` should give you a matrix power equaling the product of `amat` with itself, think again.

```
[39]: amat**2      # not equal to matrix power !!
```

```
[39]: array([[ 1,  4],
             [ 9, 16]])
```

Numpy provides a `matrix_power` routine to compute M^n for matrices M (and integers n).

```
[40]: np.linalg.matrix_power(amat, 2)
```

```
[40]: array([[ 7, 10],
             [15, 22]])
```

It does the job, but leaves elegance by the wayside.

VIII.1.9 How to slice 2D arrays?

Slicing in two-dimensional arrays is similar to slicing one-dimensional arrays. If `rslice` and `cslice` are 1D slices (or ranges) like the ones we used for one-dimensional arrays, then when applied to a 2D array `A`,

```
A[rslice, cslice]
```

the result is a submatrix of `A` using row indices in `rslice` and column indices in `cslice`.

```
[41]: A = np.array([[7, 8, 5, 1], [2, 5, 5, 2], [9, 6, 8, 9]])
      A
```

```
[41]: array([[7, 8, 5, 1],
             [2, 5, 5, 2],
             [9, 6, 8, 9]])
```

```
[42]: A[1, :], A[:, 2]
```

```
[42]: (array([2, 5, 5, 2]), array([5, 5, 8]))
```

```
[43]: A[:3:2, :3]
```

```
[43]: array([[7, 8, 5],
             [9, 6, 8]])
```

VIII.1.10 How are 2D arrays stored?

Like other programming facilities, numpy stores 2D array data internally as a 1D array, in order to get a contiguous memory block for convenient storage. For 2D arrays, Fortran and Matlab uses column-major ordering, while C uses row-major ordering. For example, the 2D array

```
[[7, 8, 5, 1],
 [2, 5, 5, 2],
 [9, 6, 8, 9]]
```

in *row-major ordering* looks like

```
7, 8, 5, 1, 2, 5, 5, 2, 9, 6, 8, 9
```

while in *column-major ordering*, it looks as follows.

```
7, 2, 9, 8, 5, 6, 5, 5, 8, 1, 2, 9
```

Numpy, by default, stores arrays in row-major ordering (like C). This thinking is reflected in some numpy's methods: e.g., when you ask numpy to reshape or flatten a array, the result is what you expect as if it were stored in row-major ordering.

```
[44]: M = np.array([[7, 8, 5, 1], [2, 5, 5, 2], [9, 6, 8, 9]])
      M
```

```
[44]: array([[7, 8, 5, 1],
             [2, 5, 5, 2],
             [9, 6, 8, 9]])
```

```
[45]: M.reshape(2, 6)    # Just a different view of the same data
```

```
[45]: array([[7, 8, 5, 1, 2, 5],
             [5, 2, 9, 6, 8, 9]])
```

```
[46]: M.ravel()          # The 1D data of M in row-major ordering
```

```
[46]: array([7, 8, 5, 1, 2, 5, 5, 2, 9, 6, 8, 9])
```

But the actual situation is more complicated since numpy allows users to *override the default storage* ordering. You can decide to store an array like in C or like in Fortran. Here is how to store the same array in Fortran's column-major ordering.

```
[47]: A = np.array(M, order='F')
      A
```

```
[47]: array([[7, 8, 5, 1],
             [2, 5, 5, 2],
             [9, 6, 8, 9]])
```

Obviously, it is the same matrix. How the data is stored internally is mostly immaterial (except for some performance optimizations). The behavior (of most) of numpy methods does not change even if the user has opted to store the array in a different ordering. If you really need to see a matrix's internal ordering, you can do so by calling the `ravel` method with keyword argument `order='A'`.

```
[48]: A.ravel(order='A')      # A's internal ordering is Fortran style
```

```
[48]: array([7, 2, 9, 8, 5, 6, 5, 5, 8, 1, 2, 9])
```

```
[49]: M.ravel(order='A')     # M's internal ordering is default C-style
```

```
[49]: array([7, 8, 5, 1, 2, 5, 5, 2, 9, 6, 8, 9])
```

VIII.1.11 Can I put booleans as indices?

If a numpy array is given indices that are boolean (instead of integers), then rows or columns are selected based on `True` indices. This is called **masking**. It is very useful together with vectorized conditionals.

```
[50]: N = np.arange(25).reshape(5,5)
      N
```

```
[50]: array([[ 0,  1,  2,  3,  4],
             [ 5,  6,  7,  8,  9],
             [10, 11, 12, 13, 14],
             [15, 16, 17, 18, 19],
             [20, 21, 22, 23, 24]])
```

How will you isolate elements in `N` whose value is between 7 and 18?

[51]:
```
mask = (N>7) & (N<18)
mask
```

[51]:
```
array([[False, False, False, False, False],
       [False, False, False,  True,  True],
       [ True,  True,  True,  True,  True],
       [ True,  True,  True, False, False],
       [False, False, False, False, False]])
```

These are the elements we needed:

[52]:
```
N[mask]
```

[52]:
```
array([ 8,  9, 10, 11, 12, 13, 14, 15, 16, 17])
```

And these are their locations:

[53]:
```
i, j = np.where(mask) # Returns i and j indices where mask[i,j] is True.
i, j                  # 1st True value of mask is at i[0],j[0],
                      # 2nd True value of mask is at i[1],j[1], etc.
```

[53]:
```
(array([1, 1, 2, 2, 2, 2, 2, 3, 3, 3]), array([3, 4, 0, 1, 2, 3, 4, 0, 1,␣
 ↪2]))
```

VIII.1.12 How do I represent higher order tensors?

Numpy can work with general-dimensional arrays, not just 1D or 2D arrays. For an n-dimensional array, the shape of a numpy array is a tuple of n integers giving the sizes in each dimension.

[54]:
```
data = np.random.randint(low=0, high=10, size=30)   # 1D array
```

[55]:
```
T2 = np.reshape(data, (6, 5))                       # 2D array
T2
```

[55]:
```
array([[3, 7, 2, 4, 2],
       [2, 6, 1, 1, 7],
       [0, 0, 5, 5, 9],
       [9, 1, 8, 7, 1],
       [2, 8, 8, 1, 3],
       [1, 6, 5, 3, 8]])
```

[56]:
```
T3 = np.reshape(data, (2, 3, 5))                   # 3D array
T3
```

[56]:
```
array([[[3, 7, 2, 4, 2],
        [2, 6, 1, 1, 7],
        [0, 0, 5, 5, 9]],
        [[9, 1, 8, 7, 1],
         [2, 8, 8, 1, 3],
         [1, 6, 5, 3, 8]]])
```

```
[57]: print('T3 is a ', T3.ndim, 'dimensional array of shape ', T3.shape)
      print('T2 is a ', T2.ndim, 'dimensional array of shape ', T2.shape)
      print('data is a ', data.ndim, 'dimensional array of shape ', data.shape)
```

```
T3 is a  3 dimensional array of shape  (2, 3, 5)
T2 is a  2 dimensional array of shape  (6, 5)
data is a  1 dimensional array of shape  (30,)
```

Here are a few other features of numpy arrays to note:

- Every numpy array has attributes `ndim` and `shape`.
- A scalar `c`, or `np.array(c)` is considered to have `ndim=0` and `shape=()`.
- A vector of length `n`, when viewed as a row vector has `ndim=1` and `shape=(n,)`.
- A vector of length `n`, when viewed as a column vector has `ndim=2` and `shape=(n, 1)`.
- You can convert a row vector `a` to a column vector by `a[:, np.newaxis]`.
- Use `newaxis` to add a new dimension, e.g., `T3[:, :, np.newaxis, :]` has `shape=(2, 3, 1, 5)`.

VIII.1.13 Would you like to add matrices of different shapes?

In mathematics, it would be an illegal operation to add matrices of different shapes. But it is not surprising that we would want to: e.g., viewing the number 10 as a 1x1 matrix and considering a matrix `A` of any other size, wouldn't it be nice to say `10 + A` to add 10 to all elements of A? Wouldn't it also be nice to be able to use + to add a vector to all columns of a matrix with more than one columns? All this and more is made possible in numpy by **broadcasting** rules, which extend the possibilities of vectorized operations. A very clear explanation of broadcasting is in [JV-H].

To see if you can add up (or apply another binary ufunc) differently shaped arrays, follow this algorithm, which uses the `ndim` and `shape` attributes we just saw.

Step 1. If two arrays differ in their `ndim`, then revise the `shape` of the one with lower `ndim` by prepending 1 on the left until the `ndims` are equal.

Step 2. If `shape[i]` (after its possible revision from Step 1) of the two arrays are unequal and one of them equals 1, then increase the latter to match the other `shape[i]`.

If the resulting revised `shapes` of the arrays are still unequal, then broadcasting fails and you can't add them. In Step 1, when we increase `ndim` by prepending a 1 to `shape`, we are not really changing the array: we are just imagining it with one extra dimension. In Step 2, when we increase `shape[i]` from 1 to something larger, we are imagining the array elements repeated along the `i`-th dimension (without actually performing an operation copying the elements). Here are a few examples to illustrate these rules.

Example 1:

```
 a + b =  [1, 8, 3]    +    [1]
```

```
          [0, 6, 5]         [8]
          a.ndim=2          b.ndim=2        <= No need for Step 1 modification
          a.shape=(2, 3)    b.shape=(2, 1)  <= Apply Step 2 to equalize shape

Step 2:   a.shape=(2, 3)    b.shape=(2, 3)

 a + b =  [1, 8, 3]    +    [1, 1, 1]     =  [ 2,  9,  4]
          [0, 6, 5]         [8, 8, 8]        [ 8, 14, 13]
```

Example 2:

```
 a + b =  [1, 8, 3]    +    [1]
          [0, 6, 5]
          a.ndim=2          b.ndim=0        <= Apply Step 1 to equalize ndim
          a.shape=(2, 3)    b.shape=()         (prepend 1 until ndim equalizes)

Step 1:   a.shape=(2, 3)    b.shape=(1, 1)  <= Next apply Step 2 to equalize shape

Step 2:   a.shape=(2, 3)    b.shape=(2, 3)

 a + b =  [1, 8, 3]    +    [1, 1, 1]     =  [ 2,  9,  4]
          [0, 6, 5]         [1, 1, 1]        [ 1,  7,  5]
```

Example 3:

```
 a + b =  [1, 8, 3]    +    [1, 3]
          [0, 6, 5]
          a.ndim=2          b.ndim=1        <= Apply Step 1 to equalize ndim
          a.shape=(2, 3)    b.shape=(2, )

Step 1:   a.shape=(2, 3)    b.shape=(1, 2)  <= Next apply Step 2 to equalize shape

Step 2:   a.shape=(2, 3)    b.shape=(2, 2)  <= Still unequal: broadcasting fails
```

As a simple exercise to further fix ideas, follow the above procedure and see if you can explain whether broadcasting rules apply, or not, to the following (with `T2` and `T3` as set previously).

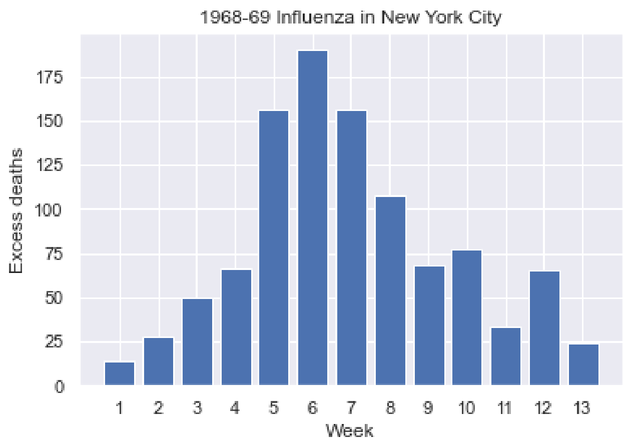
- `T2 + T3`
- `T3 + 1`
- `T2[:3,:] + T3`

IX
The SEIR model of infectious diseases

April 22, 2020

Recent news of COVID-19 has brought to our attention the stories of the many earlier pandemics the world has seen. A classic case is a strain of influenza that invaded New York City during 1968-1969, then dubbed the Hong Kong flu. The following data (from [DM]) shows the number of deaths that winter in New York City believed to be due to this flu.

[1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn; seaborn.set();
plt.bar(range(1,14), [14,28,50,66,156,190,156,108,68,77,33,65,24])
plt.xlabel('Week'); plt.ylabel('Excess deaths');
plt.xticks(range(1,14)); plt.title('1968-69 Influenza in New York City');
```



Notice how the data from Week 1 to Week 13 roughly fits into a bell-shaped curve. You have, by now, no doubt heard enough times that we all must do our part to *flatten the curve.* The bell-shaped curve, which has been identified in many disease progressions, is the curve we want to flatten. Some **mathematical models** of epidemic evolution, for instance the well-known "SIR model" discussed in [DM], produces such bell curves. Flattening the curve can then be interpreted as bringing relevant model parameters into a range that produces a shallow bell.

Mathematical models are often used as tools for prediction. However, we should be wary that models only approximate a few features of reality, and only when realistic parameter values (which are often missing) are supplied. Yet, as the saying goes, "All models are wrong, but some are useful." Even if a model is far away from the "truth", the "whole truth", it helps us understand the process being modeled by revealing the consequences of various hypotheses. Hence mathematical models are key instruments of computational thinking.

In this activity, we will study a mathematical model called the *SEIR model of infectious disease progression.* In the last few weeks, many researchers have been furiously working to fit the emerging COVID-19 data into variants of the SEIR model. A number of contributions can be viewed at the Bulletin of World Health Organization (WHO) which now maintains a special COVID-19 Open archive.

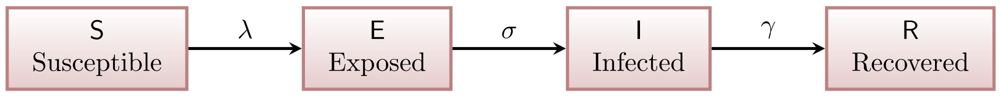
A number that emerges from models like the SIR or the SEIR model, called R_0, or the *basic reproduction number* often makes its appearance in popular science. It is even explained in a film from 2011 called the Contagion, which has now gained in popularity in view of its almost prescient plot. The *epidemiological definition of* R_0 is the average number of secondary cases produced by one infected individual introduced into a population entirely of susceptible individuals. One suspects from this definition that if $R_0 > 1$, then there will be an epidemic outbreak. We will see that this number also naturally emerges from a mathematical model. A quite readable review of R_0 (written before the COVID-19 pandemic) gives an R_0 of 14.5 for a measles outbreak in Ghana in the sixties. By all current accounts, the R_0 for COVID-19 appears to be between 2 and 3.

IX.1 Construction of the SEIR model

The SEIR model divides the population into four categories, called "S", "E", "I", and "R".

- Category "S" consists of individuals who are susceptible to the disease being modeled.
- Category "E" consists of individuals who are exposed to the disease. Diseases (like COVID-19) often have an incubation period or a latency period and this category accommodates it. (The SIR model does not have this category.)
- Category "I" consists of individuals infected with the disease and are capable of infecting others.

- Category "R" consists of individuals who can be removed from the system, e.g., because they have gained immunity to the disease, or because they have succumbed to the disease.



The model then postulates rules on how populations in each category can move to other categories. Let us consider the following simple set of rules.

- Assume that individuals move from S to E at the *exposure rate* λ, i.e., the population in category S *decreases* with respect to time t at the rate $\lambda \times S$ and the population in E correspondingly *increases* at the same rate:

$$\begin{aligned}\frac{dS}{dt} &= -\lambda S + \cdots \\ \frac{dE}{dt} &= +\lambda S + \cdots\end{aligned}$$

where "$\cdots$" serves to remind us that there may be other unmodeled factors. In this discussion, the *number* of individuals in each category (S, E, etc.) is denoted in italic type by the same letter (S, E etc.).

- The exposure rate λ should grow with I, the number of infected individuals. A standard hypothesis is that λ is the product of the **transmission rate** (or the *rate of contact*) β and the probability of infection given that contact occurred, which is I/N in a *total population of N individuals*, i.e.,

$$\lambda = \frac{\beta I}{N}.$$

- The **incubation rate** σ is the rate at which exposed hosts become infected, i.e.,

$$\begin{aligned}\frac{dE}{dt} &= +\lambda S - \sigma E + \cdots \\ \frac{dI}{dt} &= \qquad +\sigma E + \cdots\end{aligned}$$

- The **recovery rate** γ is the rate at which infected individuals move to the R category:

$$\begin{aligned}\frac{dI}{dt} &= +\sigma E - \gamma I + \cdots \\ \frac{dR}{dt} &= \qquad +\gamma I + \cdots\end{aligned}$$

Collecting the above-derived equations (and omitting the unknown/unmodeled "$\cdots$"), we have the following basic SEIR model system:

$$\begin{aligned}\frac{dS}{dt} &= -\frac{\beta I}{N} S, \\ \frac{dE}{dt} &= \frac{\beta I}{N} S - \sigma E, \\ \frac{dI}{dt} &= \sigma E - \gamma I \\ \frac{dR}{dt} &= \gamma I\end{aligned}$$

The three critical parameters in the model are β, σ, and γ.

Note that we have left several features *unmodeled*: exposed individuals in "E" might contribute to λ to spread the infection; some exposed individuals in "E" might move directly to the "R" category; some infected individuals in category "I" might not gain perfect immunity and so may move back to susceptible category "S." Despite these limitations, even this basic SEIR model can provide some useful insights on the disease evolution.

IX.2 Initial value problem

A system of ordinary differential equations (ODE) like the above, together with some initial conditions (values of the variables of the model at *initial* starting time, say $t = 0$), make up an *initial value problem*, or **IVP**. IVPs are ubiquitous in modeling systems that evolve in time. They encapsulate how a future state of a system is determined by the present state (the initial data) plus certain rules on how quantities evolve (the ODEs).

Before we talk about a python module to numerically solve an IVP, let us make a simplification. The total population N in the system (the sum of individuals in all categories) is likely to be a huge number. Instead of working with such large numbers, let us divide each side of each equation by N and work instead with the proportions

$$s = \frac{S}{N}, \quad e = \frac{E}{N}, \quad i = \frac{I}{N}, \quad r = \frac{R}{N}.$$

The equivalent ODE system to be solved for the unknown functions $s(t), e(t), i(t)$, and $r(t)$, has now become

$$\begin{aligned} \frac{ds}{dt} &= -\beta\, i\, s, \\ \frac{de}{dt} &= \beta\, i\, s - \sigma\, e, \\ \frac{di}{dt} &= \sigma\, e - \gamma\, i \\ \frac{dr}{dt} &= \gamma\, i. \end{aligned}$$

When supplemented with some initial conditions, say

$$s(0) = 0.99, \quad e(0) = 0.01, \quad i(0) = 0, \quad r(0) = 0,$$

we have completed our formulation of the IVP to be solved. Note that the above initial conditions correspond to a starting scenario where just 1% of the population is exposed.

IX.3 Solving the IVP using `scipy` module

[2]:
```python
from scipy.integrate import solve_ivp
import numpy as np
```

Scipy's `integrate` module provides a `solve_ivp` facility for solving IVPs like the above. The facility assumes you have an IVP of the form

$$\begin{aligned} \frac{d\vec{Y}}{dt} &= \vec{f}(t, \vec{Y}), & t_0 \le t \le t_1, \\ \vec{Y}(t_0) &= \vec{Y}_0, & t = t_0, \end{aligned}$$

where you know the function $\vec{f} : [t_0, t_1] \times \mathbb{R}^n \to \mathbb{R}^n$ and the initial data $\vec{Y}_0$. It can then compute an approximation of the solution $\vec{Y}(t)$ for t in the interval $[t_0, t_1]$ using numerical ODE solvers that you might learn about if you take a numerical analysis course. Type in `help(solve_ivp)` into a cell to get more information on how to use this function.

Let us apply this to the SEIR model. To fit to the setting required for `solve_ivp`, we put

$$\vec{Y} = \begin{bmatrix} s \\ e \\ i \\ r \end{bmatrix}$$

and

$$\vec{f}(t, \vec{Y}) = \begin{bmatrix} -\beta\, i\, s, \\ \beta\, i\, s - \sigma\, e, \\ \sigma\, e - \gamma\, i \\ \gamma\, i \end{bmatrix}.$$

We have to give this $\vec{f}$ as a function argument to `solve_ivp`. Let's first define this $\vec{f}$, called `seir_f` in the code below, keeping in mind that we also need to provide some values of β, σ, and γ before we can solve it. We pass these values as additional arguments to `seir_f`.

[3]:
```
def seir_f(t, y, beta, sigma, gamma):
    s, e, i, r = y
    return np.array([-beta * i * s,
                     -sigma * e + beta * i * s,
                     -gamma * i + sigma * e,
                     gamma * i])
```

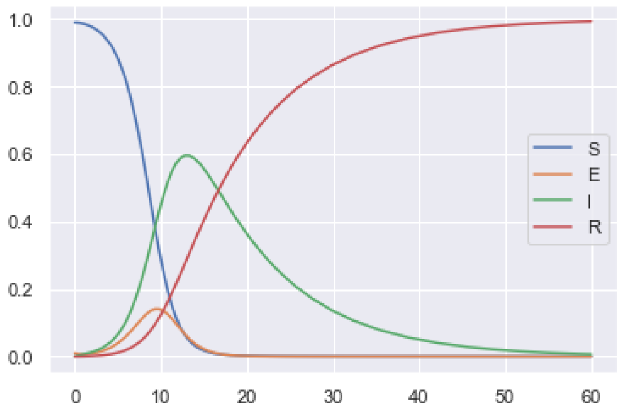
[4]:
```
# try some parameter values
beta = 1
sigma = 1
gamma = 0.1
```

Following the documentation from `help(solve_ivp)` we now proceed to solve by calling `solve_ivp` as follows.

[5]:
```
sol = solve_ivp(seir_f, [0, 60], [0.99, 0.01, 0, 0],
                rtol=1e-6, args=(beta, sigma, gamma))
```

Examining the resulting solution object `sol` you will notice that it has a numpy array as its data member `sol.y` containing the values of the computed solution $\vec{Y}(t)$ at values of `t` contained in another data member `sol.t`. We can easily send these arrays to the `matplotlib` module to get a plot of the solution.

[6]:
```
fig = plt.figure(); ax = fig.gca()
curves = ax.plot(sol.t, sol.y.T)
ax.legend(curves, ['S', 'E', 'I', 'R']);
```



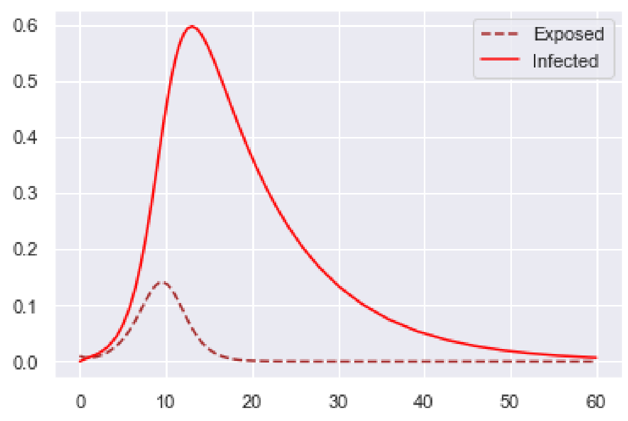
As you can see, even with 1% exposed population, the number of infections rapidly rise. However, with more time, they begin to fall, making for a bell-shaped curve, like the one from the previously mentioned New York City data.

IX.4 Parameter study

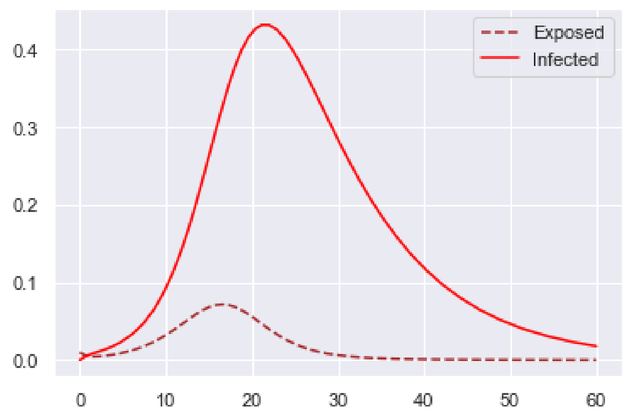
Having a function to compute and plot *E* and *I* together makes it easy to study the variations in solutions with respect to the three parameters. Let's make such a function by putting together the previous steps.

```
[7]: def plot_ei(beta=1, sigma=1, gamma=0.1, s0=0.99,
                 e0=0.01, i0=0, r0=0, t1=60):
         # apply ODE solver
         sol = solve_ivp(seir_f, [0, t1], [s0, e0, i0, r0], rtol=1e-7,
                         args=(beta, sigma, gamma))
         # plot I and E components
         fig = plt.figure(); ax = fig.gca()
         ax.plot(sol.t, sol.y[1, :].T, color='brown',
                 linestyle='dashed', label='Exposed')
         ax.plot(sol.t, sol.y[2, :].T, color='red', label='Infected')
         ax.legend()
         return ax
```

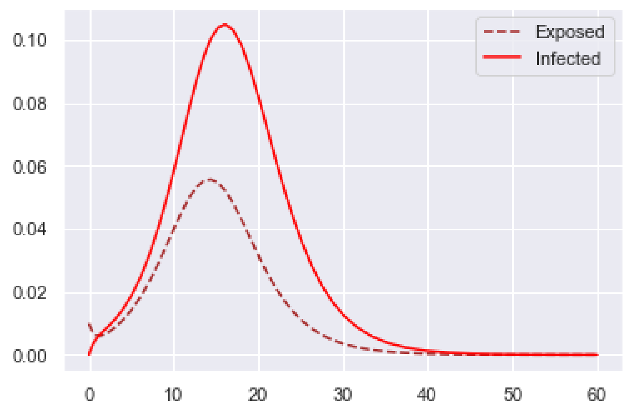
```
[8]: plot_ei();    # baseline with the default  parameters above
```



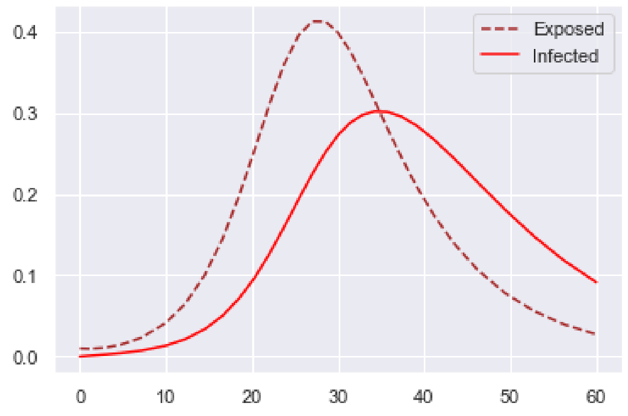
```
[9]: plot_ei(beta=0.5);     # what happens if beta is reduced?
```



```
[10]: plot_ei(gamma=0.5);    # what happens if gamma is increased?
```



```
[11]: plot_ei(sigma=0.1);    # what's the effect of sigma?
```



IX.5 Equilibria

In the study of evolution of dynamical systems like the SEIR model, equilibria play an important role. An *equilibrium state* is a value of the vector $\vec{Y}$ (i.e., values of s, e, i, and r) for which the rate of change $d\vec{Y}/dt = 0$, i.e., if the system happens to enter an exact equilibrium, then it no longer changes.

For our SEIR system, an equilibrium state s, e, i, r satisfies

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -\beta\, i\, s, \\ \beta\, i\, s - \sigma\, e, \\ \sigma\, e - \gamma\, i \\ \gamma\, i \end{bmatrix}.$$

You should be able to conclude (exercise!) that the only solutions for this system are of the form

$$s \equiv constant, \quad e = i = 0, \qquad r \equiv constant.$$

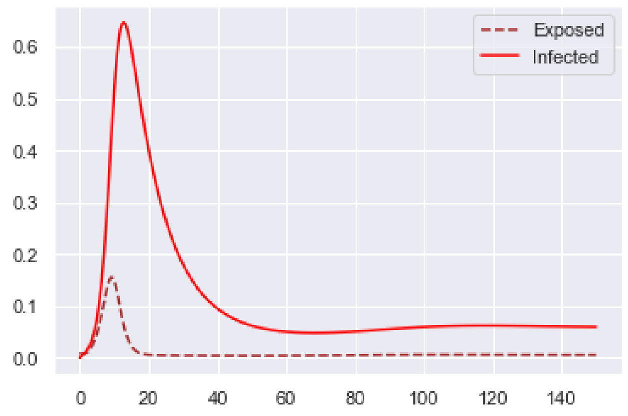
In other words, since $e = i = 0$, all equilibria of our model are **disease-free equilibria**. This matches our previous observations from our simulations. After a transitional phase, where i and e increases and decreases per the bell-curve, the system settles into an equilibrium of the form above.

There are other scenarios where an infection persists and never quite disappears from the population. Such equilibria where the disease is endemic are sometimes called **endemic equilibria**.

As an example, suppose our model represents a city's population, and suppose **travel into and out of the city is allowed**. Then we must add terms that represent the *influx* of travelers in each category (the number of people entering minus the number of people leaving). Even if we assume that infected people do not travel, a small influx into susceptible category S and exposed category E will disturb the disease-free equilibrium of our model. Let us add terms a and b representing these influxes and see what happens.

[12]:
```python
def seir_f2(t, y, beta, sigma, gamma, a, b):
    s, e, i, r = y
    return np.array([-beta * i * s + a,
                     -sigma * e + beta * i * s  + b,
                    -gamma * i + sigma * e,
                    gamma * i - (a + b)])

def plot_ei2(beta=1, sigma=1, gamma=0.1, a=0.005, b=0.001, t1=150):
    sol = solve_ivp(seir_f2, [0, t1], [0.99, 0.01, 0, 0], rtol=1e-7,
                    args=(beta, sigma, gamma, a, b))
    fig = plt.figure(); ax = fig.gca()
    ax.plot(sol.t, sol.y[1, :].T, color='brown', linestyle='dashed',␣
 ↪label='Exposed')
    ax.plot(sol.t, sol.y[2, :].T, color='red', label='Infected')
    ax.legend()
```

```
[13]: plot_ei2(a=0.005, b=0.001)
```



As you can see from this output, the percentage of the population with the disease now remains at around 5% and never quite vanishes, an example of an endemic equilibrium.

IX.6 The emergence of R_0

The stability of equilibria is another important consideration in the study of dynamical systems. Loosely speaking, an equilibrium is considered stable if a solution, when perturbed from the equilibrium, moves back to it over time. Returning to our simple model

$$\frac{d}{dt}\begin{bmatrix} s \\ e \\ i \\ r \end{bmatrix} = \begin{bmatrix} -\beta\, i\, s, \\ \beta\, i\, s - \sigma\, e, \\ \sigma\, e - \gamma\, i \\ \gamma\, i \end{bmatrix}$$

suppose we want to guess the stability of one of the previously discussed disease-free equilibrium states,

$$s = s_0, \quad e = i = 0, \quad r = r_0.$$

where s_0 and r_0 are some constants. Adding the e and the i equations, we observe that

$$\frac{d(e+i)}{dt} = (\beta\, s - \gamma)\, i.$$

Thus, despite a perturbation brought about by a small surge in the infected population (resulting in a small positive i value), if the above derivative is negative, i.e., if

$$\beta\, s_0 - \gamma < 0,$$

then, the value of $e + i$ will decrease to its equilibrium value. This simple argument already hints at the relevance of the number

$$R_0 = \frac{\beta}{\gamma} s_0,$$

which is the basic reproductive number for this model. In some texts, R_0 is defined (to match the epidemiological definition) using an initial population that is 100% susceptible, in which case $s_0 = 1$ and $R_0 = \beta/\gamma$.

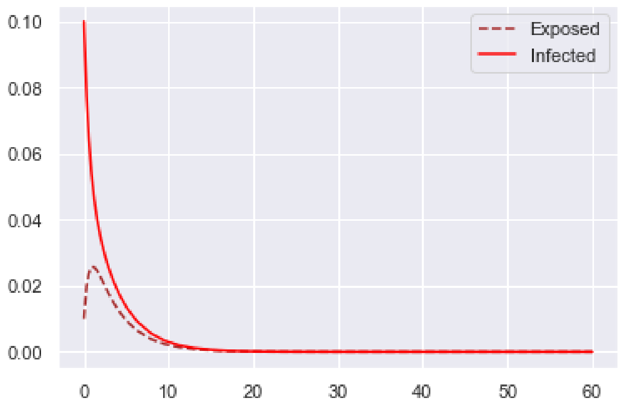
The argument sketched above was not a proof that the system will return to the disease-free equilibrium, but rather a sketch to show you why R_0 naturally emerges from the model, using only the calculus tools you have already studied. (Nonetheless, one can indeed rigorously prove that if $R_0 < 1$, the disease-free equilibrium is stable, see e.g., [HSW].)

IX.7 The effect of R_0: outbreak or no outbreak

The simple argument sketched above shows that if $R_0 = \beta s_0/\gamma > 1$ then $e + i$ will increase (at least for some time), while if $R_0 < 1$, then $e + i$ will decrease. Let us return to the code and examine the results from the model to see if there is agreement.

Here is an example where $R_0 = \beta s_0/\gamma < 1$.

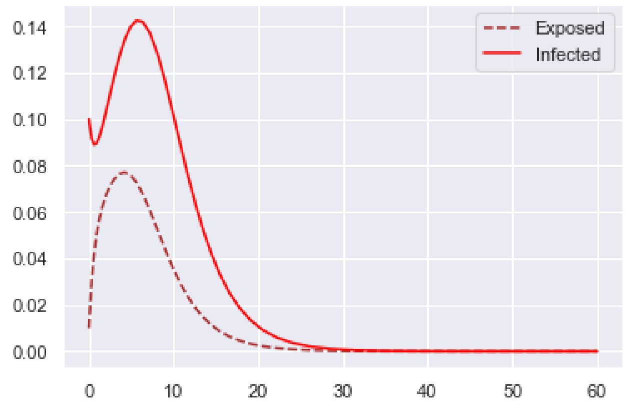
```
[14]: plot_ei(beta=0.6, gamma=1, s0=0.9, i0=0.1);
```



Clearly, the infected population, despite a positive bump in infections, decays as seen below. In other words, when $R_0 < 1$ *there is no outbreak*.

Next, consider an example where $R_0 > 1$.

```
[15]: plot_ei(beta=1, gamma=0.5, s0=0.9, i0=0.1);
```



This output clearly shows that the small percentage of introduced infections rapidly increase. (If you worry about the small initial dip in i observed in the output, do please try to plot $e + i$ to convince yourselves.) The system eventually goes on to attain (the unique) disease-free equilibrium, but only after inflicting some damage. Summarizing, *when* $R_0 > 1$, *we can expect an epidemic outbreak.*

Regarding the **impact of vaccinations** (provided a vaccine exists) the model does have something to say. If a fraction, say v, of the population is vaccinated, then that population moves directly from the S category to the R category. Therefore, changing s_0 to $s_0(1 - v)$, we see that R_0 reduces to $R_0 = \beta s_0(1 - v)/\gamma$. A vaccine would therefore be effective to prevent an epidemic outbreak if enough people are vaccinated, i.e., if v is sufficiently large in order to bring R_0 under 1.

IX.8 Application to COVID-19

There are a number of difficulties in applying the SEIR model to the COVID-19 epidemic we are now facing. One difficulty is in applying it to a single country: we would have to carefully develop terms that model inflow due to travel to or from the country. Of course, this problem disappears if we consider the entire world as our system. But other problems remain. As you must have heard in the news, we now suspect that exposed individuals in E category, who are not symptomatic, might be spreading the infection (i.e., λ might depend not only on I, but also on E). Our model does not take this into account. Although we can easily add terms to model this, without accurate testing of both symptomatic and asymptomatic populations, it is impossible to conclude the required parameter values. Notwithstanding these (significant) limitations, we can forge ahead to see what a simulation would give us, provided we can gather some data on the remaining parameter values.

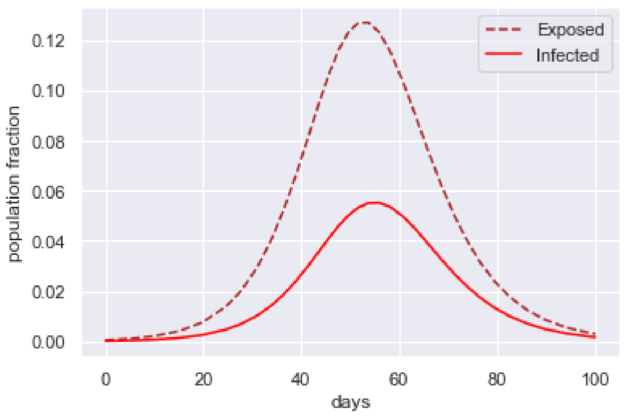
A recent submission to the Bulletin of WHO uses an R_0 of 2.2, which was reported in an earlier paper, published in January 2020, in the New England Journal of Medicine. Other reported values now found in the internet seem to be higher. (Inexact parameter values are indeed one of the difficulties in dealing with real-word problems.) Nonetheless, let us continue with $R_0 = \beta/\gamma = 2.2$. Let us also use the values of σ and γ that others have used:

- $\sigma = 1/5.2$ days^{-1},
- $\gamma = 1/2.3$ days^{-1},

- $\beta = R_0\gamma = 2.2\gamma$.

Finally, let us additionally assume a scenario where 0.02% of the world's population is infected at the start of the simulation, and thrice that many are exposed. (The current number of active infections worldwide appear to be around 0.02% of the world's population.) Here are the outputs of the simulation under these values.

```
[16]: ax = plot_ei(beta=2.2/2.3, sigma=1/5.2, gamma=1/2.3,
                   i0=0.02/100, e0=3*0.02/100, t1=100)
      ax.set_xlabel('days'); ax.set_ylabel('population fraction');
```



These output curves seem to suggest that the infection will proceed well into the next two months before subsiding.

The social distancing and other governmental measures that we are now practicing can be viewed from the perspective of this simple SEIR model. They are designed to reduce the transmission rate β. Please go ahead and experiment to see what you get with lower β values that you can imagine.

You will see that lowering β *by a little* has two effects:

- it reduces the peak value of the curves (multiply the percentage value by the world's population $\approx$ 7.5 billion, to see the effect in terms of the reduction in number of people affected), and

- it moves the location of the infection peak farther out in time (i.e., the infection persists longer but in lower numbers).

On the other hand, lowering β *by a lot* (enough to make $R_0 < 1$) will take you to a regime where $e + i$ decreases, indicating the other side of the peak, where we really want the world to be as soon as possible.

X
The Singular Value Decomposition

April 27, 2020

One of the early and natural ideas in software development for scientific computation was the idea of packaging linear algebra software around robust implementations of matrix factorizations, divested from specific applications. This enabled durable linear algebra tools like LAPACK to be developed with a stable interface, usable across numerous application domains. Commercial software like Matlab™, as well as open-source software like octave, numpy and scipy, all take full advantage of these developments. What does this mean for you as an aspiring data scientist? When you are faced with a specific computational task, if you are able to reformulate your task using off-the-shelf implementations of matrix factorizations, then you might already be half-way to finishing your task.

You already know about some matrix factorizations. In your introductory linear algebra prerequisites, you learnt about eigenvalues and eigenvectors. The computation of eigenvalues and eigenvectors is indeed the computation of a matrix factorization. This factorization is called a **diagonalization** or an **eigenvalue decomposition** of an $n \times n$ matrix A and it takes the form

$$A = XDX^{-1}$$

where D is a diagonal matrix and X is an invertible matrix, both of the same size as A. The eigenvalues of A are found in the diagonal entries of D and the eigenvectors are columns of X, as can be see by rewriting the factorization as

$$AX = XD.$$

The importance of eigenvalues in varied applications is usually highlighted well in a first linear algebra course.

Another important factorization is the SVD, or the **singular value decomposition**, which often does not get the emphasis it deserves in lower division courses. In some ways the SVD is even more important that a diagonalization. This is because not all matrices have a diagonalization. In contrast, using basic real analysis results, one can prove that any matrix has an SVD. We shall see that from an SVD, one can read off the important properties of a matrix and easily construct compressed approximations of the matrix. The two theorems stated below without proof are usually proved in a linear algebra course and can be found in many texts (see e.g., [TB]).

X.1 Definition of SVD

The SVD is a factorization of an $m \times n$ matrix A of the form

$$A = U\Sigma V^*$$

where Σ is am $m \times n$ diagonal matrix, and U and V are unitary matrices of sized $m \times m$ and $n \times n$, respectively. (Recall that a square matrix Q is called unitary if its inverse equals Q^*, the conjugate transpose of Q.) The diagonal entries of Σ are non-negative and positive ones are called the **singular values** of A. It is a convention to list the singular values in non-increasing order along the diagonal. The columns of U and V are called the left and right **singular vectors**, respectively.

Here is how we compute SVD using `scipy`.

```
[1]: from scipy.linalg import svd
     import numpy as np
     np.set_printoptions(precision=3, suppress=True)
```

```
[2]: a = np.random.rand(4, 5) + 1j * np.random.rand(4, 5)
     u, s, vh = svd(a)
```

```
[3]: u @ u.T.conjugate()    # u is unitary. Its columns are left singular␣
      ↪vectors
```

```
[3]: array([[1.+0.j, 0.-0.j, 0.-0.j, 0.+0.j],
            [0.+0.j, 1.+0.j, 0.-0.j, 0.+0.j],
            [0.+0.j, 0.+0.j, 1.+0.j, 0.-0.j],
            [0.-0.j, 0.-0.j, 0.+0.j, 1.+0.j]])
```

```
[4]: vh @ vh.T.conjugate() # Rows of vh are right singular vectors
```

```
[4]: array([[ 1.+0.j, -0.+0.j, -0.+0.j, -0.-0.j,  0.-0.j],
            [-0.-0.j,  1.+0.j,  0.-0.j, -0.-0.j,  0.+0.j],
            [-0.-0.j,  0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
            [-0.+0.j, -0.+0.j,  0.-0.j,  1.+0.j, -0.-0.j],
            [ 0.+0.j,  0.-0.j,  0.+0.j, -0.+0.j,  1.+0.j]])
```

```
[5]: s     # Only the diagonal entries of Sigma are returned in s
```

```
[5]: array([3.632, 1.088, 0.748, 0.473])
```

X.2 The algebra of SVD

An *outer product* of an $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$, is the $m \times n$ matrix xy^* (which being the product of $m \times 1$ and $1 \times n$ matrices, is of shape $m \times n$). Reversing the order of x and y^* in the product, we of course get the familiar *inner product*, which is a 1×1 matrix, or a scalar.

Although the outer product is an $m \times n$ matrix, with mn entries, it only takes $m + n$ entries to completely specify it (namely the entries of x vector and the y vector). Note that the columns of the outer product xy^* are

$$\bar{y}_1 x,\ \bar{y}_2 x, \ldots,\ \bar{y}_n x.$$

In other words all columns are scalar multiples of the same vector x. Therefore, whenever x is a nontrivial vector, the dimension of the *range* (or the *column space*) of the matrix is 1.

Recall from your linear algebra prerequisite that this dimension is what we call *rank*. All outer products are of unit rank (unless one of the vectors is trivial).

A very useful way to think of the SVD is to expand the factorization as follows. Naming the columns of U and V as u_i and v_j, we have

$$A = U\Sigma V^* = [u_1, \ldots, u_m] \begin{bmatrix} \sigma_1 & & \\ & \sigma_2 & \\ & & \ddots \end{bmatrix} [v_1, \ldots, v_n]^* = \sum_{l=1}^{\min(m,n)} \sigma_l \, u_l v_l^*.$$

Thus the SVD can be viewed as a sum of unit rank outer products. Each summand increases rank (if the corresponding singular value is nonzero) until the rank of A is reached. Let's see this in action for a small matrix.

```
[6]: a = np.random.rand(4, 5)
     u, s, vh = svd(a)
```

Numpy's broadcasting rules do not make it easy to make the outer product $u_l v_l^*$ simply. Yet, once you follow the broadcasting rules carefully, you will see that all that is needed is a placement of a `newaxis` in the right places.

```
[7]: u[0, :, np.newaxis] @ vh[np.newaxis, 0, :]
```

```
[7]: array([[ 0.249,  0.213,  0.163,  0.211,  0.214],
            [-0.205, -0.176, -0.134, -0.174, -0.177],
            [ 0.196,  0.168,  0.128,  0.167,  0.169],
            [ 0.365,  0.313,  0.238,  0.31 ,  0.314]])
```

Alternately, you can use the facility that numpy itself provides specifically for the outer product, namely `np.outer`.

```
[8]: np.outer(u[0, :], vh[0, :])
```

```
[8]: array([[ 0.249,  0.213,  0.163,  0.211,  0.214],
            [-0.205, -0.176, -0.134, -0.174, -0.177],
            [ 0.196,  0.168,  0.128,  0.167,  0.169],
            [ 0.365,  0.313,  0.238,  0.31 ,  0.314]])
```

Executing the sum $\sum_l \sigma_l (u_l v_l^*)$, we find that it is equal to a:

```
[9]: ar = np.zeros_like(a)
     for i in range(4):
         ar += np.outer(u[:, i], s[i] * vh[i, :])
```

```
[10]: a - ar    # a and ar are identical
```

```
[10]: array([[-0.,  0., -0., -0., -0.],
             [-0., -0., -0., -0., -0.],
             [-0., -0., -0., -0.,  0.],
             [-0., -0., -0.,  0., -0.]])
```

The factors of the SVD tell us all the important properties of the matrix immediately, as we see next. If you enjoy linear algebra, I encourage you to prove the following simple result yourself.

Theorem 1. Suppose $A = U\Sigma V^*$ is the SVD of an $m \times n$ matrix A. Then the following statements hold.

1. The rank r of A is the number of nonzero singular values.
2. A basis for the range (column space) of A is $\{u_1, u_2, \ldots, u_r\}$.
3. A basis for the null space (kernel) of A is $\{v_{r+1}, \ldots, v_{n-1}, v_n\}$.
4. The singular values of A are non-negative square roots of eigenvalues of A^*A.

Notice how the rank-nullity theorem (something you may have been tortured with in your first linear algebra course) follows as a trivial consequence of items (2) and (3).

X.3 The geometry of SVD

The geometry of any linear operator in $\mathbb{R}^n$ is easy to describe: application of a matrix transforms (hyper)spheres to (hyper)ellipses - if you did not know this, you will momentarily see this from the code below. Unitary operators are special in that they are coordinate changes that maps (hyper)spheres to (hyper)spheres. In general unitary operators don't change angles - they include operations like rotation and reflection in higher dimensions.

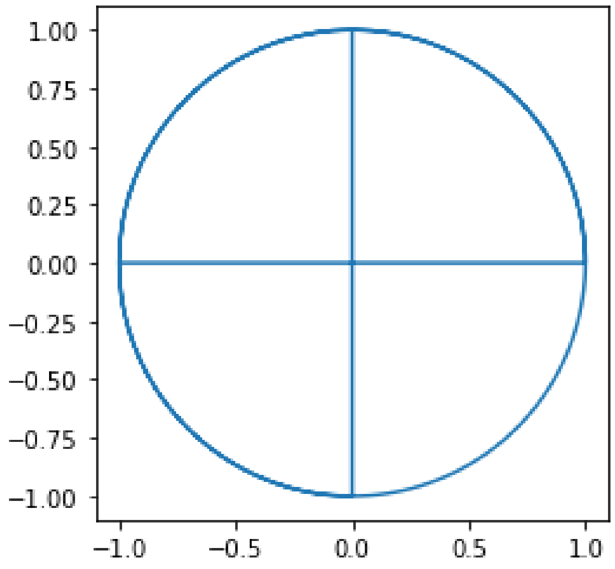
The presence of unitary factors in the SVD is significant. The SVD provides a geometrical decomposition of a linear operator into factors U and V^* that do not change the shapes, and a factor Σ that stretches axial directions (so that the shape change is transparent). Let us see this in action for a 2×2 matrix.

[11]:
```
a = np.array([[0.1, 0.5], [0.4, 0.8]])
u, s, vh = svd(a)
```

To see how the geometry gets transformed (squashed) by the linear operator (matrix) `a`, we first plot the unit circle and the parts of the x and y axis within it. Then, we track how these points are mapped by `a`, as well as by each of the components of the SVD of `a`.

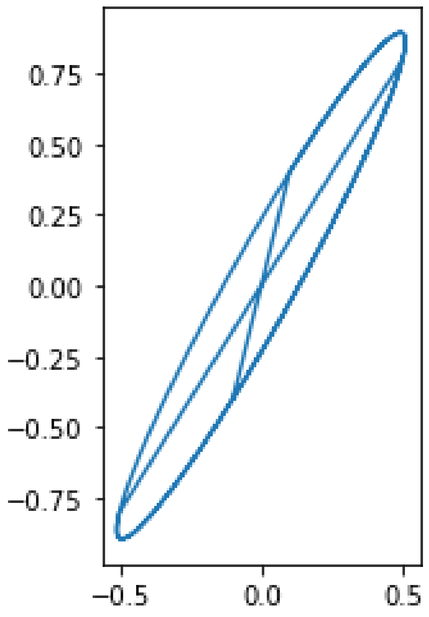
[12]:
```
import matplotlib.pyplot as plt
%matplotlib inline

def show(c):
    plt.plot(c[0, :], c[1, :])
    plt.axis('image');

# plot the unit circle and axis segments:
```

```
t = np.linspace(0, 3.5 * np.pi , num=300)
l = np.linspace(-1, 1, num=10)
z = np.zeros_like(l)
c = np.array([np.concatenate([l, np.cos(t), z]),
              np.concatenate([z, np.sin(t), l])])
show(c)
```



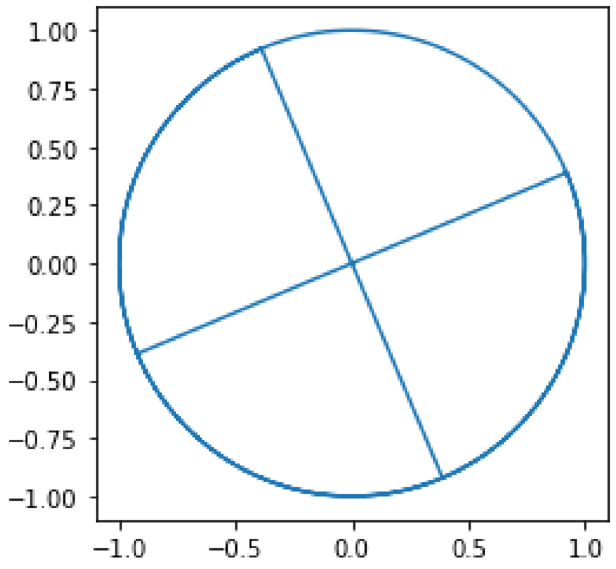
This is what a does to this geometry:

[13]:
```
show(a @ c)
```

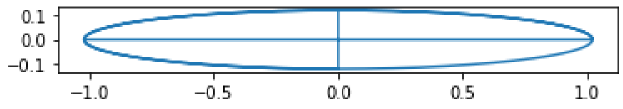


Now, let us see this transformation as a composition of the following three geometrical operations:

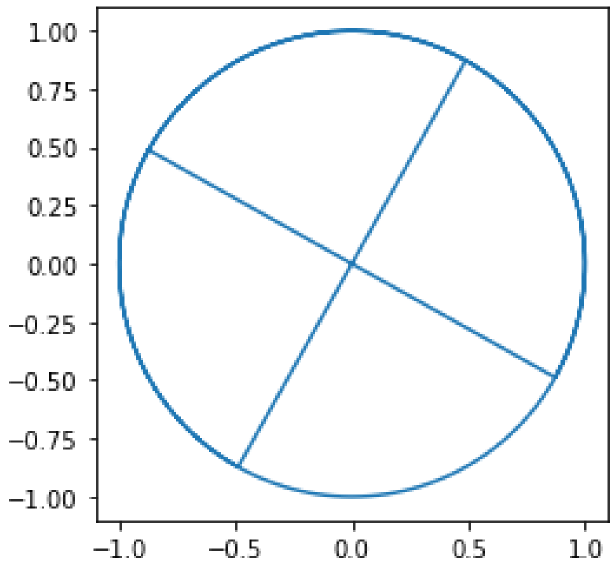
```
[14]: show(vh @ c)
```



```
[15]: show(np.diag(s) @ c)
```

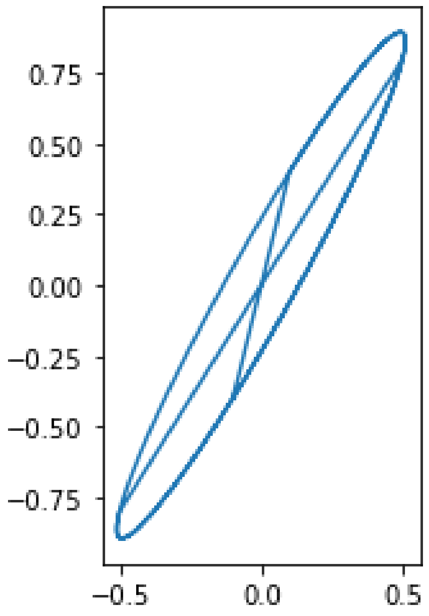


```
[16]: show(u @ c)
```



When you compose these operations, you indeed get the transformation generated by a.

```
[17]: show(u @ np.diag(s) @ vh @ c)
```



X.4 Low rank approximation

There are many ways of expressing a matrix as a sum of low rank matrices, e.g.,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & b \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ c & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & d \end{bmatrix}.$$

Each of the matrices on the right can have rank at most one.

As we have already seen, the SVD also expresses A as a sum of rank-one outer products. However, the way the SVD does this, is special in that a low-rank minimizer can be read off the SVD, as described in the following (Eckart-Young-Mirksy) theorem. Here we compare matrices of the same size using the *Frobenius* norm

$$\|A\|_F = \left(\sum_{i,j} |A_{ij}|^2 \right)^{1/2}.$$

The theorem answers the following question: how close can we get to A using matrices whose rank is much lower than the rank of A?

Theorem 2. Suppose A be an $m \times n$ matrix (complex or real). For any $0 \leq \ell \leq r = \text{rank}(A)$, define the matrix

$$A_\ell = \sum_{j=1}^{\ell} \sigma_j u_j v_j^*,$$

using the singular values σ_j and the left and right singular vectors u_j, v_j of A, i.e., A_ℓ is the sum of the first ℓ terms of the SVD when written as a sum of outer products. Then, the minimum of $\|A - B\|_F$ over all $m \times n$ matrices B of rank not more than ℓ is attained by $\|A - A_\ell\|_F$ and the minimum is $(\sigma_{\ell+1}^2 + \cdots + \sigma_r^2)^{1/2}$.

This matrix approximation result is perhaps best visualized using images. Think of an image as a matrix. Using `matplotlib`'s `imread` we can load PNG images. Here is an example.

[18]:
```
cats = plt.imread('../figs/GeoLea.png')
cats.shape
```

[18]:
```
(1040, 758, 4)
```

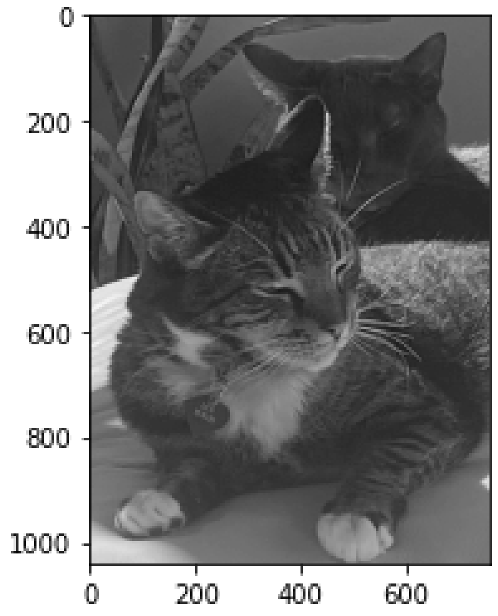
This is a 3-dimensional array because images are represented using RGBA values at each pixel (red, green, blue and an alpha value for transparency). However this image of my cats is actually a black and white image, so all RGB values are the same, as can be verified immediately:

[19]:
```
np.linalg.norm(cats[..., 0] - cats[..., 2], 'fro')
```

[19]:
```
0.0
```

The above line contains two features that you might want to note: the use of the ellipsis to leave the dimension of a numpy slice unspecified, and the way to compute the Frobenius norm in numpy. Restricting to the first of the three identical image channels, we continue:

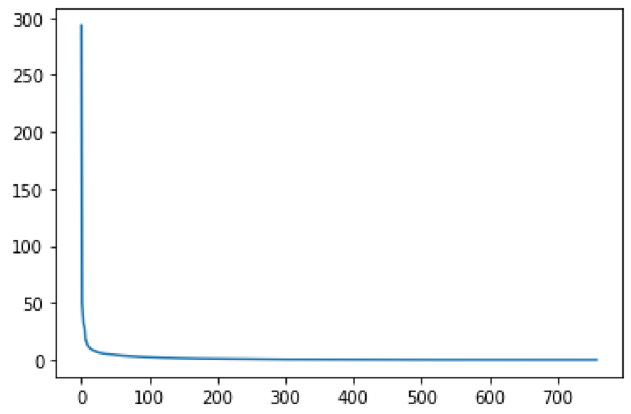
[20]:
```
c = cats[..., 0]
plt.imshow(c, cmap='gray');
```



Let us take the SVD of this 1040 x 758 matrix.

```
[21]: u, s, vh = svd(c)
```

```
[22]: plt.plot(s);
```

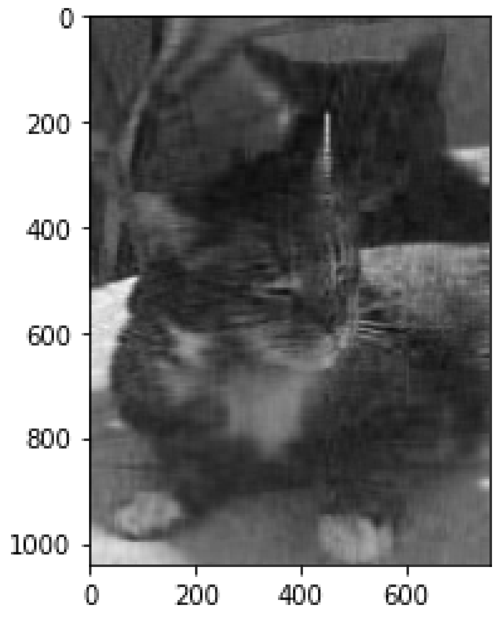


You can see a sharp drop in the magnitude of the singular values. This is a good indication that the later summands in the SVD representation of A,

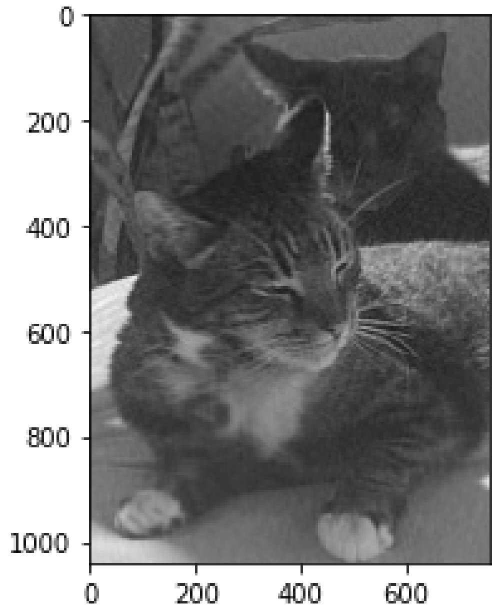
$$A = \sum_{j=1}^{\min(m,n)} \sigma_j u_j v_j^*$$

are adding much less to A than the first few summands. Therefore, we should be able to represent the same A using the first few outer products.

```
[23]: # Rank 20 approximation of the cats:
l = 20;   cl = u[:, :l] @ np.diag(s[:l]) @ vh[:l, :]
plt.imshow(cl, cmap='gray');
```



[24]:
```
# Rank 50 approximation of the cats:
l = 50;   cl = u[:, :l] @ np.diag(s[:l]) @ vh[:l, :]
plt.imshow(cl, cmap='gray');
```



If you increase the rank `l` to 100, you will find that the result is visually indistinguishable from the original.

Returning to Theorem 2, notice that the theorem also gives one the ability to specify an error tolerance and let that dictate the choice of the rank ℓ. E.g., if I do not want the error in my low-rank approximation to be more than some specific ε, then I need only choose ℓ

so that

$$(\sigma_{\ell+1}^2 + \cdots + \sigma_r^2)^{1/2} \le \varepsilon.$$

As an example, suppose I declare I want a low-rank approximation within the following relative error in Frobenius norm:

```
[25]: relative_error = 1.e-1
```

Then we can find the needed ℓ using an *aggregation* and *masking* (see [JV-H] for the prerequisite material on this) as follows.

```
[26]: s2 = s**2
      total = np.sum(s2)
      diff = np.sqrt((total - np.add.accumulate(s2)) / total)
      l = np.argmax(diff < relative_error) + 1
      l
```

```
[26]: 41
```

Then here is the needed low rank approximation:

```
[27]: cl = u[:, :l] @ np.diag(s[:l]) @ vh[:l, :]
```

You can check that the error is indeed less than the prescribed error tolerance.

```
[28]: np.linalg.norm(c - cl, 'fro') / np.linalg.norm(c, 'fro')
```

```
[28]: 0.09942439
```

As you can see, the low rank approximation does give some image compression. The number of entries needed to store a rank ℓ approximation `cl` of an $m \times n$ matrix is $m\ell + \ell + \ell n$:

```
[29]: u.shape[0] * l + l + l * vh.shape[0]
```

```
[29]: 73759
```

In contrast, to store the original image (single channel) we would need to minimally store mn numbers:

```
[30]: c.shape[0] * c.shape[1]
```

```
[30]: 788320
```

Comparing the two previous output, we can certainly conclude that we have some compression. However, for image compression, there are better algorithms.

The utility of the low-rank approximation technique using the SVD is not really in image compression, but rather in other applications needing a condensed operator. Being an compressed approximation of an *operator* (i.e., a matrix) is much more than being just a compressed visual image. For example, one can not only reduce the storage for a matrix, but also reduce the time to apply the operator to a vector using a low-rank approximation. Because the SVD tells us the essentials of an operator in lower dimensional spaces, it continues to find new applications in many modern emerging concepts, some of which we shall see in later lectures.

XI
Bikes on Tilikum Crossing

May 1, 2020

A car-free bridge is still considered a ridiculous idea in many parts of our country. Portlanders beg to differ. Portland's newest bridge, the *Tilikum Crossing*, opened in 2015, and is highly multimodal, allowing travel for pedestrians, bikes, electric scooters, trains, streetcars, and buses (but the modality of travel by personal car is missing). Bike lanes were not an afterthought, but rather an integral part of the bridge design. One therefore expects to see a good amount of bike traffic on Tilikum.

In this activity, we examine the data collected by the bicycle counters on the Tilikum. Portland is divided into east side and west side by the north-flowing Willamette river and the Tilikum connects the two sides with both eastbound and westbound lanes. Here is a photo of the bike counter (the black display, located in between the streetcar and the bike lane) on the bridge.

Portlanders use the numbers displayed live on this little device to boast about Portland's bike scene in comparison to other cities. The data from the device can also be used in more complex ways. The goal of this lesson is to share the excitement of extracting knowledge or information from data - it is more fun than a Sherlock Holmes tale. In this activity, you get to be Mr. Holmes while you wrangle with the data and feel the thrill of uncovering the following facts that even many of the locals don't know about. (a) Most of those who bike to work on Tilikum live on the east side. (b) Recreational bikers on Tilikum prefer afternoon rides. (c) There are fewer bikers on the bridge after social distancing and they appear to use the bridge during afternoons.

Comparison with Seattle's Fremont bridge bike counter data reveals more, as we shall see: (a) there are fewer bikers on Portland's Tilikum than on Seattle's Fremont bridge in general. (b) During peak hours, bikers are distributed more evenly on Seattle's Fremont bridge travel lanes than on Tilikum. (c) The bike usage on both bridges have shifted to a recreational pattern after social distancing.

The BikePed Portal provides some of the data collected from the counter for the public, but currently only subsampled data can be downloaded from there. Here we shall instead use the full raw data set collected by the counters, which is not yet publicly downloadable. I gratefully acknowledge Dr. Tammy Lee and TREC for making this data accessible. This activity is motivated by the material in Working with Time Series section of [JV-H].

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
```

XI.1 Initial examination of the data

As you have seen in previous activities, the first step in dealing with real data is data wrangling to make the data fit our tools. The case of this data is no different. (If you haven't yet heard of Hadley Wickham's famous paper Tidy Data, *J. Stat. Software*, I recommend you take a look. It begins with the sentence, "A huge amount of effort is spent cleaning data to get it ready for analysis ...")

[2]:
```
# metadata file (small file)
tm = pd.read_csv('../../data_external/tilikum_metadata.csv')

# data file (large file)
td = pd.read_csv('../../data_external/tilikum_20200501.csv')
td.head()
```

[2]:
```
         id              start_time                end_time measure_period␣
 ↪ \
0  36586735  2015-08-09 08:00:00+00  2015-08-09 08:15:00+00        00:15:00
1  36586736  2015-08-09 08:15:00+00  2015-08-09 08:30:00+00        00:15:00
2  36586737  2015-08-09 08:30:00+00  2015-08-09 08:45:00+00        00:15:00
3  36586738  2015-08-09 08:45:00+00  2015-08-09 09:00:00+00        00:15:00
4  36586739  2015-08-09 09:00:00+00  2015-08-09 09:15:00+00        00:15:00
```

```
   volume  flow_detector_id
0       0              1903
1       0              1903
2       0              1903
3       0              1903
4       0              1903
```

[3]:
```
td.tail()
```

[3]:
```
              id                start_time                  end_time  \
324700  96966870  2020-04-30 05:45:00+00  2020-04-30 06:00:00+00
324701  96966871  2020-04-30 06:00:00+00  2020-04-30 06:15:00+00
324702  96966872  2020-04-30 06:15:00+00  2020-04-30 06:30:00+00
324703  96966873  2020-04-30 06:30:00+00  2020-04-30 06:45:00+00
324704  96966874  2020-04-30 06:45:00+00  2020-04-30 07:00:00+00

       measure_period  volume  flow_detector_id
324700       00:15:00       4              1905
324701       00:15:00       0              1905
324702       00:15:00       0              1905
324703       00:15:00       0              1905
324704       00:15:00       0              1905
```

Looking through first few (of the over 300,000) data entries above, and then examining the meta data file contents in `tm`, we conclude that `volume` gives the bike counts. The volume is for 15-minute intervals as seen from `measure_period`. A quick check indicates that every data entry has a starting and ending time that conforms to a 15-minute measurement.

[4]:
```
dif = pd.to_datetime(td['end_time']) - pd.to_datetime(td['start_time'])
(dif == dif[0]).all()
```

[4]:
```
True
```

Therefore, let us rename `start_time` to just `time` and drop the redundant data in `end_time` and `measure_period` (as well as the `id`) columns.

[5]:
```
td = td.rename(columns={'start_time':'time'}).drop(columns=['end_time',␣
 ↪'measure_period', 'id'])
```

The meta data also tells us to expect three detectors and three values of `flow_detector_id`. Here are a few entries from the meta data:

[6]:
```
tm.T.loc[['detector_description', 'flow_detector_id', 'detector_make',␣
 ↪'detector_name', 'facility_description'], :]
```

[6]:
```
                                                        0  \
detector_description                  Inbound towards East
flow_detector_id                                      1903
```

```
detector_make                                                  EcoCounter
detector_name                                        Tilikum Crossing 1 EB
facility_description  South bike lane of Tilikum Crossing Bridge

                                                                        1  \
detector_description                                Inbound towards West
flow_detector_id                                                     1904
detector_make                                                  EcoCounter
detector_name                                      Tilikum Crossing (EAST)
facility_description  North bike lane of Tilikum Crossing Bridge

                                                                        2
detector_description                                Inbound towards West
flow_detector_id                                                     1905
detector_make                                                  EcoCounter
detector_name                                        Tilikum Crossing 2 WB
facility_description  North bike lane of Tilikum Crossing Bridge
```

Although there are three values of `flow_detector_id` listed above, one of these values never seems to appear in the data file. You can check that it does not as follows:

```
[7]: (td.flow_detector_id==1904).sum()
```

```
[7]: 0
```

Therefore, going through the meta data again, we conclude that eastbound and westbound bikes pass through the flow detectors with id-numbers 1903 and 1905, respectively.

The next step is to reshape the data into the form of a time series. The `start_time` seems like a good candidate for indexing a time series. But it's a red herring. A closer look will tell you that the times are repeated in the data set. This is because there are distinct data entries for the eastbound and westbound volumes with the same time stamp. So we will make two data sets (since our data is not gigabytes long, memory will not be an issue), a `tE` for eastbound volume and `tW` for westbound volume.

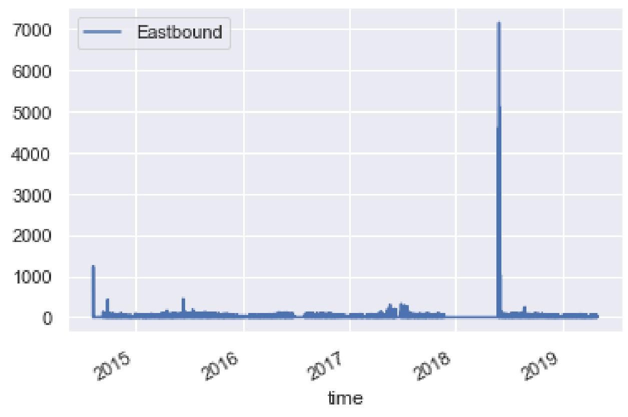
```
[8]: tE = td.loc[td['flow_detector_id']==1903, ['time', 'volume']]
     tE.index = pd.DatetimeIndex(pd.to_datetime(tE['time'])).tz_convert('US/
      ↪Pacific')
     tE = tE.drop(columns=['time']).rename(columns={'volume':'Eastbound'})
```

```
[9]: tW = td.loc[td['flow_detector_id']==1905, ['time', 'volume']]
     tW.index = pd.DatetimeIndex(pd.to_datetime(tW['time'])).tz_convert('US/
      ↪Pacific')
     tW = tW.drop(columns=['time']).rename(columns={'volume':'Westbound'})
```

Note that we have now indexed eastbound and westbound data by time stamps, and renamed `volume` to `Eastbound` and `Westbound` respectively in each case.

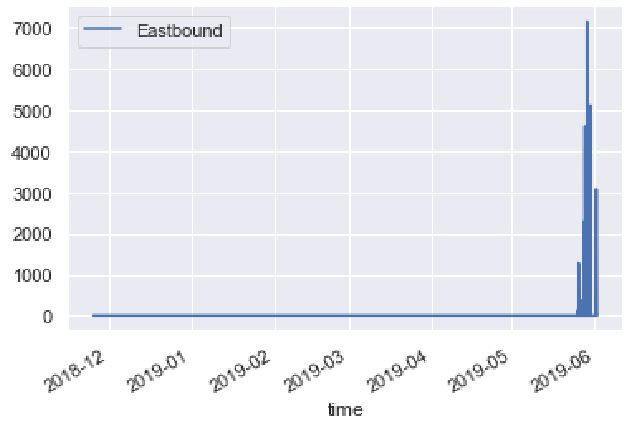
We are now ready for a first look at the full time series. Let us consider the eastbound data first.

```
[10]: tE.plot();
```



Clearly, we have problems with this data. A spike of 7000 bikers passing through in 15 minutes, even for a bike-crazed city like Portland, just does not seem right. Zooming in, we find the situation even more disturbing, with a lot of zero readings before the spike:

```
[11]: tE['2018-11-25':'2019-06-01'].plot();
```



There are reports from TriMet of construction in 2018 and city traffic advisories in 2019 that might all affect bike counter operation, but since the data set seems to have no means to indicate these outages, we are forced to come up with some strategy ourselves for discarding the false-looking entries from the data.

First, exploiting `pandas`' ability to work with missing values, we declare the entries for the dates in the above plot to be missing. Note that missing data is not the same as zero data. When the bike counter is not working, the data should ideally be marked as missing, not zero. Since our suspicion is that outages might have resulted in defective counts, we shall effectively remove all data entries for these dates from the data set, as follows:

```
[12]: tE['2018-11-25':'2019-06-01'] = np.nan
```

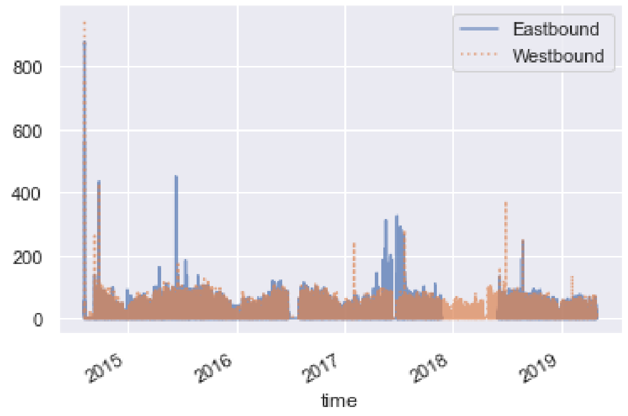
Next, we shall declare all entries with a volume of more than 1000 bikes per 15 minute to be a missing/defective value on both the westbound and eastbound data.

[13]:
```
tE[tE > 1000] = np.nan
tW[tW > 1000] = np.nan
```

XI.2 Visualize cleaned up data

After the preparations above, we are now ready to visualize. Let us merge the east and west two data sets on the same time stamp axis.

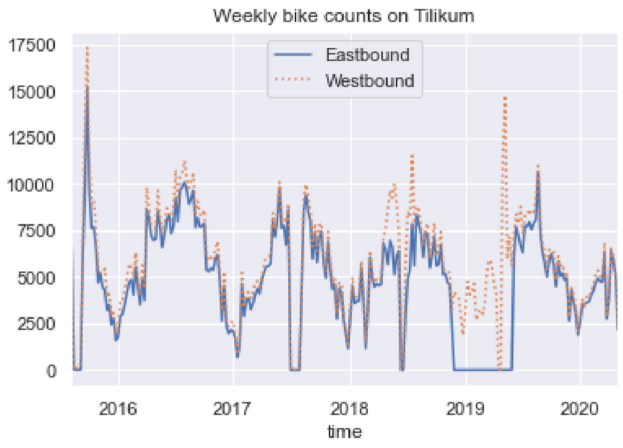
[14]:
```
t = pd.merge(tE, tW, on='time')
t.plot(alpha=0.7, style=['-',':']);
```



Examining the above graph, we still see spikes that look unreasonably high in the beginning of the data, but they may actually be real because at the official opening of the bridge there were 30,000 to 40,000 people and at least 13,000 bikes milling around. Similarly, the other spikes may be real data. One can try to explain them, e.g., by consulting https://bikeportland.org/events/, from which you might conclude that the spike on August 25, 2019 is due to a Green Loop event, and that large spike on June 29, 2019 might be due to all the people coming over for the World Naked Bike Ride; or was it some afterparty of Loud'n Lit event? I can't really tell. We'll just leave it at that, and blame the remaining spikes on the groovy bike scene of Portland.

The quarter-hour samples look too dense in the plot above. A better picture of the situation is obtained by extracting weekly counts of bikes in both directions from the data.

[15]:
```
t.resample('W').sum().plot(style=['-',':'], title='Weekly bike counts on␣
↪Tilikum');
```



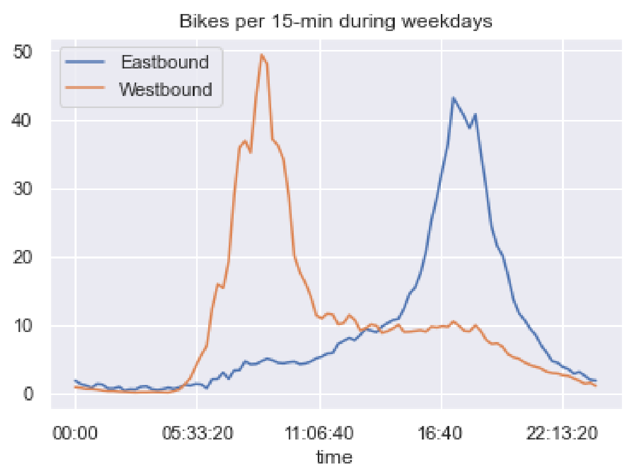
XI.3 The pattern of use

The Tilikum is being used both by people who commute to work using a bicycle as well as recreational bicycle users. We can understand more about this division among bikers by dividing the data into weekend and weekday entries.

The only technical skills you need for this are `numpy.where` and an understanding of `pandas.Timestamp` objects. (Please ensure you have studied Working with Time Series section of [JV-H] before proceeding.) Combined with a use of `pandas.groupby`, we can then extract the mean biker volumes for each 15-minute interval during the day.

The result is the distribution plotted below.

[16]:
```python
def weekplot(d, onlyweekend=False, title=None):
    weekend = np.where(d.index.weekday < 5, 'Weekday', 'Weekend')
    by_time = d.groupby([weekend, d.index.time]).mean()
    if onlyweekend:
        if title is None: title = 'Bikes per 15-min during weekends'
        by_time.loc['Weekend'].plot(title=title)
    else:
        if title is None: title = 'Bikes per 15-min during weekdays'
        by_time.loc['Weekday'].plot(title=title)

weekplot(t)
```

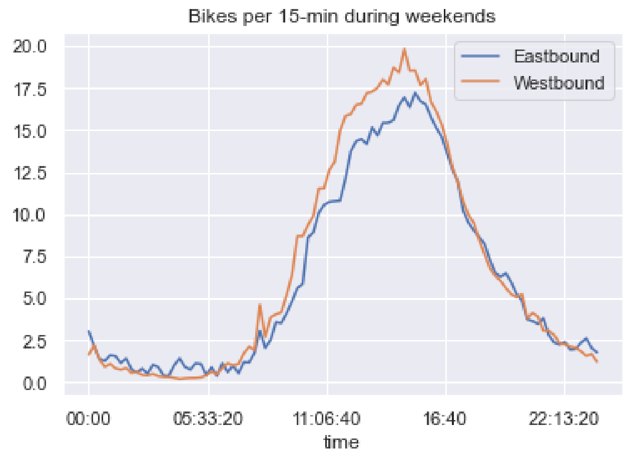


The hourly distribution is distinctly “bimodal”. There is a group of westbound commuters (on their way to work) on the bridge in the morning, and a group (probably the same people) traveling eastbound after work. If you look closely, you will find that there are two slightly smaller bumps indicating that there are some (although many fewer) eastbound morning bikers and westbound evening bikers across the bridge. Yet, on the whole, the data leads us to the interesting conclusion that the overwhelming majority of the **bike commuters on the Tilikum live on the east side** and commute to the west for work and return daily.

Often the purpose of understanding data is to guide policy and action. What might one do with the pattern we have just discovered? The current numbers are small enough not to pose a bike traffic problem. But envision a future where the bike counts will grow. If it grows maintaining the same lop-sided utilization pattern, what are the city’s options to encourage optimal bridge usage? Bike traffic flow control modifications? Generation of more jobs on the east side? More residential zoning near the west end of the bridge? These are complex issues where an urban planner’s expertise is needed. Nonetheless, I hope to have convinced you of the importance of going from data (clicks on a counter) to knowledge (patterns of use).

Next, let us look at the non-commuter, recreational, use, assuming that they occur in the weekends. In sharp contrast to the weekday distribution, below we find that the weekend distribution has just one peak.

[17]:
```
weekplot(t, onlyweekend=True)
```

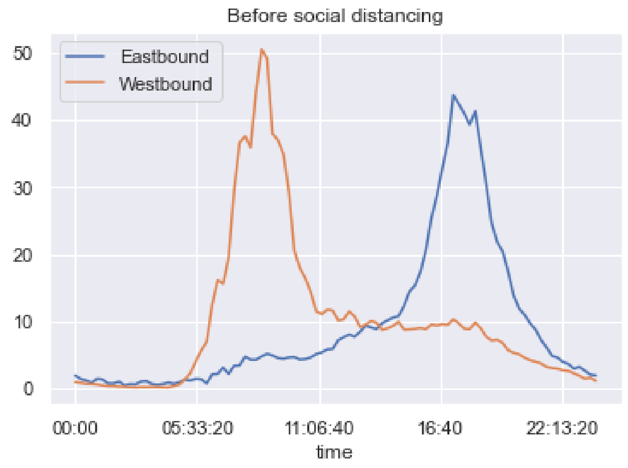


Both the eastbound and westbound lanes seem to find a good amount of use in the weekend. There is, most definitely, a preference for recreational riding in the afternoon. I suppose that is not a major surprise in Portland as afternoons are most often when we are given a reprieve from the battleship gray of the cloud cover.

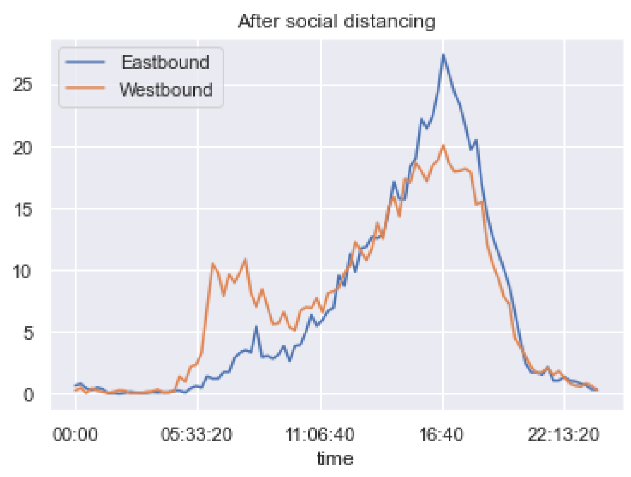
XI.4 Changes due to isolation

As you know, on March 18, 2020, in-person instructional activities at all universities in Oregon were suspended, and on March 23 our governor issued the "Stay Home, Save Lives" executive order. Since the Tilikum is near two major universities in Portland, we expect the weekday bike traffic to be impacted by these measures. Let us examine what the data tells us.

[18]:
```
weekplot(t.loc[:'2020-03-17'], title='Before social distancing')
```



[19]:
```
weekplot(t.loc['2020-03-17':], title='After social distancing')
```



Clearly, the strong bimodal distribution has weakened considerably after we all started isolating ourselves. This perhaps comes as no surprise, since both universities on the west side of Tilikum have switched to remote classes. It makes sense that there are fewer westbound commuters in the morning. What about the afternoon peak? One could imagine various explanations for this: people isolating themselves all morning, getting restless in the afternoon, especially with such unusually good weather we were having in April, and deciding to take their bikes out for some fresh air. Whatever be the case, we can summarize our conclusion from the data as follows: **social distancing has changed the weekday bike use on Tilikum from a commuter to a recreational pattern.**

Of course, we can also compare the overall statistics before and after social distancing, but the results are too blunt to point out differences like the above. From the statistics outputs below, we see that the average number of bikers per quarter-hour in each direction has decreased by about 1:

```
[20]: t.loc[:'2020-03-17'].mean() - t.loc['2020-03-17':].mean()
```

```
[20]: Eastbound    0.868757
      Westbound    1.247903
      dtype: float64
```

The data can also tell us the reduction in terms of number of bikers per week, although we should perhaps use it with some caution as not enough weeks have passed after social distancing started to form a robust sample.

```
[21]: t.loc[:'2020-03-17'].resample('W').sum().mean() - t.loc['2020-03-17':].
       ↪resample('W').sum().mean()
```

```
[21]: Eastbound     208.005903
      Westbound    1088.181228
      dtype: float64
```

The westbound lane certainly seems to have suffered more reduction in traffic after social distancing, whichever way we slice it.

XI.5 Comparison with Seattle's Fremont bridge

Although Portland claims to be the first city in the US to adopt the open data program, Seattle's open data program is something to envy. Seattle's Fremont bridge bike counter data, even way back from 2012, is readily available for anyone to download, thanks to their open data program (at the URL below). Let's take a peek at their data.

[22]:
```
import os
import shutil
import urllib

url = "https://data.seattle.gov/api/views/65db-xm6k/rows.csv?
 ↪accessType=DOWNLOAD"
f = "../../data_external/Fremont_Bridge_Bicycle_Counter.csv"

if not os.path.isdir('../../data_external/'):
    os.mkdir('../../data_external/')

if not os.path.exists(f):
    with open(f, 'wb') as fo:
        r = urllib.request.urlopen(url)
        shutil.copyfileobj(r, fo)
```

[23]:
```
sd = pd.read_csv(f)
sd.tail()
```

[23]:
```
                         Date  Fremont Bridge Total  \
66403  04/30/2020 07:00:00 PM                 156.0
66404  04/30/2020 08:00:00 PM                  51.0
66405  04/30/2020 09:00:00 PM                  25.0
66406  04/30/2020 10:00:00 PM                  15.0
66407  04/30/2020 11:00:00 PM                  13.0

       Fremont Bridge East Sidewalk  Fremont Bridge West Sidewalk
66403                          68.0                          88.0
66404                          30.0                          21.0
66405                          17.0                           8.0
66406                           4.0                          11.0
66407                           6.0                           7.0
```

Let's do some quick clean up and renaming.

[24]:
```
sd = sd.rename(columns={'Date' : 'time',
                        'Fremont Bridge East Sidewalk' : 'East',
                        'Fremont Bridge West Sidewalk' : 'West'})
sd.index = pd.to_datetime(sd.loc[:, 'time'])
sd = sd.drop(columns=['time', 'Fremont Bridge Total'])
sd.head()
```

```
[24]:                      East  West
      time
      2012-10-03 00:00:00   4.0   9.0
      2012-10-03 01:00:00   4.0   6.0
      2012-10-03 02:00:00   1.0   1.0
      2012-10-03 03:00:00   2.0   3.0
      2012-10-03 04:00:00   6.0   1.0
```

XI.5.1 Volume comparison

Note that the Seattle data gives counts per hour, not counts per 15-minutes like the Tilikum data. To compare the general statistics, we should resample the Tilikum to get hourly counts.

```
[25]: th = t.resample('H').sum()
      th.describe()    # Portland's Tilikum
```

```
[25]:           Eastbound      Westbound
      count  41423.000000  41423.000000
      mean      28.467856     34.979504
      std       53.384867     55.594878
      min        0.000000      0.000000
      25%        0.000000      2.000000
      50%        7.000000     14.000000
      75%       32.000000     46.000000
      max     2606.000000   1577.000000
```

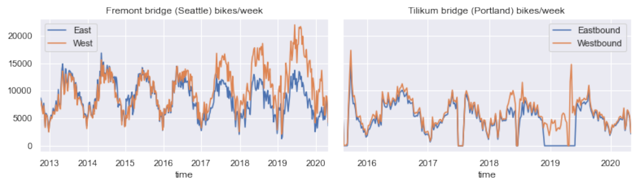
```
[26]: sd.describe()    # Seattle's Fremont
```

```
[26]:                East          West
      count  66398.000000  66398.000000
      mean      51.653047     61.499277
      std       66.661856     90.060985
      min        0.000000      0.000000
      25%        6.000000      7.000000
      50%       28.000000     30.000000
      75%       69.000000     74.000000
      max      698.000000    850.000000
```

The Tilikum data is spikier than Seattle's Fremont data (compare the `max` values in the above outputs), but the average volumes (`mean`) are clearly higher in Seattle. That the volume is higher in Seattle in even more clear if we plot weekly counts on both bridges on the same axis.

```
[27]: sw = sd.resample('W').sum()
      tw = t.resample('W').sum()
      fig, axs = plt.subplots(1, 2, figsize=(13, 3), sharey=True)
      plt.subplots_adjust(wspace=0.05)
```

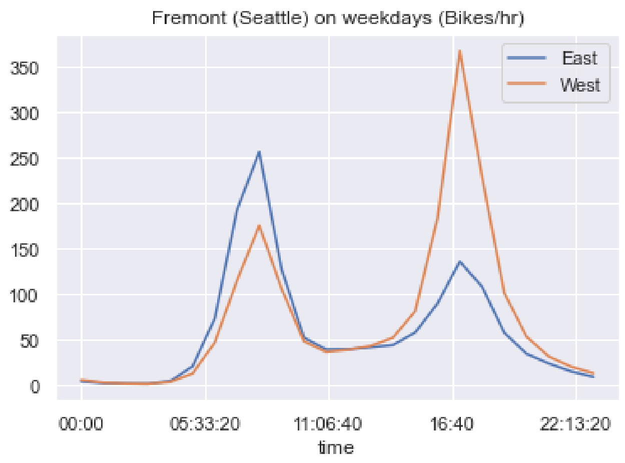
```
sw.plot(ax=axs[0], title='Fremont bridge (Seattle) bikes/week');
tw.plot(ax=axs[1], title='Tilikum bridge (Portland) bikes/week');
```

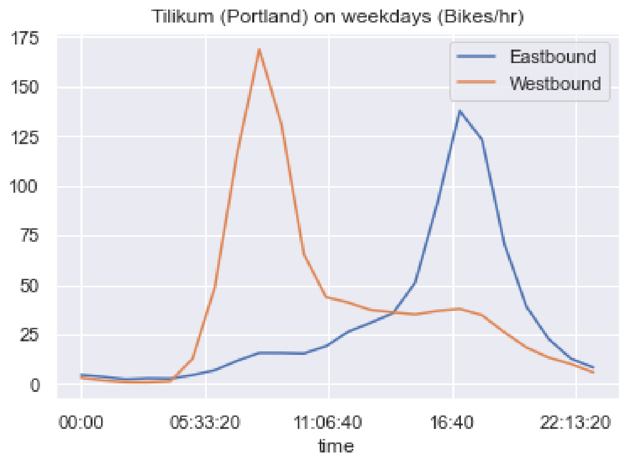


XI.5.2 Daily patterns

There is a striking difference in the distribution of the average number of bikes/hour during weekdays on the two bridges.

[28]:
```
weekplot(sd, title='Fremont (Seattle) on weekdays (Bikes/hr)')
weekplot(th, title='Tilikum (Portland) on weekdays (Bikes/hr)')
```

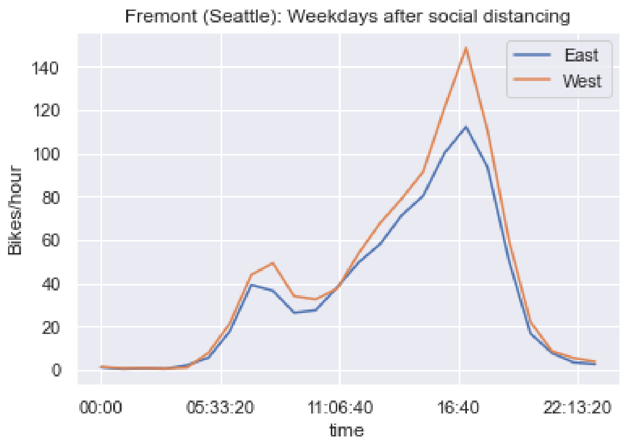


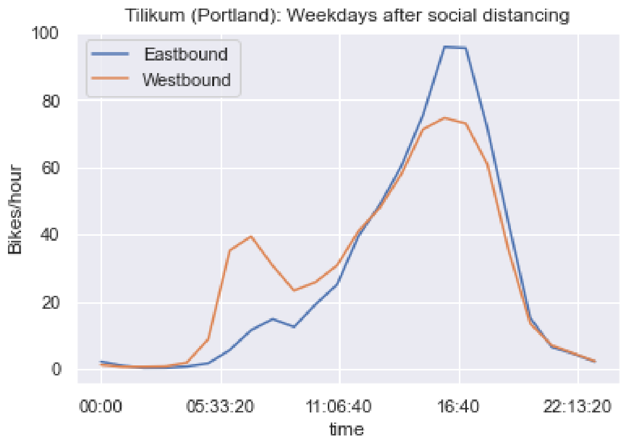


The Fremont bridge has good bike traffic flow in both directions during the peak hours, unlike the Tilikum. We conclude that **during peak hours, bikers are distributed more evenly on Seattle's Fremont bridge travel lanes than on Portland's Tilikum.**

XI.5.3 Changes after social distancing

```
[29]: weekplot(sd['2020-03-17':], title='Fremont (Seattle): Weekdays after␣
       ↪social distancing'); plt.ylabel('Bikes/hour');
      weekplot(th['2020-03-17':], title='Tilikum (Portland): Weekdays after␣
       ↪social distancing'); plt.ylabel('Bikes/hour');
```





Somewhat remarkably, despite all the above-seen differences, the weekday bike counts of both cities respond to social distancing in quite the same fashion: the bimodal weekday distribution of commuting to work has become a unimodal afternoon recreation pattern.

XII Visualizing geospatial data

May 6, 2020

Geospatial data refers to data which has a geographic component in it. Usually this means that the data entries are associated to some point on the surface of the earth. Therefore, geospatial data are usually visualized on maps.

Because the earth is round, in order to make a flat map, one must transform the earth's surface to a flat surface. Such transformations are called *projections* by cartographers (not to be confused with linear projections from linear algebra). Mathematicians know that a transformation between topologically different regions must be discontinous somewhere. So these projections, while very useful, cannot be perfect replicas of reality. It is useful to know this and a bit more about python modules for projections while attempting to visualize geospatial data on the globe.

Many references, including Geographic Data with Basemap of [JV-H], use the python module `basemap`. However in recent years, the module basemap has been deprecated in favor of the new python mapping project called Cartopy. Therefore, this activity aims at taking a quick look at cartopy. Cartopy, together with geopandas, a package built on top of pandas, shows promise in easing geospatial visualization. They are nonetheless relatively new efforts. You will notice that their documentation, while constantly improving, is not as mature as, say numpy. There will likely be a number of changes in the provided facilities as these efforts take hold. Our goal in this activity is to get acquainted with these new developments for visualizing geospatial data.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd
import geopandas as gpd
from cartopy import crs
```

XII.1 Geometry representation

The `GeoDataFrame` class of `geopandas` is a `pandas` data frame with a special column representing geometry. This column is a `GeoSeries` object, which may be viewed as a `pandas` series where each entry is a set of shapes. The shapes are geometric objects like a a set of points, lines, a single polygon, or many polygons. These shapes are objects made using the shapely package. Together they allow easy interaction with `matplotlib` for plotting geospatial data.

Here is a `GeoDataFrame` object we have already used in 01 Overview:

```
[2]: world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
     type(world)
```

```
[2]: geopandas.geodataframe.GeoDataFrame
```

The above-mentioned special column of the `world` data frame is this.

```
[3]: type(world.geometry)
```

```
[3]: geopandas.geoseries.GeoSeries
```

A `GeoDataFrame` can have many columns of type `GeoSeries`, but there can only be one *active* geometry column, which is what is returned by the attribute `GeoDataFrame.geometry`. Note also that the name of this active `GeoSeries` can be anything (not necessarily `geometry`), but in the `world` object, the column happens to be called `geometry`, as can be seen below.

```
[4]: world.geometry.name
```

```
[4]: 'geometry'
```

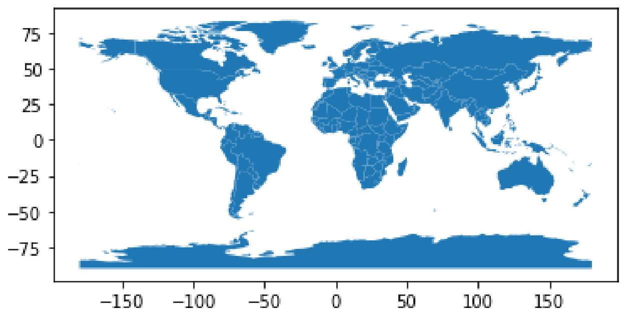
```
[5]: world.head()
```

```
[5]:      pop_est      continent                      name iso_a3  gdp_md_est  \
     0     920938        Oceania                      Fiji    FJI      8374.0
     1   53950935         Africa                  Tanzania    TZA    150600.0
     2     603253         Africa                 W. Sahara    ESH       906.5
     3   35623680  North America                    Canada    CAN   1674000.0
     4  326625791  North America  United States of America    USA  18560000.0

                                                 geometry
     0  MULTIPOLYGON (((180.00000 -16.06713, 180.00000...
     1  POLYGON ((33.90371 -0.95000, 34.07262 -1.05982...
     2  POLYGON ((-8.66559 27.65643, -8.66512 27.58948...
     3  MULTIPOLYGON (((-122.84000 49.00000, -122.9742...
     4  MULTIPOLYGON (((-122.84000 49.00000, -120.0000...
```

The `plot` method of the data frame is redefined in a `GeoDataFrame` to use the geometry objects in the active geometry column. So to plot this map, all we have to do is use the `plot` method:

```
[6]: world.plot()
```

```
[6]: <matplotlib.axes._subplots.AxesSubplot at 0x119f93ca0>
```



A `GeoDataFrame` has more attributes than a regular `pandas` data frame. For example, it stores the centroids of the shapes in the active geometry column.

```
[7]: type(world.centroid)
```

```
[7]: geopandas.geoseries.GeoSeries
```

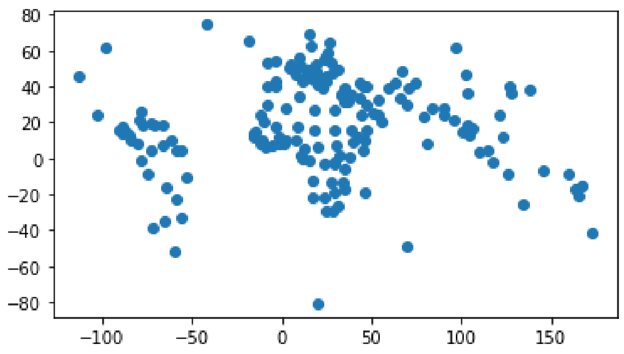
This is a `GeoSeries` that did not show up when we queried `world.head()`, but it is an attribute of `world`. We can, of course, make it an additional column of `world` by in the usual pandas way.

```
[8]: world['centroids'] = world.centroid
     world.head()
```

```
[8]:      pop_est      continent                      name iso_a3   gdp_md_est  \
     0     920938        Oceania                      Fiji    FJI       8374.0
     1   53950935         Africa                  Tanzania    TZA     150600.0
     2     603253         Africa                 W. Sahara    ESH        906.5
     3   35623680  North America                    Canada    CAN    1674000.0
     4  326625791  North America  United States of America    USA   18560000.0

                                                 geometry  \
     0  MULTIPOLYGON (((180.00000 -16.06713, 180.00000...
     1  POLYGON ((33.90371 -0.95000, 34.07262 -1.05982...
     2  POLYGON ((-8.66559 27.65643, -8.66512 27.58948...
     3  MULTIPOLYGON (((-122.84000 49.00000, -122.9742...
     4  MULTIPOLYGON (((-122.84000 49.00000, -120.0000...

                          centroids
     0  POINT (163.85316 -17.31631)
     1    POINT (34.75299 -6.25773)
     2   POINT (-12.13783 24.29117)
     3   POINT (-98.14238 61.46908)
     4  POINT (-112.59944 45.70563)
```

Now, `world` has two `GeoSeries` columns. If we make the `centroids` column the *active* geometry column, then the output of the `plot` method changes since it uses the active column's geometry specifications.

[9]:
```
world = world.set_geometry('centroids')    # change the active geometry␣
    ↪column
world.plot();
```



XII.2 Coordinate Reference Systems

An essential data structure of cartopy is CRS, or *Coordinate Reference Systems*, the name used by cartopy (and other python projects) for the projections used in maps. A CRS often used as default is the the Plate Carrée projection, which cartopy provides as follows.

[10]:
```
crs.PlateCarree()
```

[10]:
```
<cartopy.crs.PlateCarree at 0x11c140a90>
```

As you guessed from the above output, the CRS object is able to plot itself using matplotlib. This points to one avenue for visualizing geospatial data that has no need for geopandas. Cartopy produces a matplotlib axis on which you can overlay your data as you see fit: if your data has latitude and longitude associated to it, cartopy can apply the relevant projection automatically to place it at the proper place in the figure. Below, we will focus on alternative visualization methods using geopandas and facilities to interact between cartopy and geopandas.

The `world` object of class `GeoDataFrame` comes with a CRS attribute, another attribute that does not show up in `world.head()` output.

[11]:
```
world.crs
```

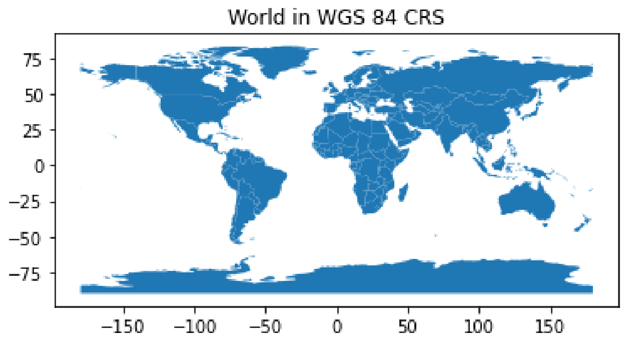
[11]:
```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

We see that the CRS above is codenamed EPSG:4326. Here EPSG stands for European Petroleum Survey Group. They maintain several data sets at https://spatialreference.org/ref/epsg/. Each set in "EPSG Geodetic Parameter Dataset is a collection of definitions of coordinate reference systems and coordinate transformations which may be global, regional, national or local in application."

EPSG-number 4326 that we have here belongs to the WGS 84 coordinate system, the latest in the World Geodetic System (WGS) where the earth is represented as an oblate spheroid with coordinates in decimal degrees (`Lat, Lon`).

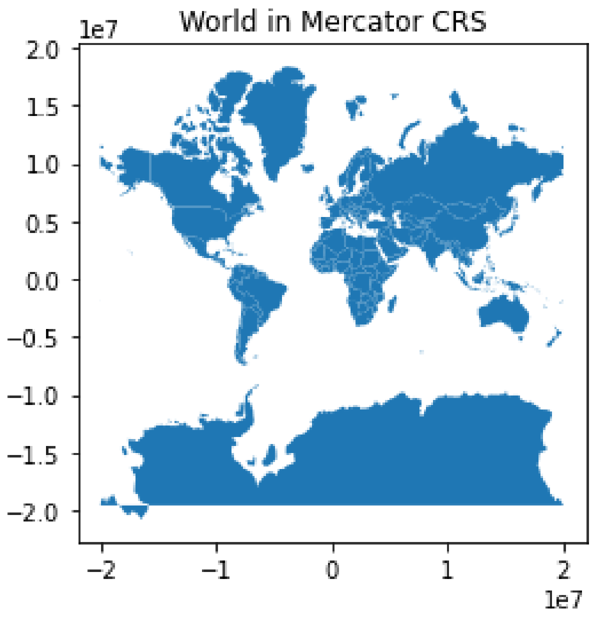
Changing the active geometry from previously set `centroids` to the original `geometry` column, let us plot the `world` again in order to compare the result with another CRS.

[12]:
```
world = world.set_geometry('geometry')    # set the active geometry
world.plot(); plt.title('World in WGS 84 CRS');
```



We compare this output with another very well-known projection, the Mercator projection, which has the nice property that it is conformal, i.e., it preserves angles. EPSG has made available a *Web Mercator* projection. We can easily convert `world` from the WGS 84 to the Mercator CRS:

[13]:
```
world_Mercator = world.to_crs("EPSG:3395")
world_Mercator.plot();
plt.title('World in Mercator CRS');
```



Generations of school kids have grown up with this Mercator map. Note how the Mercator projection distorts the size of objects as the latitude increases from the equator to the poles. Some countries near equator look smaller than they really are. For example, Brazil is actually *larger* than the contiguous United States, but it looks smaller in the Mercator projection. If you have time for a digression, have a look into the many discussions online, e.g., The Problem With Our Maps, on how the false sizes on maps (perhaps inadvertently) shape our views on countries.

XII.3 Two other CRS

The *Azimuthal Equidistant* and *Albers Equal Area* coordinate reference systems show areas of the globe in better proportions than the Mercator projection, as their names suggest. These are implemented in cartopy. We want to leverage geopandas' ability to work with cartopy in the next step. We don't always get meaningful plots after an arbitrary CRS to CRS conversion, but what is officially possible is laid out in the current geopandas documentation, which would be a good source to check back on for future changes.

First, for conversion from the default WGS 84 to the azimuthal equidistant CRS, we create a cartopy CRS object.

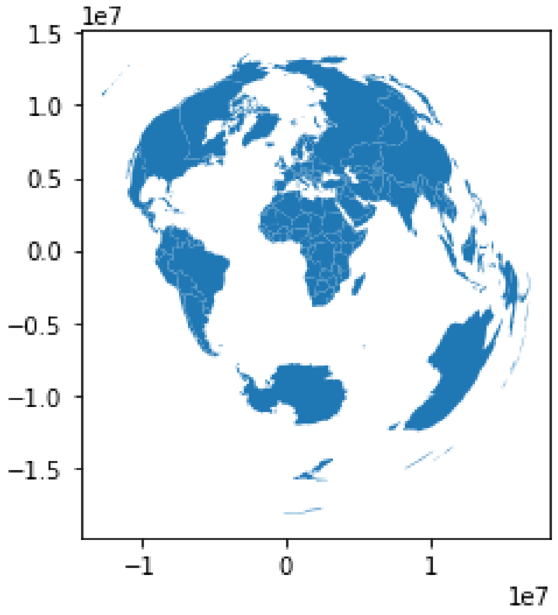
[14]:
```
ae = crs.AzimuthalEquidistant()
type(ae)
```

[14]:
```
cartopy.crs.AzimuthalEquidistant
```

Then we convert the cartopy object to a form usable by geopandas through an intermediate step. That step offers a glimpse of what lies at the core of many of the mapping tools, the PROJ project. (All these dependencies made installing geopandas more involved, as you recall from our early install sessions.)

```
[15]: aeproj4 = ae.proj4_init              # Convert to`proj4` string/dict usable␣
      ↪in gpd
      world_ae = world.to_crs(aeproj4)  # Then call to_crs method
      world_ae.plot()
```

```
[15]: <matplotlib.axes._subplots.AxesSubplot at 0x12794f3a0>
```



This represents the geopandas object `world_ae`, which is the `world` object whose geometry has been converted to the azimuthal equidistant CRS. From the above output, it we see that the azimuthal equidistant projection shows the central view in good proportions, with obvious distortions farther out although the distortions are evocative of the globe. However, this CRS is often difficult to use for showing data that is spread over the populous parts of the globe. (You can change the central view as shown in the next cell.) See, for example, how difficult it is to get the far east, the west, and Europe, together in any perspective, due to the vastness of the intermediate Pacific ocean.

```
[16]: crs.AzimuthalEquidistant(central_longitude=200, central_latitude=10)
```

```
[16]: <cartopy.crs.AzimuthalEquidistant at 0x127856310>
```

It is therefore useful to get to know another projection from `cartopy` called `AlbersEqualArea` projection.

```
[17]: aea = crs.AlbersEqualArea()
      aea
```

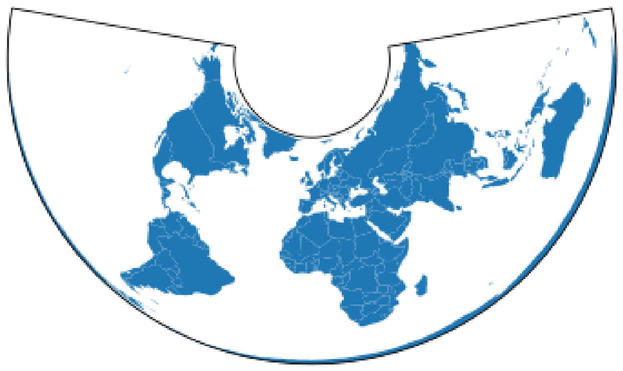
```
[17]: <cartopy.crs.AlbersEqualArea at 0x127a86e50>
```

Finally, as an illustration of how to plot geopandas geometries into an axis generated by cartopy, we will convert or project the existing geometry objects in `world_ae` to `AlbersEqualArea` CRS as shown below.

```
[18]: aea_geo = [aea.project_geometry(ii, src_crs=ae)
                for ii in world_ae['geometry'].values]
```

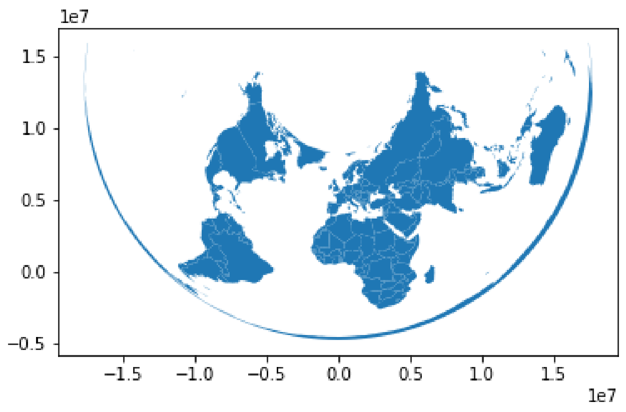
Since `cartopy` works directly with `matplotlib`, we can immediately render the resulting geometries in matplotlib's axis.

```
[19]: fig, ax = plt.subplots(subplot_kw={'projection': aea})
      ax.add_geometries(aea_geo, crs=aea);
```



We can alternately produce essentially the same plot using geopandas as follows.

```
[20]: gpd.GeoDataFrame(world, geometry=aea_geo, crs=aea.proj4_init).plot();
```



XII.4 Mapping COVID-19 cases on the globe

As an application of the above-discussed geospatial visualization techniques, we will now make a map of COVID-19 cases throughout the world using the `AlbersEqualArea` coordinate reference system.

```
[21]: import os
      from git import Repo
```

First update your data folder by pulling the newest reports of COVID-19 from the GitHub repository maintained by Johns Hopkins' researchers.

```
[22]: covidfolder = '../../data_external/covid19'
      if os.path.isdir(covidfolder):    # if repo exists, pull newest data
          repo = Repo(covidfolder)
          repo.remotes.origin.pull()
      else:                             # otherwise, clone from remote
          repo = Repo.clone_from('https://github.com/CSSEGISandData/COVID-19.
       ↪git',
                                 covidfolder)
      datadir = repo.working_dir + '/csse_covid_19_data/
       ↪csse_covid_19_time_series'
      f = datadir + '/time_series_covid19_confirmed_global.csv'
```

```
[23]: c = pd.read_csv(os.path.abspath(f))
      c = c.rename(columns={'Country/Region': 'country'}).iloc[:, 1:]
      c.head()
```

```
[23]:        country       Lat       Long  1/22/20  1/23/20  1/24/20  1/25/20  \
      0  Afghanistan  33.93911  67.709953        0        0        0        0
      1      Albania  41.15330  20.168300        0        0        0        0
      2      Algeria  28.03390   1.659600        0        0        0        0
      3      Andorra  42.50630   1.521800        0        0        0        0
      4       Angola -11.20270  17.873900        0        0        0        0

         1/26/20  1/27/20  1/28/20  ...  7/14/20  7/15/20  7/16/20  7/17/20  \
      0        0        0        0  ...    34740    34994    35070    35229
      1        0        0        0  ...     3667     3752     3851     3906
      2        0        0        0  ...    20216    20770    21355    21948
      3        0        0        0  ...      861      862      877      880
      4        0        0        0  ...      541      576      607      638

         7/18/20  7/19/20  7/20/20  7/21/20  7/22/20  7/23/20
      0    35301    35475    35526    35615    35727    35928
      1     4008     4090     4171     4290     4358     4466
      2    22549    23084    23691    24278    24872    25484
      3      880      880      884      884      889      889
      4      687      705      749      779      812      851

      [5 rows x 187 columns]
```

Some countries are repeated (as their provinces are being counted in separate data entries), as can be seen from the following nonzero difference:

```
[24]: len(c['country']) - len(set(c['country']))
```

```
[24]: 78
```

Therefore, we sum up each country's confirmed COVID-19 counts for each day using a pandas groupby operation. Of course, it doesn't make sense to sum up the latitudes and longitudes, so we take their average values within countries. (Taking the mean of latitudes and longitudes will do for our current purposes, but it is definitely not a perfect solution, which is why I'll be asking you to improve on it by making a chloropleth map in the next assignment.)

```
[25]: cg = c.groupby('country')[c.columns[3:]].sum()
      cg['Lat'] = c.groupby('country')['Lat'].mean()
      cg['Long'] = c.groupby('country')['Long'].mean()
```

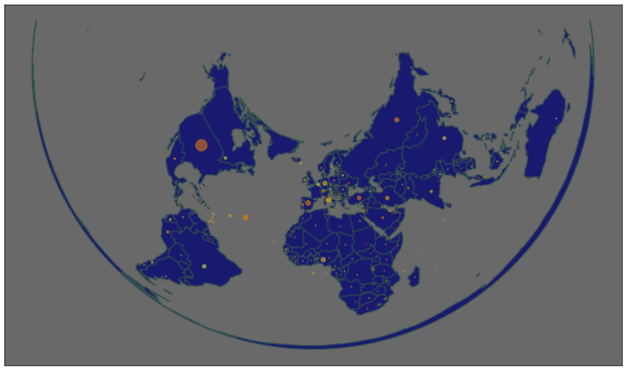
This newly created data frame has no `GeoSeries`. We use the latitude and longitude information to create point geometries. With this information, we can make a `GeoDataFrame`.

```
[26]: geo = gpd.points_from_xy(cg['Long'], cg['Lat'])
      c_aea_geo = [aea.project_geometry(ii) for ii in geo]
      cg = gpd.GeoDataFrame(cg, geometry=c_aea_geo, crs=aea.proj4_init)
```

Now, in cg, we have a `GeoDataFrame` object that should know how to plot its data columns in the data's associated places on the globe.

```
[27]: def covidworldmap(date):

          fig, ax = plt.subplots(figsize=(12, 10))
          # put the world map on an axis
          w = gpd.GeoDataFrame(world, geometry=aea_geo, crs=aea.proj4_init)
          w.plot(ax=ax, color='midnightblue', edgecolor='darkslategray')
          ax.set_facecolor('dimgray')
          mx = cg.iloc[:, :-3].max().max()    # get max across data

          # set marker sizes, with a min marker size for cases > 1000
          msz = 500 * np.where(cg[date]-1000, np.maximum(cg[date]/mx, 0.001), 0)
          cg.plot(ax=ax, cmap='Wistia', markersize=msz, alpha=0.5)

          ax.set_xticks([])   # remove axis marks
          ax.set_yticks([]);
```

```
[28]: covidworldmap('5/5/20')
```



The world is now littered with COVID-19 cases. I wish the world and our country had fared better, but the data doesn't lie.

XIII Gambler's Ruin

May 11, 2020

A gambler G starts with two chips at a table game in a casino, pledging to quit once 8 more chips are won. G can either win a chip or lose a chip in each game, with probabilities p and $1 - p$, respectively (independent of past game history). What is the probability that G accumulates a total of 10 chips playing the game repeatedly, without being ruined while trying? The ruining state is one where G has no chips and can no longer play. What is the probability that G is ruined while trying to get to 10 chips?

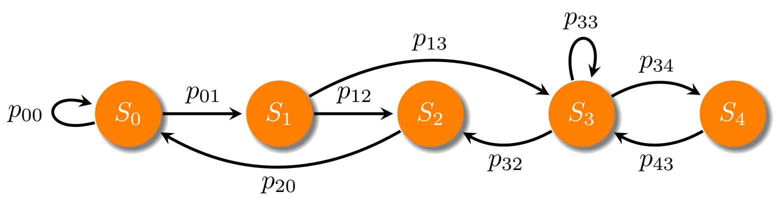
The goal of this activity is to introduce you to the rich subject of Markov chains, and in the process also get you acquainted with graphs, random walks, and stochastic matrices, with the gambler G as an entry point example. Since a probability course is not a prerequisite for this course, I will try to present the results colloquially and without proofs, with my advance apologies to the probabilists among you. The two theorems you will find stated below are proved using probabilistic tools (see, for example, [S]) and is material one might usually find in a statistics program. In the next activity, I will connect this to material from other fields.

XIII.1 Markov chains

A Markov chain is an abstraction used to model systems that transition from a current state to the next state according to some given probability. It has proven itself to be a powerful construct in statistics due to its wide applicability. Specifically, given

- a set of *states* $\mathbb{S} = \{S_0, S_1, \ldots\}$,
- and a set of numbers $0 \leq p_{ij} \leq 1$,

a *Markov chain* is a sequence whose elements are taken from $\mathbb{S}$ in such a way that probability to go from state S_i to S_j is p_{ij}. The number p_{ij} is called the *transition probability.* The states and transition probabilities are often represented in diagrams like this:



The assumptions when considering a Markov chain are that the system is required to move from state to state (the next state can be the same as the current state), and that the next state is determined only by the current state's transition probabilities (not by prior states). The latter is called the *memorylessness property* or the Markov property. The former implies that the probability that the system will transition from the current state is one, so that for any i,

$$\sum_j p_{ij} = 1.$$

Note that the sum above may be finite or infinite: if the set of states is a finite set, say $\mathbb{S} = \{S_0, S_1, \ldots, S_N\}$, then the above sum runs from $j = 0, 1$, through N; otherwise the sum should be treated as an infinite sum. Irrespective of the finiteness of the set of states $\mathbb{S}$, the Markov chain itself is thought of as an infinite sequence.

(Optional note: Here is a formal definition using the conditional probability notation, $\texttt{Pr}(A|B)$. A stochastic sequence X_n taking values from a set of states $\mathbb{S} = \{S_0, S_1, \ldots\}$ is called a Markov chain if for any subset of states $S_i, S_j, S_{k_0}, S_{k_1}, \ldots, S_{k_{n-1}} \in \mathbb{S}$,

$$\begin{aligned}
&\texttt{Pr}(X_{n+1} = S_j | X_n = S_i) \\
&\quad = \texttt{Pr}(X_{n+1} = S_j | X_n = S_i, X_{n-1} = S_{k_{n-1}}, X_{n-2} = S_{k_{n-2}}, \ldots, X_0 = S_{k_0}) \\
&\quad = p_{ij}.
\end{aligned}$$

Throughout, we only consider what are known as *time-homogeneous* Markov chains, where the probabilities p_{ij} are independent of the "time" step n.)

To connect this to the concept of random walks, let us first introduce graphs.

XIII.2 Graphs

In mathematics, we often use the word graph in a sense completely different from the graph or plot of a function.

A *graph* (V, E) is a set V of n *vertices*, together with a set E of m *edges* between (some) vertices. Although vertices are often pictorially represented as points, technically they can be whatever things lumpable into a set V, e.g., - people, labels, companies, power stations, cities, etc.

Edges are often pictorially represented as line segments (or curves) connecting two points representing two vertices, but technically, they are just a "*choice of two vertices*" (not necessarily distinct) from V, e.g., corresponding to the above-listed vertex examples, an edge can represent - friendship, similarities, spinoffs, wires, roads, etc.

When the above-mentioned "choice of two vertices" is changed to an ordered tuple, then the ordering of the two vertex choices that form an edge is significant, i.e., the edge has a direction. Thus a directed edge from vertex v_i to vertex v_j is the tuple (v_i, v_j). If all edges in E are directed, the graph is called a *directed graph* or a *digraph*. If a non-negative number, a *weight*, is associated to each edge of a digraph, then we call the graph a *weighted digraph*.

Python does not come with a graph data structure built in. Before you begin to think this somehow runs counter to the "batteries-included" philosophy of python, let me interrupt. Python's built-in dictionary data structure encapsulates a graph quite cleanly. Here is an example of a graph with vertices `a, b, c, d`:

[1]:
```
gd = {'a': ['b', 'd'],              # a -> b,  a -> d
      'b': ['c', 'd', 'a'] }        # b -> c,  b -> d, b -> a
```

You can use a dict of dicts to incorporate more edge properties, such as assign names/labels, or more importantly, weights to obtain a weighted digraph.

[2]:
```
gd = {'a': {'b': {'weight': 0.1},
            'd': {'weight': 0.8}},
      'b': {'d': {'weight': 0.5},
            'c': {'weight': 0.5}}
     }
```

Although we now have a graph data structure using the python dictionary, for this to be useful, we would have to implement various graph algorithms on it. Thankfully, there are many python packages that implement graph algorithms. Let's pick one package called NetworkX as an example. Please install it before executing the next code cell. NetworkX allows us to send in the above dictionary to its digraph constructor.

[3]:
```
import networkx as nx

g = nx.DiGraph(gd)     # dictionary to graph
```

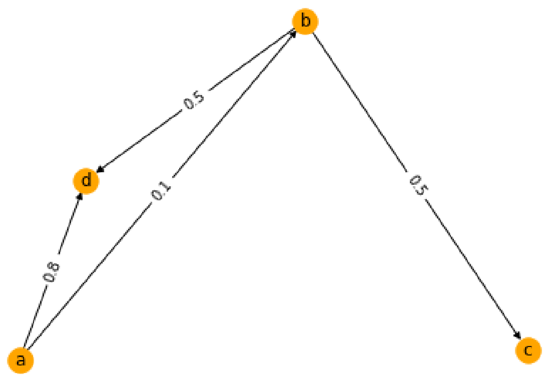
Now g is a DiGraph object with many methods. To see all edges connected to vertex a, a dictionary-type access is provided. We can use it to double-check that the object is made as intended.

[4]:
```
g['a']
```

[4]:
```
AtlasView({'b': {'weight': 0.1}, 'd': {'weight': 0.8}})
```

You can plot this graph using NetworkX's facilities (which uses matplotlib under the hood).

[5]:
```
import matplotlib.pyplot as plt
%matplotlib inline

def plot_gph(g):
    pos = nx.spectral_layout(g)
    nx.draw(g, pos, with_labels=True, node_color='orange')
    labels = nx.get_edge_attributes(g, 'weight')
    nx.draw_networkx_edge_labels(g, pos, edge_labels=labels);

plot_gph(g)
```



XIII.3 Random walks

Consider a weighted digraph (V, E) where the weight associated to a directed edge $e = (v_i, v_j)$ is a number $0 < p_{ij} \leq 1$. Let us extend these numbers to every pair of vertices v_i and v_j such that $p_{ij} = 0$ if (v_i, v_j) is not an edge of the graph. Let us restrict ourselves to the scenario where

$$\sum_j p_{ij} = 1$$

for any i.

A *random walk* on such a directed graph is a sequence of vertices in V generated stochastically in the following sense. Suppose the nth element of the sequence is the ith vertex v_i in V. Then one of its outgoing edges (v_i, v_j) is selected with probability p_{ij}, and the $(n+1)$th element of the random walk sequence is set to v_j. This process is repeated to get the full random walk, once a starting vertex is selected.

XIII.4 Conceptual equivalences

There are three equivalent ways of viewing what is essentially the same concept:

- a probabilistic transition of states,
- a vertex-to-vertex probabilistic movement in digraphs, or
- a non-negative matrix of unit row sums.

Given a random walk on a weighted digraph, the sequence it generates is a Markov chain. Indeed, the digraph's edge weights give the transition probabilities. The graph vertices form the Markov chain states. Conversely, given a Markov chain, there is a corresponding random walk. We first generate a digraph using the Markov chain states as the graph vertices. Positive transition probabilities indicate which directed edges should exist in the graph and what their edge weight should be. The sequence of states of the Markov chain is now identifiable as the sequence of vertices generated by a random walk on this digraph. This equivalence is betrayed even by our very first figure above, where we illustrated a Markov chain using a graph.

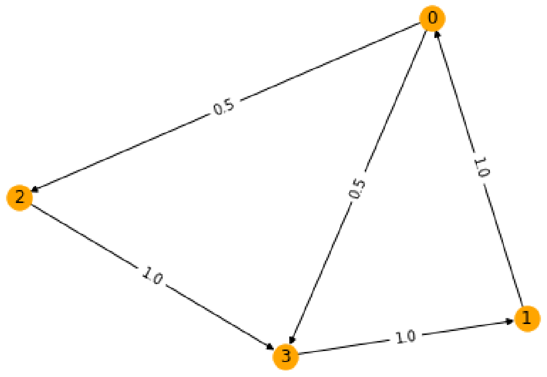
To understand why the third concept is equivalent, it is sufficient to note that all information to specify either a Markov chain, or a random walk is encapsulated in a single mathematical object, namely the matrix P whose (i,j)th entry is p_{ij}. This matrix of probabilities is called a *transition matrix* (sometimes also called a *stochastic matrix*) and it can be associated either to a Markov chain or a random walk provided its rows sum to one.

Here is an example of a transition matrix.

[6]:
```
import numpy as np
np.set_printoptions(suppress=True)

#               S0   S1   S2   S3
P = np.array([[0,   0.0, 0.5, 0.5],   # S0
              [1.0, 0.0, 0.0, 0.0],   # S1
              [0.0, 0.0, 0.0, 1.0],   # S2
              [0,   1.0, 0.0, 0.0]])  # S3

# Here S0, S1, S2, S3  are conceptual labels either
# for the Markov chain states or the digraph vertices
```

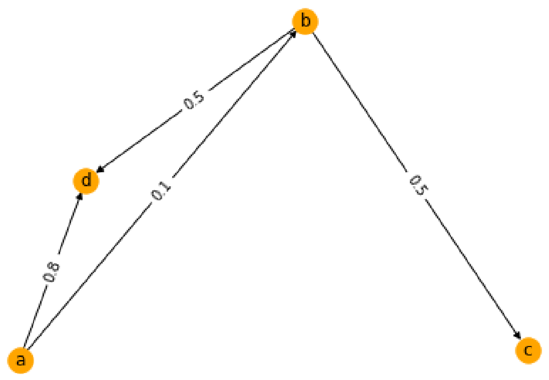
Matrix to digraph The above-mentioned conceptual equivalences are often tacitly used in graph programs. For instance, in NetworkX, one can easily make a graph out of the above transition matrix P as follows.

[7]:
```
gP = nx.from_numpy_array(P, create_using=nx.DiGraph)

plot_gph(gP)
```



Digraph to matrix We can, of course, also go the other way. For example, consider the small graph g we made "by hand" previously:

[8]:
```
plot_gph(g)
```



```
[9]: g.nodes    # note the ordering of vertices
```

```
[9]: NodeView(('a', 'b', 'd', 'c'))
```

This NetworkX object `g` can produce a matrix of the graph's edge weights. It is typical to make this as a `scipy.sparse` matrix since one anticipates many zero entries (corresponding to edges that do not exist in a realistic large graph). The result `Pg` below is a sparse matrix, which we convert to a dense matrix, just for printing.

```
[10]: Pg = nx.convert_matrix.to_scipy_sparse_matrix(g)
      Pg.todense()
```

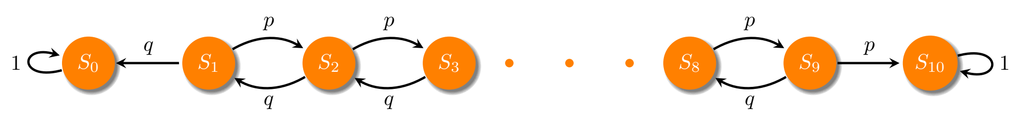
```
[10]: matrix([[0. , 0.1, 0.8, 0. ],
              [0. , 0. , 0.5, 0.5],
              [0. , 0. , 0. , 0. ],
              [0. , 0. , 0. , 0. ]])
```

Note how the matrix entries and edge weights in the figure are in correspondence. The matrix is generally called the *adjacency matrix* of a weighted graph. (Note that many textbooks will define adjacency matrix entries with 1 in place of the nonzeros above to accommodate graphs without weights.) For digraphs in a random walk discussion, we shall refer to this adjacency matrix as the *transition matrix*, as previously noted.

XIII.5 The example of the gambler

Let us return to the gambler G with whom we made the acquaintance in the beginning of this activity. We formulate a Markov chain for G as follows.

Let S_i be the state of play where G has i chips. In the next step of the game, G can win the game and go to S_{i+1} with probability p, or lose and go to state S_{i-1} with probability $q = 1 - p$. The only possible states for G to be in are $S_0, S_1, \ldots, S_{10}$. The directed graph on which G is the random walker is as follows.



Here we have also indicated two additional pieces of information: G has pledged to quit upon reaching state S_{10}, so once the Markov chain reaches S_{10} it will not go to any other state forever. Furthermore, if G's Markov chain reaches the ruining state of S_0, then G can't play any more, so the Markov chain cannot go to any other state forever.

Let us look at the corresponding transition matrix, say when $p = 0.4$.

```
[11]: def PforG(p=0.4, N=10):
          q = 1 - p
          P = np.diag(q*np.ones(N), k=-1) + np.diag(p*np.ones(N), k=1)
          P[0, :] = 0
          P[0, 0] = 1
          P[N, :] = 0
          P[N, N] = 1
          return P

      PforG(p=0.4)
```

```
[11]: array([[1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
             [0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
             [0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
             [0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. , 0. ],
             [0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. ],
             [0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. ],
             [0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. ],
             [0. , 0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. ],
             [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. ],
             [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4],
             [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. ]])
```

In the scenario described in the beginning, G starts with 2 chips and wants to reach a total of 10 chips. This translates to the following question: what is the probability that the above random walk enters the state S_{10}? It turns out that such questions are so often asked off Markov chains that the answer is a basic theoretical result on Markov chains (see the next theorem below).

XIII.6 Getting to a state

Let A and B be a partitioning of indices such that the subset of states

$$\mathbb{S}_A = \{S_i : i \in A\}, \qquad \mathbb{S}_B = \{S_i : i \in B\}$$

form a disjoint partitioning of the set $\mathbb{S}$ of all the states of a Markov chain. The probability that the Markov chain attains a state in $\mathbb{S}_A$ in some finite number of steps (i.e., if it ever hits $\mathbb{S}_A$) starting from a state S_i is called a *hitting probability* of $\mathbb{S}_A$ and is denoted by h_i. Of course, if $i \in A$, then $h_i = 1$. What is not obvious is the value of h_i when the chain starts from $i \in B$. Classic probabilistic arguments prove that the h_i values satisfy the system in the next theorem.

Theorem 1. The hitting probability of a subset A of states in a Markov chain is the minimal non-negative solution of the system of equations

$$h_i = \sum_j p_{ij} h_j$$

for all i with $i \in B$, and $h_i = 1$ if $i \in A$. Here minimality means that if x is another solution, then $x_i \geq h_i$ for all i.

The reasoning that leads to the system of equations in the theorem is as follows: if the Markov chain starts from state S_i, then in the next step, it can be in any of the states S_j with probability p_{ij}, from where the hitting probability is h_j, so the hitting probability h_i from S_i must be the sum of all $p_{ij} \times h_j$ over all states S_j. This idea can be formalized into a proof of Theorem 1. Let me highlight a few more things to note about Theorem 1:

- *First trivial solution:* From the definition of Markov chain, recall that

  $$\sum_j p_{ij} = 1.$$

  Hence an obvious solution to the system of equations in Theorem 1 is

  $$h_i = 1$$

  for all i. However, this solution need not be the *minimal* one mentioned in the theorem.

- *Second trivial solution:* One case where the minimal nonnegative solution is obvious is when A is such that $p_{ij} = 0$ for all $i \in B$ and $j \in A$, i.e., when it is not possible to go from B to A. Then

  $$h_i = \begin{cases} 1, & i \in A, \\ 0, & i \in B, \end{cases}$$

  obviously satisfies the system of equations in Theorem 1. Since the h_i values for $i \in B$ cannot be reduced any further, this is the minimal solution.

- Collecting h_i into a vector h, the system of equations in Theorem 1 can *almost* be written as the eigenvalue equation for eigenvalue 1,

  $$h = Ph,$$

  but *not* quite, because the equation need not hold for indices $i \in A$. Indeed, as stated in the theorem, h_i must equal 1 if $i \in A$.

XIII.7 Application to the gambler G

Setting $A = \{10\}$ and $B = \{0, 1, \ldots, 9\}$ in the above general theory, we see that G wins with probability h_2. Let us try to apply Theorem 1 to calculate h_2.

The approach I present here is not standard, but has the advantage that it uses eigenvector calculations that you are familiar with from your prerequisites. Even though the eigenvector remark I made at the end of the previous section is pessimistic, looking at G's transition matrix, we find that the condition $h_i = 1$ for $i \in A$ can be lumped together with the remaining equations. Indeed, because the last row of P contains only one nonzero entry (of 1),

$$h_{10} = \sum_j p_{10,j} h_j$$

holds. Therefore in G's case, h is a solution of the system in Theorem 1 if and only if h is a non-negative eigenvector of P corresponding to eigenvalue 1 and scaled to satisfy $h_{10} = 1$. (Be warned again that this may or may not happen for other Markov chains: see exercises.) What gives us further hope in the example of G is that we have a key piece of additional information:

$$h_0 = 0,$$

i.e., if G starts with no chips, then G cannot play, so G will stay in state S_0 forever. We might guess that this condition will help us filter out the irrelevant first trivial solution with $h_i = 1$ for all i.

Let me make one more remark before we start computing. The system of equations of Theorem 1 in the case of G reduces to

$$h_i = p h_{i+1} + (1-p) h_{i-1}$$

for $1 \le i \le 9$ together with $h_0 = 0$ and $h_{10} = 1$. You can make intuitive sense of this outside the general framework of the theorem. If G starts with i chips ($1 \le i \le 9$) so that the probability of hitting A is h_i, then in the next step there are two cases: (a) G has $i+1$ chips with probability p, or (b) G has $i-1$ chips with probability $1-p$. The probability of hitting A in case (a) is $p \times h_{i+1}$, and the probability of hitting A in case (b) is $q \times h_{i-1}$. Hence h_i must be equal to the sum of these two, thus explaining the theorem's equation $h_i = p h_{i+1} + (1-p) h_{i-1}$.

Let us now compute h_i using the knowledge that in G's case, h is a non-negative eigenvector of P corresponding to eigenvalue 1, scaled to satisfy $h_{10} = 1$.

[12]:
```
from numpy.linalg import eig, inv, det

P = PforG(p=0.4)
ew, ev = eig(P)
ew
```

[12]:
```
array([-0.93184127, -0.79267153, -0.57590958, -0.30277358, -0.        ,
        0.93184127,  0.79267153,  0.30277358,  0.57590958,  1.        ,
        1.        ])
```

The computed set of eigenvalues of P include 1 *twice*. Since the diagonalization (the factorization produced by `eig`) was successful, we know that the eigenspace of P corresponding to eigenvalue 1 is two-dimensional. *If* there are vectors h in this eigenspace satisfying

$$h_0 = 0, \quad h_{10} = 1,$$

then such vectors clearly solve the system in Theorem 1. We can algorithmically look for eigenvectors satisfying these two conditions in a two-dimensional space: it's a system of two equations and two unknowns, made below in the form

$$Mc = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

```
[13]: H = ev[:, abs(ew - 1) < 1e-15]      # Eigenvectors of eigenvalue 1
      M = np.array([H[0, :], H[-1, :]])   # Matrix of the two conditions
      det(M)
```

```
[13]: 0.2825542003687932
```

The nonzero determinant implies that M is invertible. This means that there is a unique solution c and hence a *unique* vector in the eigenspace satisfying both the conditions, which can be immediately computed as follows.

```
[14]: def Gchances(p=0.4, N=10):
          P = PforG(p, N)
          ew, ev = eig(P)
          H = ev[:, abs(ew - 1) < 1e-15]
          M = np.array([H[0, :], H[-1, :]])
          c = inv(M) @ np.array([0, 1])
          return H @ c
```

```
[15]: h = Gchances(p=0.4)
      h
```

```
[15]: array([0.        , 0.00882378, 0.02205946, 0.04191297, 0.07169324,
             0.11636364, 0.18336924, 0.28387764, 0.43464024, 0.66078414,
             1.        ])
```

The significance of the above-mentioned uniqueness is that we no longer have to check if this `h` is the *minimal non-negative* solution of Theorem 1, since we have no more degrees of freedom to further reduce the above non-negative components.

We have just solved G's problem posed in the beginning.

The answer `h`, printed in the output above, tells us that *the probability of G accumulating 10 chips starting from 2 chips when $p = 0.4$ is* `h[2]`, whose value is approximately 0.022.

This is a lousy probability! Have we made a mistake? We began by assuming that the casino gives G *almost* a fair chance at winning each game, at a probability of $p = 0.4$, which is pretty close to the exactly fair chance of $p = 0.5$ (which we suspect no casino would give). Yet, the chance of G getting out with 10 chips is much less than p, per our computation. In fact, looking at which printed out entries of `h` that are above 0.5, we find that G has more than a 50% chance of making 10 chips only if G starts with 9 chips!

XIII.8 Cross checking

The answer we got above is correct, even if not intuitive. In fact, this is a manifestation of the phenomena that goes by the name of **Gambler's Ruin.** How can we double-check the above answer? One way to double-check the answer is the analytical technique described in the optional exercise below.

Optional exercise: Solve the equations of Theorem 1 in closed form to conclude that the probability of G making 10 chips, starting from i chips, is

$$h_i = \frac{1 - (q/p)^i}{1 - (q/p)^{10}}$$

whenever $p \neq q$.

I'll omit more details on this analytical way for verification, since this course is aimed at learning computational thinking. Instead, let's consider another computational way.

To let the computer cross check the answer, we design a completely different algorithm whose output "should" approximate the correct result. Namely, we simulate many many gambles, get the statistics of the outcomes, and then cross check the frequency of G's winnings. (That this "should" give the right probability is connected to the law of large numbers.)

Here is a simple way to implement many gambles (each gamble is a sequence of games until G reaches either S_0 or S_{10}) using the built-in random module that comes with python (and using the uniform distribution in $[0, 1]$).

[16]:
```python
from random import uniform

def gamble(init=2, p=0.4, win=10, n=10000):

    """Let G gamble "n" times, starting with "init" chips."""

    wl = np.zeros(n)    # mark win or lose here for each gamble i
    for i in range(n):
        chips = init
        while chips:
            if uniform(0, 1) > p:   # losing game
                chips -= 1
            else:                   # winning game
                chips += 1
            if chips == win:        # reached wanted winnings
                wl[i] = 1
                break
    return wl
```

[17]:
```python
n = 500000
wl = gamble(n=n)
print('Proportion of winning gambles:', np.count_nonzero(wl) / n)
```

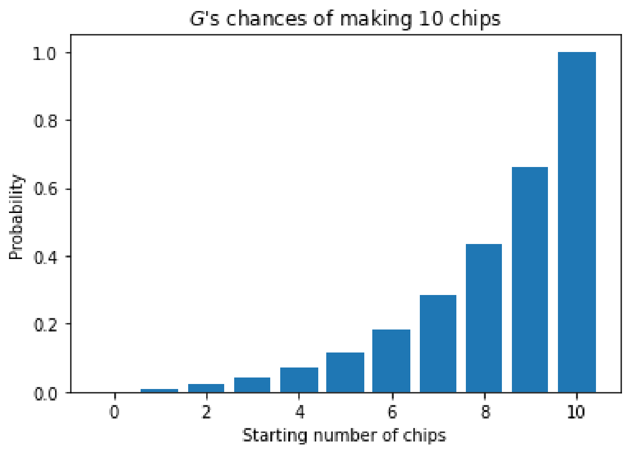
```
Proportion of winning gambles: 0.02191
```

The number produced as the output is pretty close to the previously obtained `h[2]`. Indeed, it gets closer to `h[2]` with increasing number of gambles. Now that we have built more confidence in our answer computed using the eigensolver, let us proceed to examine all the components of `h` more closely.

XIII.9 Gambler's Ruin

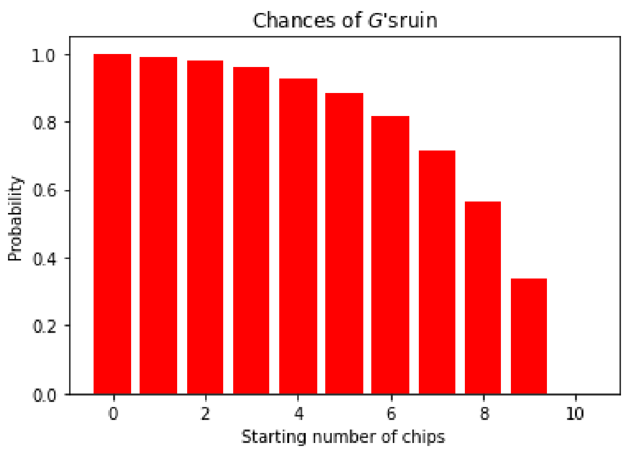
Visualizing `h` in a bar plot, we find that G's computed chances of winning seem to decrease exponentially as the starting chip count decreases.

```
[18]: plt.bar(range(len(h)), h)
      plt.title('$G$\'s chances of making 10 chips');
      plt.xlabel('Starting number of chips'); plt.ylabel('Probability');
```



Since G either quits winning or gets ruined with 0 chips (not both), the probability of G's ruin is $1 - h_i$.

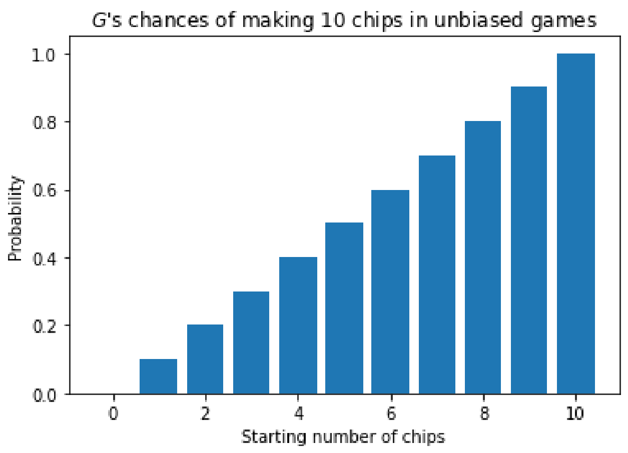
```
[19]: plt.bar(range(len(h)), 1-h, color='red')
      plt.title('Chances of $G$\'sruin');
      plt.xlabel('Starting number of chips'); plt.ylabel('Probability');
```



This exemplifies the concept of *Gambler's Ruin*: in a biased game (where $p < 1/2$), the probability of G's ruin could be much higher than the "intuitive" $1 - p$ for most starting values.

Note that if all games are unbiased with $p = 1/2$, then we get the following linear plot, which perhaps jives with the intuition more than the unbiased case.

```
[20]: plt.bar(range(len(h)), Gchances(p=0.5, N=10))
      plt.title('$G$\'s chances of making 10 chips in unbiased games');
      plt.xlabel('Starting number of chips'); plt.ylabel('Probability');
```



XIII.10 Absorbing Markov chains

To round out our discussion of hitting probabilities, I should tell you that there is another, easier, way to algorithmically compute h_i in some circumstances.

In the example of G's Markov chain, we were able to extract the minimal non-negative solution of Theorem 1 uniquely from an eigenspace. Unique representations of solutions make for nice algorithmic prescriptions. There is a class of Markov chains for which we can always find certain hitting probabilities through a unique representation, and we don't even have to compute an eigenspace for it: we just need to solve a linear system. This is the subject of the next theorem (Theorem 2 below) also proved using basic probability methods.

A state S_i of a Markov chain is called an *absorbing state* if $p_{ii} = 1$. Clearly, once the chain reaches an absorbing state, it cannot transition to any other state forever.

An *absorbing Markov chain* is a Markov chain which has at least one absorbing state and has paths made of directed edges from any state to an absorbing state.

Partition the states in an absorbing Markov chain using index sets A and B, like before, except that now A denotes the indices for all the absorbing states (and B indicates the remaining states). Then the following partitioning of the transition matrix is both easy to conceptualize and easy to implement using numpy's slicing operations:

$$P = \begin{bmatrix} P_{AA} & P_{AB} \\ P_{BA} & P_{BB} \end{bmatrix}$$

Note that P_{AA} is an identity matrix and P_{AB} is the zero matrix, because $p_{ii} = 1$ for all $i \in A$.

Example: The gambler G has two absorbing states S_0 and S_{10}, and G's Markov chain is an absorbing Markov chain. Setting $A = \{0, 10\}$ and $B = \{1, \ldots, 9\}$, the blocks of the above partitioning for this case are as follows:

[21]:
```
A = [0, 10]
B = range(1, 10)
P = PforG()
PAA = P[np.ix_(A, A)]
PBA = P[np.ix_(B, A)]
PBB = P[np.ix_(B, B)]
```

[22]:
```
PBA
```

[22]:
```
array([[0.6, 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0. ],
       [0. , 0.4]])
```

[23]:
```
PBB
```

[23]:
```
array([[0. , 0.4, 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4, 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0.6, 0. , 0.4],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.6, 0. ]])
```

[24]:
```
PAA
```

[24]:
```
array([[1., 0.],
       [0., 1.]])
```

As already noted above, P_{AA} should always be the identity matrix in an absorbing Markov chain per our definition.

Now we are ready to state the second result on hitting probabilities, this time with an easier algorithmic prescription for computing them.

Theorem 2. In any finite absorbing Markov chain, the matrix $I - P_{BB}$ is invertible (where I denotes the identity matrix of the same size as P_{BB}). Moreover, letting H denote the matrix whose (i, j)th entry equals the probability that the chain hits an absorbing state S_j, $j \in A$, starting from another state S_i, $i \in B$, in any number of steps, we may compute H by

$$H = (I - P_{BB})^{-1} P_{BA}.$$

Example: In one line of code, we can apply Theorem 2 to gambler G using the matrices made just before the theorem:

[25]:
```
np.linalg.inv(np.eye(len(B)) - PBB) @ PBA
```

[25]:
```
array([[0.99117622, 0.00882378],
       [0.97794054, 0.02205946],
       [0.95808703, 0.04191297],
       [0.92830676, 0.07169324],
       [0.88363636, 0.11636364],
       [0.81663076, 0.18336924],
       [0.71612236, 0.28387764],
       [0.56535976, 0.43464024],
       [0.33921586, 0.66078414]])
```

Since the *second* entry of A represents the winning state S_{10}, the *second* column above gives the probability of hitting S_{10} from various starting states. Note that that second column is the same as the previously computed `h`. The first column in the output above gives the probability of G's ruin for various starting states.

XIII.11 Greedy gambler

Let us conclude with another manifestation of the *Gambler's Ruin* concept that emerges when you ask the following question. What happens if G gets greedy and reneges on the pledge to quit upon reaching 10 chips? In other words, G continues to play infinitely many games unless ruined in between. What is the probability of G's ruin?

This is the same as considering $N = \infty$ case in our previous setting. This case results in an infinite set of states. Theorem 1 applies both to finite and infinite set of states, but we can only simulate Markov chains with finite number of states. Nonetheless, we can certainly apply Theorem 2 to compute the hitting probabilities for larger and larger N and get a feel for what might happen when $N = \infty$.

But first, we have to make our code better to go to large N. We use scipy's `sparse` facilities to remake the matrices and improve efficiency.

[26]:
```
from scipy.sparse import diags, eye
from scipy.sparse.linalg import spsolve
```

[27]:
```
def sparseGmats(p=0.4, N=10000):

    """ Return I - PBB and PBA as sparse matrices """

    q = 1 - p
    # Note that the first and last row of P are not accurate
    # in this construction, but we're going to trim them away:
    P = diags(q*np.ones(N), offsets=-1, shape=(N+1, N+1)) \
    +   diags(p*np.ones(N), offsets=1,  shape=(N+1, N+1))

    A = [0, N]
    B = range(1, N)
    I_PBB = (eye(N-1) - P[np.ix_(B, B)]).tocsc()
    PBA = P[np.ix_(B, A)].tocsc()

    return I_PBB, PBA

def ruinG(p=0.4, N=10000):

    """ Given that the winning probability of each game is "p",
    compute the probability of G's ruin for each starting state """

    I_PBB, PBA = sparseGmats(p, N)
    return spsolve(I_PBB, PBA[:, 0])
```

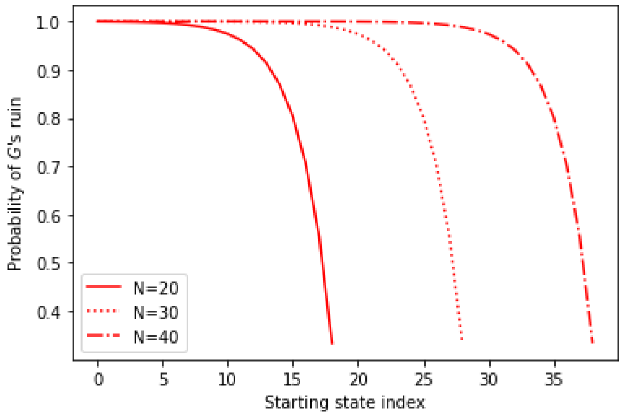
[28]:
```
ruinG(N=10)
```

[28]:
```
array([0.99117622, 0.97794054, 0.95808703, 0.92830676, 0.88363636,
       0.81663076, 0.71612236, 0.56535976, 0.33921586])
```

After verifying that for the $N = 10$ case, we obtained the same result, we proceed to examine the higher values of N. One quick way to visualize the resulting `h`-values are as plots over starting states.

[29]:
```
fig = plt.figure()
ax = plt.gca()
```

```
hs = ruinG(N=20)
ax.plot(hs[:21], 'r-', label='N=20')

hs = ruinG(N=30)
ax.plot(hs, 'r:', label='N=30')

hs = ruinG(N=40)
ax.plot(hs, 'r-.', label='N=40')

ax.set_ylabel('Probability of $G$\'s ruin')
ax.set_xlabel('Starting state index')
ax.legend();
```



Clearly, as N increases, we see a larger range of starting indices for which G hardly stands a chance of escaping ruin.

For a specific case, suppose G is unable to start with more than 20 chips, but is willing to play N games, for larger and larger N. Then we compute G's *least ruin probability*, the lowest probability of G's ruin among all possible starting values, namely

$$\min_{i=0,\ldots,20} h_i.$$

Let us examine how this minimal value changes with N.

[30]:
```
def least_ruin_prob(p=0.4, N0=20, dbl=11):

    """ Compute least ruin probability starting with N="N0" and
    recompute "dbl" times, doubling N each time. """

    for i in range(dbl):
        print('N = %5d,  least ruin probability = %5.4f'
              %(N0*2**i,  min(ruinG(p=p, N=N0*2**i)[:21])))
```

[31]:
```
least_ruin_prob(p=0.4, dbl=7)
```

```
N =    20,  least ruin probability = 0.3334
N =    40,  least ruin probability = 0.9995
N =    80,  least ruin probability = 1.0000
N =   160,  least ruin probability = 1.0000
N =   320,  least ruin probability = 1.0000
N =   640,  least ruin probability = 1.0000
N =  1280,  least ruin probability = 1.0000
```

Clearly, G is *ruined with certainty,* i.e., with probability 1, as $N \to \infty$.

What if the games are fair? The above results were with $p = 0.4$. Rerunning the code with the *fair* chance $p = 0.5$, we again observe convergence, albeit slower, to the inevitable.

```
[32]: least_ruin_prob(p=0.5, dbl=11)
```

```
N =    20,  least ruin probability = 0.0500
N =    40,  least ruin probability = 0.4750
N =    80,  least ruin probability = 0.7375
N =   160,  least ruin probability = 0.8687
N =   320,  least ruin probability = 0.9344
N =   640,  least ruin probability = 0.9672
N =  1280,  least ruin probability = 0.9836
N =  2560,  least ruin probability = 0.9918
N =  5120,  least ruin probability = 0.9959
N = 10240,  least ruin probability = 0.9979
N = 20480,  least ruin probability = 0.9990
```

What is illustrated in this output is often identified as another, perhaps stronger, manifestation of the *Gambler's Ruin* concept: *even when the games are fair,* G is certain to be ruined if G continues to play forever.

XIV Google's PageRank

May 13, 2020

In the history of the internet, a collection of papers proposing *PageRank* has been influential, in particular, a 1998 paper by Sergey Brin and Lawrence Page, two graduate students, now better known as Google co-founders. They proposed an objective metric to order the results of a user's internet search. For those who don't remember, there was indeed a time when "junk results often wash[ed] out any results that a user is interested in," to quote the paper. Of course, search engines now operate in a competitive business world, and the algorithms that Google and other companies use to rank their search results currently are not public knowledge. But the concept of PageRank is now routinely applied beyond Google, not just to the internet, but to general data on graphs and networks. It serves as an automatic tool to rank the relative importance of parts of any internet-like giant network.

We shall view the web (internet) as a directed graph. Each internet location (a webpage) is a vertex of the graph. A hyperlink from one webpage to another is a directed edge of the graph. From this viewpoint, the central idea of Brin & Page was to exploit the "link structure of the web to calculate a quality ranking for each web page."

To introduce PageRank, we shall build on our previous discussion of Markov chains (from Gambler's Ruin), which was entirely from the *statistical* or probabilistic perspective. Below, we will connect to theorems of Perron and Frobenius, which are results that one might usually learn in a *mathematics* program. Of course, all of this helps us understand the effectiveness of PageRank, a topic that has entered the *computer science* curricula in recent decades. Taken together, we then have an example of propitious convergence of ideas from the distinct fields of *computer science, mathematics, and statistics.*

XIV.1 Probability distributions on graphs

Throughout this discussion, we have in mind a directed graph with vertices $V_1, \ldots, V_N$, associated to a Markov chain with an $N \times N$ stochastic matrix $P = (p_{ij})$. We consider a random walker on this digraph, who we name W. The random walker W is a "stochastic being". We cannot know W's precise location on the graph; we only know that W's location is determined by a probability distribution on the graph.

A *probability vector* is a vector $x \in \mathbb{R}^N$ whose entries x_i satisfy

$$0 \le x_i \le 1, \qquad \sum_{i=1}^{N} x_i = 1.$$

Such a vector represents a probability distribution on the vertices of the graph. We may think of x_i as the probability that the system is in state V_i. Alternatively, we may think of x_i as the probability of finding the random walker W at the digraph vertex V_i.

How does the probability of finding W on the graph change when W takes a step? Here is another way of asking the same question: if x_i is the probability that the Markov chain is in state V_i, then what is the probability that the next state of the Markov chain is V_j? Since V_j can be arrived at from V_k with probability p_{kj}, and since the prior state was V_k with probability x_k, we conclude that the answer should be the sum of $p_{kj} \times x_k$ over all the prior states V_k. In other words, the probability that the next state is V_j equals

$$\sum_{k=1}^{N} p_{kj} x_k,$$

which is the jth component of $P^t x$. This argument can be formalized to obtain the following basic result.

Theorem 1. The probability distribution of a random walk on a directed graph with stochastic matrix P changes from x to $P^t x$ in each step (where P^t denotes the transpose of P).

Note that if x is a probability vector and P is a transition matrix, then $P^t x$ is guaranteed (exercise!) to come out as a probability vector.

XIV.2 Stationary distributions

We have just seen that as the random walk progresses, an initial probability distribution x changes as follows:

$$x, \quad P^t x, \quad (P^t)^2 x, \quad (P^t)^3 x, \quad \ldots.$$

Suppose this sequence converges to a limiting vector s. Then that limit should obviously not change if one more P^t is applied to it, i.e., it should satisfy

$$P^t s = s.$$

Any probability vector s satisfying $P^t s = s$ is called a *stationary distribution*, (or a *stationary probability vector* or an *equilibrium*) of the random walk. Notice that the stationary probability vector is always an eigenvector of P^t associated to eigenvalue 1. Notice also that the limit, if it exists, is *independent of the initial distribution* x.

For the random walker W, if the limit of the above sequence exists, then the stationary distribution can be used to identify the vertices of the graph with a high probability of finding W in the long run.

[1]:
```
import numpy as np
from numpy.linalg import eig, matrix_power, norm
```

Example A

[2]:
```
PA = np.array([[1/2,  1/4, 1/4],
               [1/3,  1/3, 1/3],
               [1/3,  1/3, 1/3]])
```

Does the sequence of probability distributions $x, \; P^t x, \; (P^t)^2 x, \; (P^t)^3 x, \ldots,$ converge for this Markov chain? To answer this, let's take the matrix powers $(P^t)^n$ and compute the

Frobenius norm of the successive differences,

$$\|(P^t)^{n+1} - (P^t)^n\|_F.$$

If this approaches 0, then we obtain a numerical indication of convergence.

```
[3]: [norm(matrix_power(PA.T, n+1) - matrix_power(PA.T, n), 'fro') for n in␣
      ↪range(1, 20)]
```

```
[3]: [0.1402709019216955,
      0.023378483653615948,
      0.0038964139422693537,
      0.0006494023237115095,
      0.00010823372061852081,
      1.80389534364696e-05,
      3.0064922394683963e-06,
      5.010820398912459e-07,
      8.351367329688623e-08,
      1.3918945514670359e-08,
      2.3198243477163972e-09,
      3.86637279525466e-10,
      6.44396235375308e-11,
      1.073992260029489e-11,
      1.789976112476022e-12,
      2.9830188647232445e-13,
      4.977570743895776e-14,
      8.26743511340278e-15,
      1.343782689223772e-15]
```

This indicates convergence. Let's check that the convergence actually occurs to a stationary distribution s that is an eigenvector of P^t.

```
[4]: ew, ev = eig(PA.T)
     ew
```

```
[4]: array([0.16666667, 1.        , 0.        ])
```

```
[5]: v = ev[:, abs(ew-1) < 1.e-14]; print(v)
```

```
[[0.68599434]
 [0.51449576]
 [0.51449576]]
```

This is the eigenvector corresponding to eigenvalue 1. In order to make this a probability distribution, let's normalize by the sum.

```
[6]: sA = v / v.sum()
     sA
```

```
[6]: array([[0.4],
            [0.3],
            [0.3]])
```

This is the stationary distribution s for this example. To see that the same vector is obtained as the limit of $(P^t)^n$, we simply raise the matrix to a large power and examine the result:

```
[7]: matrix_power(PA.T, 1000)
```

```
[7]: array([[0.4, 0.4, 0.4],
            [0.3, 0.3, 0.3],
            [0.3, 0.3, 0.3]])
```

The values of s show that the random walker W will, in the limit, be found in state V_0 at a higher probability (0.4) than the other two states (0.3).

Example B

```
[8]: PB = np.array([[0,   1/3, 1/3, 1/3],
                    [0.9, 0,   0,   0.1],
                    [0.9, 0.1, 0,     0],
                    [0.9, 0,   0.1,   0]])
```

```
[9]: ew, ev = eig(PB.T); print(ew)
```

```
[-0.9 +0.j          1.  +0.j         -0.05+0.08660254j -0.05-0.08660254j]
```

```
[10]: # stationary distribution:
      v = ev[:, abs(ew-1) < 1.e-14];
      sB = v.real / sum(v.real); print(sB)
```

```
[[0.47368421]
 [0.1754386 ]
 [0.1754386 ]
 [0.1754386 ]]
```

In this example, there is convergence of the powers to the stationary distribution, but it is slower than Example A. We find this out by taking higher powers than before:

```
[11]: [norm(matrix_power(PB.T, n) - sB, 'fro') for n in range(300, 305)]
```

```
[11]: [2.0819420081737047e-14,
       1.9224558387957245e-14,
       1.6814381771214046e-14,
       1.558237665379239e-14,
       1.3619335994971806e-14]
```

Example C

```
[12]: PC = np.array([[0,  1,  0],
                     [0,  0,  1],
                     [1,  0,  0]])
```

```
[13]: ew, ev = eig(PC.T); print(ew)
```

```
[-0.5+0.8660254j -0.5-0.8660254j  1. +0.j       ]
```

```
[14]: # stationary distribution:
      v = ev[:, abs(ew-1) < 1.e-14].real; sC = v/v.sum(); print(sC)
```

```
[[0.33333333]
 [0.33333333]
 [0.33333333]]
```

In this example, we do not see convergence of the powers P^t to the above stationary distribution. There seems to be no convergence to anything:

```
[15]: [norm(matrix_power(PC.T, n+1) - matrix_power(PC.T, n)) for n in range(100,␣
       ↪105)]
```

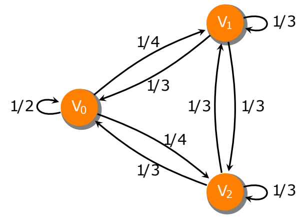
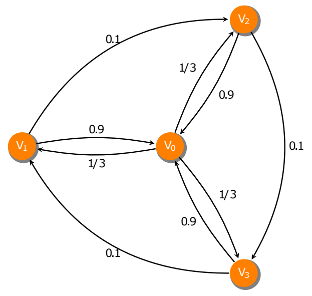
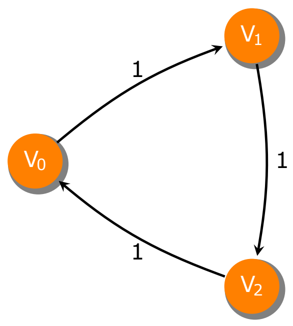
```
[15]: [2.449489742783178,
       2.449489742783178,
       2.449489742783178,
       2.449489742783178,
       2.449489742783178]
```

These numbers clearly do not seem to be approaching zero, a sign of non-convergence. In fact, the transition matrix here is such that all its powers cycle between three matrices P^t, $(P^t)^2$ and $(P^t)^3$, thus preventing convergence!

```
[16]: [print('The %dth power:\n'%i, matrix_power(PC.T, i)) for i in range(300,␣
       ↪306)];
```

```
The 300th power:
 [[1 0 0]
 [0 1 0]
 [0 0 1]]
The 301th power:
 [[0 0 1]
 [1 0 0]
 [0 1 0]]
The 302th power:
 [[0 1 0]
 [0 0 1]
 [1 0 0]]
The 303th power:
 [[1 0 0]
 [0 1 0]
 [0 0 1]]
The 304th power:
 [[0 0 1]
 [1 0 0]
 [0 1 0]]
The 305th power:
```

```
[[0 1 0]
[0 0 1]
[1 0 0]]
```

Example A	Example B	Example C
 V₁ 1/3 1/4 1/3 1/2 V₀ 1/3 1/3 1/4 1/3 V₂ 1/3	 V₂ 0.1 1/3 0.9 V₁ 0.9 1/3 V₀ 0.1 1/3 0.9 0.1 V₃	
Convergent: $\lim_{n\to\infty}(P^t)^n x = s$	Convergent: $\lim_{n\to\infty}(P^t)^n x = s$	Not convergent:$\lim_{n\to\infty}(P^t)^n$ doesn't exist
Stationary distribution: $s = \begin{bmatrix} 0.4 \\ 0.3 \\ 0.3 \end{bmatrix}$	Stationary distribution: $s = \begin{bmatrix} 0.474 \\ 0.175 \\ 0.175 \\ 0.175 \end{bmatrix}$	Stationary distribution: $s = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$

Summary of the three examples: Note how associating the values of s_i to vertex V_i produces something that matches our intuition on where to find the random walker in the long run. The convergence in Example A is a consequence of Perron's theorem that we discuss next.

XIV.3 Perron's theorem

The following celebrated result in linear algebra was proved by Oskar Perron (about 90 years before Brin & Page's paper). Research papers continue to be written on subjects surrounding the theorem. The theorem applies to any *positive matrix:* a square matrix is called a positive matrix if all its entries are positive.

Theorem 2. The following statements hold for any positive matrix A.

- There is a positive real number μ that is an eigenvalue of A such that any other eigenvalue λ of A is smaller in absolute value: $|\lambda| < \mu$. (This μ is called the dominant eigenvalue of A.)
- The eigenspace of the eigenvalue μ is one-dimensional and contains an eigenvector v whose entries v_i are all positive.
- The limit $\lim_{n\to\infty} \dfrac{1}{\mu^n} A^n$ exists and equals a matrix whose columns are all scalar multiples of v.

Graphical illustration If you have never seen Theorem 1 before, you might be mystified how so many strong statements can be concluded simply from the positivity assumption. I'd like to give you an idea of the reasoning that leads to these statements, without writing out a formal proof, through the following simple example of a 2×2 positive matrix.

```
[17]: A = np.array([[0.1, 0.9],
                    [0.6, 0.4]])
      ew, ev = eig(A)
      ew
```

```
[17]: array([-0.5,  1. ])
```

We see that the dominant eigenvalue is 1 in this case. To get an idea of why A^n converges, as claimed in the theorem, see what happens when we multiply A by A in terms of the first and second columns (A_0 and A_1) of $A = [A_0, A_1]$:

$$A^2 = A[A_0, A_1] = [AA_0, AA_1]$$

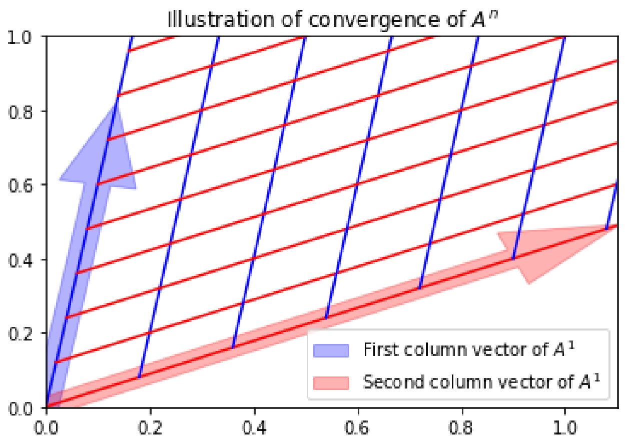
When A is multiplied by a positive vector the result is a linear combination of the columns of A with positive combination coefficients. This is best seen using pictures. To this end, we define a function below that plots the columns of A (as two thick arrows) and the region in between (a two-dimensional cone) using criss-cross lines.

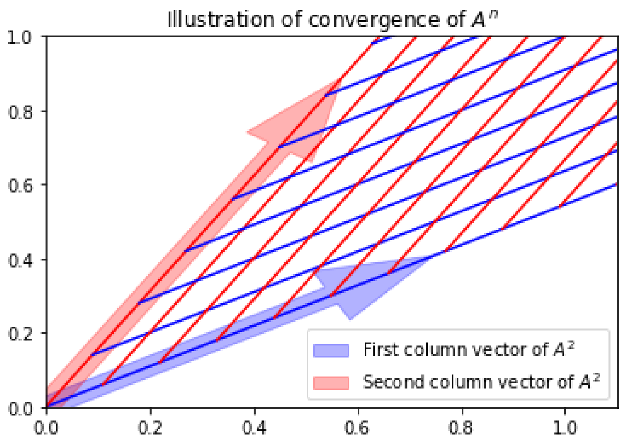
Using it we see what happens to the cone region under repeated application of A.

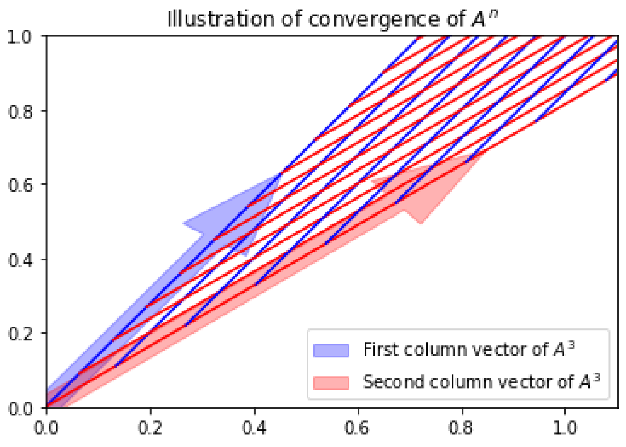
```
[18]: import matplotlib.pyplot as plt
      %matplotlib inline

      def plotcone(A0, A1, xlim=(0,1.1), ylim=(0,1.), matlabel='$A$',␣
       ↪tt='Illustration of convergence of $A^n$'):
          t = np.linspace(0, 3, num=100)
          gridline0 = t[:, np.newaxis] * A0
          gridline1 = t[:, np.newaxis] * A1
          fig = plt.figure(); ax = plt.gca()
          for i in range(20):
              ax.plot(gridline0[:, 0], gridline0[:, 1], 'b')
              ax.plot(gridline1[:, 0], gridline1[:, 1], 'r')
              gridline0 += (1/5) * A1
              gridline1 += (1/5) * A0
          ax.set_xlim(xlim);  ax.set_ylim(ylim)
          ax.set_title(tt)
          a0 = ax.arrow(0, 0, A0[0], A0[1], width=0.05, color='blue', alpha=0.3)
          a1 = ax.arrow(0, 0, A1[0], A1[1], width=0.05, color='red', alpha=0.3)
          plt.legend((a0, a1), ('First column vector of '+matlabel, 'Second␣
       ↪column vector of '+matlabel), loc='lower right');
```

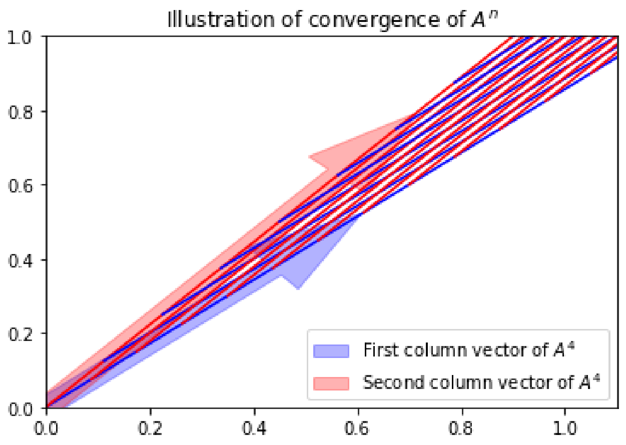
```
[19]: M = A.copy()
      for i in range(5):  # plot the cone between columns for each matrix power
          A0 = M[:, 0]
```

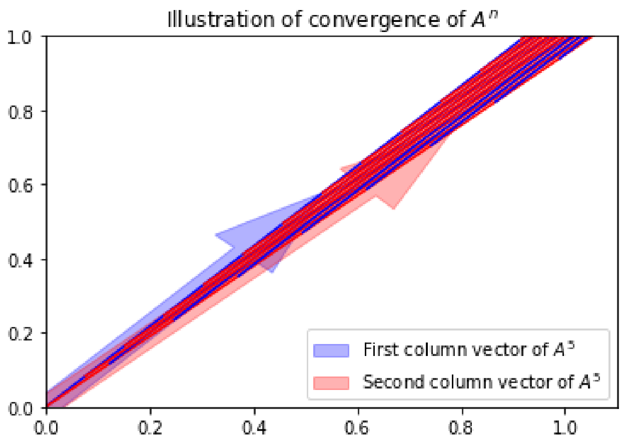
```
    A1 = M[:, 1]
    plotcone(A0, A1, matlabel='$A^'+str(i+1)+'$')
    M = M @ A
```









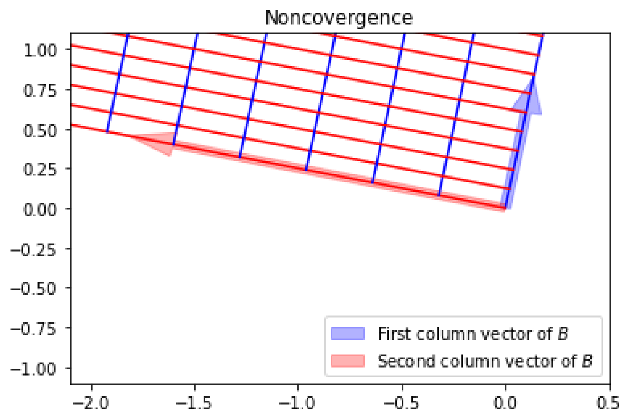


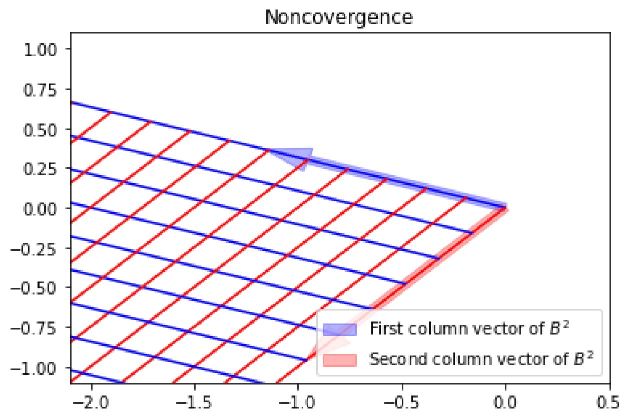
As you can see, repeated application of A eventually squeezes the cone region to a linear region. The vectors in the boundary of the region getting squeezed are the columns of A^n as $n \to \infty$, so you have just seen a pictorial illustration of existence of the limit of A^n, and also of the theorem's claim that in the limit, the columns become multiples of a single vector. Moreover, the limiting linear region in the figure should remain unaltered under further applications of A, so it must represent an eigenspace of A. Note also that all of this happens in the positive quadrant of the plane, so obviously the squeezed in line is the span of a vector v with positive entries, so this should be the positive eigenvector mentioned in the theorem.

This squeezing phenomena happens because A has positive entries. If A had negative entries, the region between its column vectors need not get so squeezed and can dance all over the place preventing convergence of its powers. Considering another matrix, also with dominant eigenvalue 1, but now with a negative entry, you get the picture:

```
[20]: B = np.array([[0.1, -1.6],    # change sign of one entry of A to get B
                    [0.6, 0.4]])
      plotcone(B[:,0], B[:,1], xlim=(-2.1, 0.5), ylim=(-1.1, 1.1),␣
       ↪matlabel='$B$', tt='Noncovergence')
      B = B @ B
```

```
plotcone(B[:,0], B[:,1], xlim=(-2.1, 0.5), ylim=(-1.1, 1.1),␣
↪matlabel='$B^2$', tt='Noncovergence')
```





Any overlaps between the cones disappear as you take further powers.

This completes our graphical illustration of the connection between positivity of entries, the convergence of matrix powers, and the resulting capture of a positive eigenvector by successively squeezing cones. In the next subsection, where we apply Perron's theorem to transition matrices, we will be more rigorous, and yet, use nothing more than what you already know from your linear algebra prerequisites.

Application to stochastic matrices In this subsection, *we consider a Markov chain whose transition matrix $P = (p_{ij})$ has positive entries.* Accordingly, there is a dominant positive eigenvalue μ and corresponding eigenvector v with positive components: $Pv = \mu v$. Normalizing v such that its maximum entry v_i is one, the ith equation of the system $Pv = \mu v$ reads as

$$\sum_{j=1}^{N} p_{ij} v_j = \mu v_i = \mu.$$

We also have

$$\sum_{j=1}^{N} p_{ij} v_j \leq \sum_{j=1}^{N} p_{ij} = 1.$$

Putting these together, we conclude that the dominant eigenvalue satisfies $\mu \le 1$. But any transition matrix P always has 1 as an eigenvalue. This is because the fact that its rows sums are one

$$\sum_{j=1}^{N} p_{ij} = 1$$

can be rewritten in matrix terms as

$$P \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}.$$

Therefore μ must be 1. Let's highlight this conclusion:

- *$\mu = 1$ is the dominant eigenvalue of any positive transition matrix P and the corresponding eigenvector is the vector whose entries are all ones.*

Of course, Perron's theorem applies to both P and P^t, since both are positive matrices. Since the eigenvalues of P and P^t are the same, we can further say this:

- *$\mu = 1$ is also the dominant eigenvalue of P^t* (but the eigenvector corresponding to eigenvalue 1 may be different for P^t and P).

The theorem also tells us that the limit of $(P^t)^n$ exists. Relating to our discussion of stationary distributions, we conclude:

- *The sequence of probability distributions of a random walk*

  $$x, \quad P^t x, \quad (P^t)^2 x, \quad (P^t)^3 x, \quad \ldots$$

  always converges for Markov chains with $p_{ij} > 0$ and the limit s is independent of the initial distribution x.

The theorem also tells us that the limit of P^n exists, and moreover, the limit matrix must have columns that are scalar multiples of the eigenvector of the dominant eigenvalue: in other words, the columns of $\lim_{n\to\infty} P^n$ must be scalar multiples of the vector of ones. On the other hand, the limit of $(P^t)^n = (P^n)^t$ must have columns that are multiples of the eigenvector s satisfying $P^t s = s$, which we previously called the stationary distribution. Thus, having pinned down the rows and the columns, we make the following conclusion:

- *The limit of P^n as $n \to \infty$ takes the following form*

$$\lim_{n\to\infty} P^n = \begin{bmatrix} s_1 & s_2 & \ldots & s_N \\ s_1 & s_2 & \ldots & s_N \\ \vdots & \vdots & & \vdots \\ s_1 & s_2 & \ldots & s_N \end{bmatrix}.$$

Example A has a positive transition matrix. It provides an instance where all the previous statements can be verified. For example, $\lim_{n\to\infty} P^n$ is approximated by the matrix below which reveals the above pattern of stationary distributions in each row.

```
[21]: matrix_power(PA, 1000)    # P^1000 for Example A
```

```
[21]: array([[0.4, 0.3, 0.3],
             [0.4, 0.3, 0.3],
             [0.4, 0.3, 0.3]])
```

```
[22]: sA                          # the stationary distribution for Example A
```

```
[22]: array([[0.4],
             [0.3],
             [0.3]])
```

XIV.4 PageRank

We shall now define the *PageRank* of vertices on an (unweighted) directed graph with N vertices.

1. First, set $a_{ij} = 1$ if there is an edge from vertex v_i to v_j in the graph and set $a_{ij} = 0$ otherwise.

2. Next, let
$$m_i = \sum_{k=1}^{N} a_{ik}.$$
If m_i is 0, then the ith vertex is a *dangling node* (which has no outgoing edges). Define
$$w_{ij} = \begin{cases} \dfrac{a_{ij}}{m_i} & \text{if } m_i > 0, \\ \dfrac{1}{N} & \text{if } m_i = 0. \end{cases}$$
These may be thought of as weights on a directed edge from v_i to v_j if the edge exists (if not, the weight is zero). The weight w_{ij} may also be viewed as providing equal probabilities to all outgoing edges from v_i.

3. Now that we have a weighted directed graph, we may associate it to a Markov chain, setting transition probabilities to w_{ij}, but hold on: if we do so, a random walker W on the graph can get stuck in vertices with no outgoing edges or in cycles within the graph. (This is certain to happen on graphs as complex as the internet.) To avoid this situation, one sets a *restart probability* $0 < r \ll 1$ with which W is allowed to jump from one vertex to any other vertex in the graph. (Page called $1 - r$ the *damping* factor.)

4. Finally, set the Markov chain transition probabilities by
$$p_{ij} = \frac{r}{N} + (1 - r) w_{ij}.$$

The PageRank of a vertex is defined to be the value of the stationary probability distribution at that vertex obtained using the above p_{ij} as the transition probabilities.

Note that the transition matrix (p_{ij}) defined above is a positive matrix. Hence, due to Perron's theorem, and our prior discussion on its application to stochastic matrices, the limit of the probability distributions

$$x, \quad P^t x, \quad (P^t)^2 x, \quad (P^t)^3 x, \quad \ldots$$

exists, is equal to the stationary probability distribution, which we have just decided to call PageRank. In particular, PageRank is independent of the starting distribution x of the random walk. Furthermore, we may also arrive at the interpretation of the PageRank of a graph vertex as the limiting probability that a relentless random walker visits that vertex.

Here is a simple implementation for small graphs using the same notation (a_{ij}, w_{ij}, p_{ij}) as above. (Think about what would need to change for a giant graph like the internet. We'll consider big graphs in later exercises.)

[23]:
```
def pagerank(a, r):
    """ Return pagerank based on adjacency matrix "a" (square matrix
    of 0s or 1s) and given restart probability "r". Use only for small
    dense numpy matrices a. """

    m = a.sum(axis=1)
    dangling = (m==0)
    m[dangling] = 1

    w = (1 / m[:, np.newaxis]) * a
    w[dangling, :] = 1 / a.shape[0]

    p = (1-r) * w  +  (r / a.shape[0])
    ew, ev = eig(p.T)
    s = ev[:, abs(ew-1) < 1e-15].real
    return  s / s.sum()
```

Let's quickly consider a small example to illustrate PageRank.

Example D

[24]:
```
#               0  1  2  3  4  5  6  7  8   (Adjacency Matrix of the above␣
 ↪graph)
A = np.array([[0, 1, 0, 0, 1, 0, 0, 0, 0],  # 0
              [0, 0, 0, 0, 1, 0, 0, 0, 0],  # 1
              [0, 0, 0, 0, 1, 0, 0, 0, 0],  # 2
              [0, 0, 0, 0, 1, 0, 0, 0, 0],  # 3
              [0, 0, 0, 0, 0, 0, 1, 0, 0],  # 4
              [0, 0, 0, 0, 1, 0, 0, 0, 0],  # 5
              [0, 0, 0, 0, 0, 1, 0, 0, 0],  # 6
              [0, 0, 0, 0, 0, 1, 0, 0, 0],  # 7
              [0, 0, 0, 0, 0, 1, 0, 0, 0]]) # 8
```

[25]:
```
pagerank(A, 0.1)
```

```
[25]: array([[0.01111111],
              [0.01611111],
              [0.01111111],
              [0.01111111],
              [0.32328823],
              [0.30297458],
              [0.30207052],
              [0.01111111],
              [0.01111111]])
```

Notice from the output that V_4 is ranked highest. A vertex to which many other vertices points to usually get a higher PageRank. Notice also how a vertex to which a highly ranked vertex points to inherits a high PageRank: this is the case with vertex V_6. Vertex V_5 is also highly ranked because it has its own cluster of vertices (V_6, V_7, V_8) pointing to it which included one highly ranked vertex V_6.

It is instructive to look at how PageRank changes as the restart probability is decreased to 0:

```
[26]: pagerank(A, 0.01)
```

```
[26]: array([[0.00111111],
              [0.00166111],
              [0.00111111],
              [0.00111111],
              [0.33239996],
              [0.33019631],
              [0.33018707],
              [0.00111111],
              [0.00111111]])
```

```
[27]: pagerank(A, 0.0)
```

```
[27]: array([[-0.        ],
              [-0.        ],
              [-0.        ],
              [-0.        ],
              [ 0.33333333],
              [ 0.33333333],
              [ 0.33333333],
              [-0.        ],
              [-0.        ]])
```

This identifies the cycle where the random walker would end up if it were not for the restart mechanism.

On internet search results As I mentioned above, PageRank was proposed specifically to order the world wide web. In view of our previous discussion, when applied to the giant graph of the internet, the PageRank of a webpage can be interpreted as the steady state probability that a random web surfer, following hyperlinks from page to page (with infinite dedication and with no topical preference), is at that webpage.

When a user types in a search query, a search engine must first be able to mine all the webpages relevant to the query from its database. (PageRank does not help with this task.) Then, it must present these pages in some order to user. If the search engine has already computed a ranking of relative importance of each webpage, then it can present the results to the user according to that ranking. This is where PageRank helps. It does require the search engine to solve for a giant eigenvector (with billions of entries) to compute PageRank on the entire world wide web. The results of this computation (which cannot be done in real time as the user searches) are stored by the search engine. The stored ranking is then used to order the results presented to the user. There are reports that Google does this a couple of times a year (but I don't know how to verify this).

XIV.5 Perron-Frobenius theorem

Georg Frobenius generalized Perron's theorem to nonnegative matrices. The key discovery of Frobenius was that although many of the nice properties of positive matrices fail to hold for general non-negative matrices, they continue to hold for non-negative matrices whose directed graph exhibits a "nice" property. Recall that the directed graph of an $N \times N$ matrix $A = (a_{ij})$ is a graph with vertices $1, 2, \ldots, N$ which has a directed edge from vertex i to vertex j whenever a_{ij} is nonzero. The "nice" property is the following.

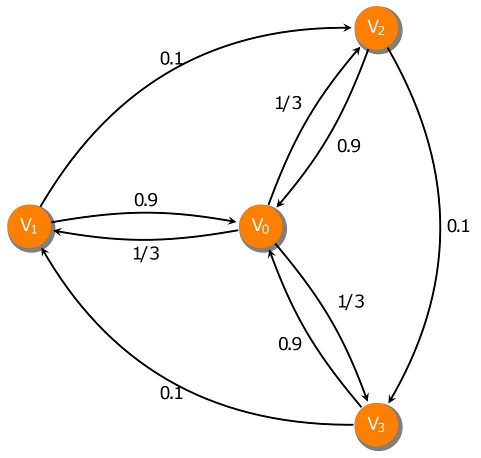
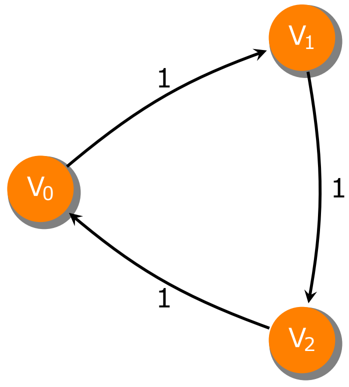
A square matrix is called *irreducible* if its directed graph is such that there is a path made of directed edges from any vertex to any other vertex.

Theorem 3. (Perron-Frobenius) The following statements hold for any irreducible $N \times N$ matrix $A = (a_{ij})$ whose entries satisfy $a_{ij} \geq 0$ are all non-negative.

- The maximum μ of the absolute value of all eigenvalues of A is an eigenvalue of A.
- The eigenspace of the eigenvalue μ is one-dimensional and contains an eigenvector v whose entries v_i are all positive.
- The limit $\lim\limits_{k\to\infty} \dfrac{1}{k} \sum\limits_{n=0}^{k-1} \dfrac{1}{\mu^n} A^n$ exists and equals a matrix whose columns are all scalar multiples of v.

Note the main differences in Theorem 3 in comparison to Theorem 2:

- Unlike positive matrices, now there might be more than one eigenvalue whose absolute value is μ.
- Unlike positive matrices, we can no longer assert that limit A^n/μ^n exists, only that the limit of averages of A^n/μ^n exists.

Example B	Example C
	
Convergent: $\lim_{n\to\infty}(P^t)^n x = s$	Not convergent:$\lim_{n\to\infty}(P^t)^n$ doesn't exist
Stationary distribution:	Stationary distribution:
$s = \begin{bmatrix}0.474\\0.175\\0.175\\0.175\end{bmatrix}$	$s = \begin{bmatrix}1/3\\1/3\\1/3\end{bmatrix}$

Reconsider Examples B & C Note that the Markov chains in both Examples B and C have non-negative transition matrices that are irreducible: the irreducibility is obvious by looking at the previous figures of the digraphs for each example. Hence the Perron-Frobenius theorem applies to both. Therefore, in both cases we may conclude that the stationary distribution s is the limit of the averages

$$\frac{x + P^t x + (P^t)^2 x + \cdots + (P^t)^k x}{k+1}$$

as $k \to \infty$, for any starting distribution x. Although $(P^t)^n$ does not converge for Example C, these averages do. For Example B, we observed convergence for $(P^t)^n$ so, of course, the averages also converge.

Let me conclude with a few words on nomenclature. In the statistics literature, a Markov chain is called an *ergodic Markov chain* if it is possible to arrive at any state from any other state in one or more steps. In other words, a Markov chain is ergodic if its digraph has paths made of directed edges from any vertex to any other vertex. This concept is therefore equivalent to its transition matrix being *irreducible*, and indeed, several texts use the adjective irreducible instead of ergodic when studying such Markov chains. In computer science and in graph theory, a directed graph whose vertices can be connected by a path of directed edges is called *strongly connected,* yet another name for the same concept. We have seen several such instances of distinct names for essentially the same concept in this and the previous activity. While it may be a nuisance that the names are not standardized, it's not surprising for a concept that emerged as important in different disciplines to get distinct names attached to it; it may even speak to the universality of the concept.

XV

Supervised learning by regression

May 21, 2020

Machine learning refers to mathematical and statistical techniques to build *models of data.* A program is said to *learn* from data when it chooses a model or adapts tunable model parameters to observed data. In broad strokes, machine learning techniques may be divided as follows:

- *Supervised learning*: Models that can predict labels based on labeled training data
  - *Classification*: Models that predict labels as two or more discrete categories
  - *Regression*: Models that predict continuous labels
- *Unsupervised learning*: Models that identify structure in unlabeled data
  - *Clustering*: Models that detect and identify distinct groups in the data
  - *Dimensionality reduction*: Models that identify lower-dimensional structure in higher-dimensional data

In this activity, we focus on supervised learning. Note the two further subdivisions mentioned above within the category of supervised learning, and here are two examples within each for further orientation:

- Classification example: identify an email as spam or not (discrete label) based on counts of trigger words.
- Regression example: predict the arrival time (continuous label) of a streetcar at a station based on past data.

We shall further focus on *regression* in this activity. Regression addresses an age-old fitting problem: given a set of data, find a line (or a curve, or a surface, or a hypersurface in higher dimensions) that approximately fits the data. The equation of the line, in the machine learning language, is the *model* of the data that has been "learnt." The model can then "predict" the values, i.e., "labels" at points not covered by the original data set. Finding equations of curves that pass through a given set of points is the problem of *interpolation*, which goes at least as far back as Newton (1675). The fitting problem in regression, also known at least as far back as Gauss (1809), is a relaxed version of the interpolation problem in that it does not require the curves to pass through the given data, and is generally more suitable to handle noisy data. These days, when machine learning comes at you with the brashness of an overachieving new kid on the block, it is not fashionable to view the subject from the perspective of established mathematical techniques with rich histories. Instead, it has somehow become more fashionable to view machine learning as some sort of new AI miracle. Please do not expect any miracles here.

XV.1 Linear Regression

Let's start from the linear regression in a form you have seen previously: given data points (x_i, f_i), $i = 0, 1, \ldots, N$, fit a linear equation

$$f(x) = a_0 + a_1 x$$

to the data, in such a way that the error in the fit

$$e = \sum_{i=0}^{N} |f(x_i) - f_i|^2$$

is minimized. Since the quantity on the right is a sum of squares, this is called the *least-squares* error. It is easy to solve this minimization problem. Writing

$$Y^{\text{data}} = \begin{bmatrix} f_0 \\ \vdots \\ f_N \end{bmatrix}, \qquad Y^{\text{fit}} = \begin{bmatrix} f(x_0) \\ \vdots \\ f(x_N) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & x_0 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} a_0 \\ a_1 \end{bmatrix}}_{a}$$

the error e can also be expressed as $e = \|Y^{\text{fit}} - Y^{\text{data}}\|^2 = \|Xa - Y^{\text{data}}\|^2 = (Xa - Y^{\text{data}})^t (Xa - Y^{\text{data}})$. Now, either from linear algebra, or from calculus, one concludes that e is minimized when

$$a = (X^t X)^{-1} X^t Y^{\text{data}}.$$

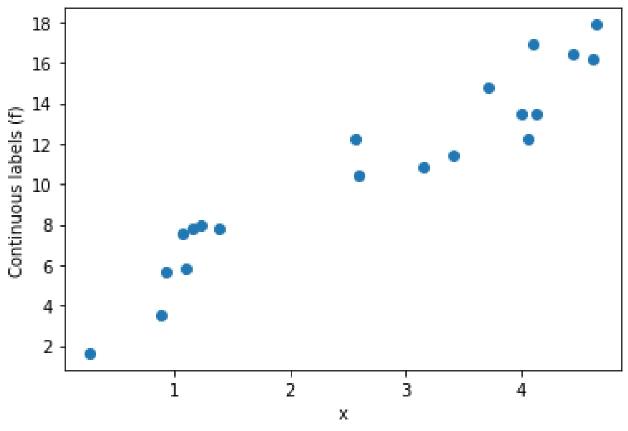
This is the *least-squares solution to the linear regression problem.*

In the machine learning language, - f_i are (continuous) "labels", - the "model" is the linear formula $a_0 + a_1 x$, - the "labeled training data" is (x_i, f_i), and - the "predictions" are values of $f(x)$ at various x-values.

Here is an example.

[1]:
```python
import numpy as np
from numpy.linalg import inv
%matplotlib inline
import matplotlib.pyplot as plt
rng = np.random.default_rng(123)
```

[2]:
```python
x = 5 * rng.random(20)
f = 3 * x + 5 * rng.random(20)
plt.scatter(x, f); plt.xlabel('x'); plt.ylabel('Continuous labels (f)');
```

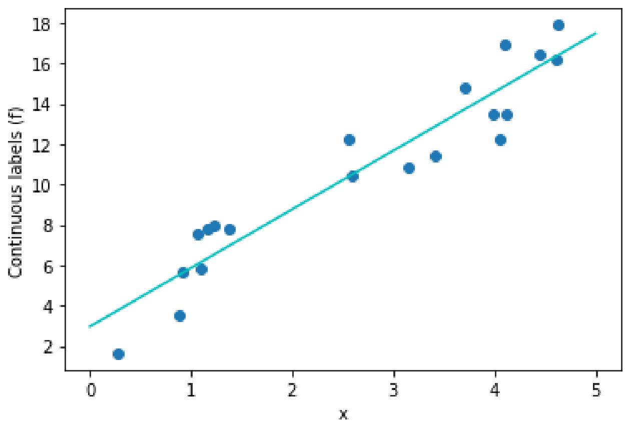


The data has a linear trend, so let's try to fit a line to it using linear regression formula we derived above.

```
[3]: X = np.array([np.ones(len(x)), x]).T
     a = inv(X.T @ X) @ X.T @ f                  # Create the "model"
```

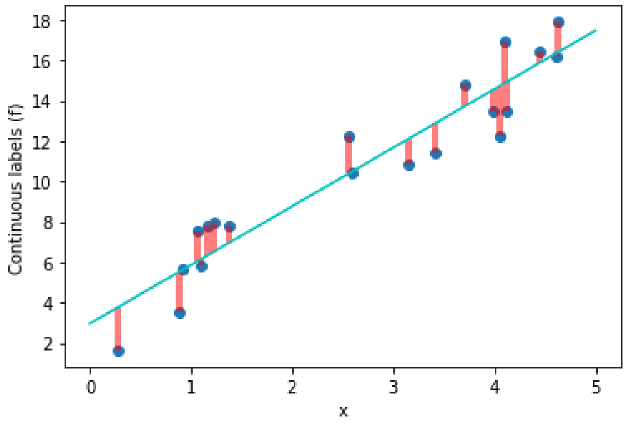
```
[4]: x_predict = np.linspace(0, 5, num=100)
     f_predict = a[0]  + a[1] * x_predict        # "Predict" using the model
```

```
[5]: plt.scatter(x, f)
     plt.xlabel('x'); plt.ylabel('Continuous labels (f)');
     plt.plot(x_predict, f_predict, 'c');
```



To have a visual representation of the error that is minimized by this line, we can plot line segments (the red thick lines below) whose sum of squared lengths is what we minimized:

```
[6]: from matplotlib.collections import LineCollection
     fp = X @ a
     plt.scatter(x, f)
     lc = LineCollection([[(x[i], f[i]), (x[i], fp[i])]
                                     for i in range(len(x))], color='r', linewidth=4,␣
     ↪alpha=0.5)
     plt.gca().add_collection(lc)
     plt.xlabel('x'); plt.ylabel('Continuous labels (f)');
     plt.plot(x_predict, f_predict, 'c');
```



Let us save there results for later comparison.

[7]:
```
linear_example = {'data': [x, f], 'model': a}
```

XV.2 Higher dimensions

The process of linear regression is very similar in higher dimensions. To fit some given data f_i on $N+1$ points $\vec{x}_i$, each of which are m-dimensional, we express the points in coordinates $\vec{x}_i = (x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(m)})$. The model now is

$$f(x^{(1)}, x^{(2)}, \ldots, x^{(m)}) = a_0 + a_1 x^{(1)} + \cdots + a_m x^{(m)}.$$

Exactly the same algebra as before yields the *same solution formula*

$$a = (X^t X)^{-1} X^t Y^{\text{data}},$$

the only difference now being that

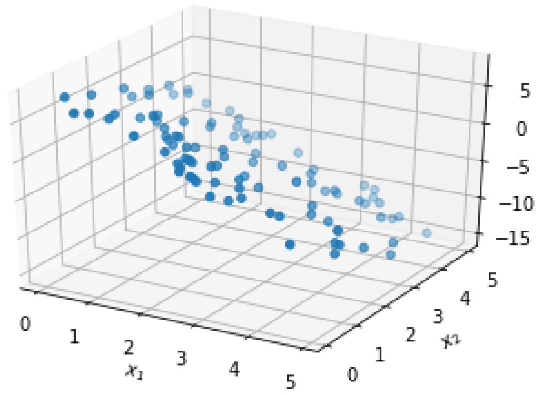
$$a = \begin{bmatrix} a_0 \\ \vdots \\ a_m \end{bmatrix}, \qquad X = \begin{bmatrix} 1 & x_0^{(1)} & x_0^{(2)} & \cdots & x_0^{(m)} \\ 1 & x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ \vdots & \vdots & & & \\ 1 & x_N^{(1)} & x_N^{(2)} & \cdots & x_N^{(m)} \end{bmatrix}$$

Here is a 3D example, where we can still attempt to visualize:

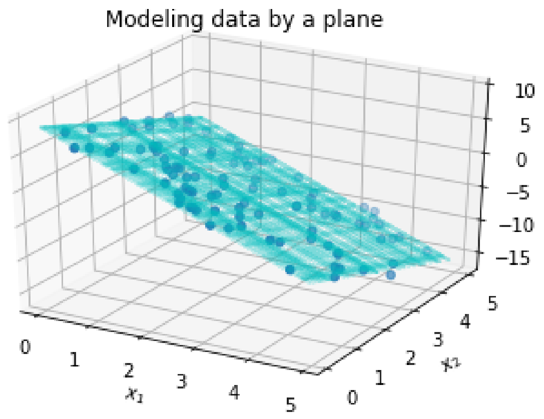
```
[8]: x1 = 5 * rng.random(100)
     x2 = 5 * rng.random(100)
     f = 10 - (3*x1 + 2* x2 +  2 * rng.random(100))
```

```
[9]: X = np.array([np.ones(len(x1)), x1, x2]).T
     a = np.linalg.inv(X.T @ X) @ X.T @ f
```

```
[10]: from mpl_toolkits.mplot3d import Axes3D
      ax = plt.figure().gca( projection='3d')
      ax.scatter(x1, x2, f)
      ax.set_xlabel('$x_1$'); ax.set_ylabel('$x_2$');
```



```
[11]: ax = plt.figure().gca( projection='3d')
      ax.set_xlabel('$x_1$'); ax.set_ylabel('$x_2$')
      xx1, xx2 = np.meshgrid(x1, x2)
      zz = a[0] + a[1] * xx1 + a[2] * xx2
      ax.plot_wireframe(xx1, xx2, zz, color='c', alpha=0.2)
      ax.scatter(x1, x2, f); ax.set_title('Modeling data by a plane');
```



We save these results for later examination.

[12]:
```
planar_example = {'data': [np.array([x1, x2]).T,  f], 'model': a}
```

XV.3 Curve fitting

If you know that your data is exponential, you might get better results by fitting with exponentials instead of linear functions. The "linear" regression process can be adapted to use exponentials, or gaussians, or indeed any basis functions you feel are particularly appropriate for your data set. The linearity in "linear" regression refers to the linear dependence of the model on the data (and has nothing to do which whether your model f is linear or not).

Deriving the general formula is done by the same method. Using basis functions $\phi_j(\vec{x})$, we can fit given data f_i on $N+1$ points $\vec{x}_i$ using the model

$$f(\vec{x}) = a_0\phi_0(\vec{x}) + a_1\phi_1(\vec{x}) + \cdots + a_m\phi_m(\vec{x}).$$

Again, the previous algebra yields the *same solution formula*

$$a = (X^t X)^{-1} X^t Y^{\text{data}},$$

for the model parameters that provide the minimizer of

$$e = \sum_{i=0}^{N} |f(\vec{x}_i) - f_i|^2.$$

The only difference now is that

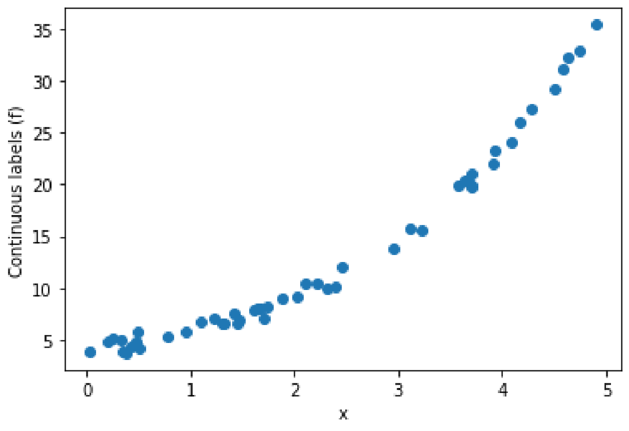
$$a = \begin{bmatrix} a_0 \\ \vdots \\ a_m \end{bmatrix}, \qquad X = \begin{bmatrix} \phi_0(\vec{x}_0) & \phi_1(\vec{x}_0) & \cdots & \phi_m(\vec{x}_0) \\ \phi_0(\vec{x}_1) & \phi_1(\vec{x}_1) & \cdots & \phi_m(\vec{x}_1) \\ \vdots & \vdots & & \\ \phi_0(\vec{x}_N) & \phi_1(\vec{x}_N) & \cdots & \phi_m(\vec{x}_N) \end{bmatrix}.$$

Here is an example where we fit a quadratic curve to a simple one-dimensional data set, i.e., here

$$f(x) = a_0 + a_1 x + a_2 x^2$$

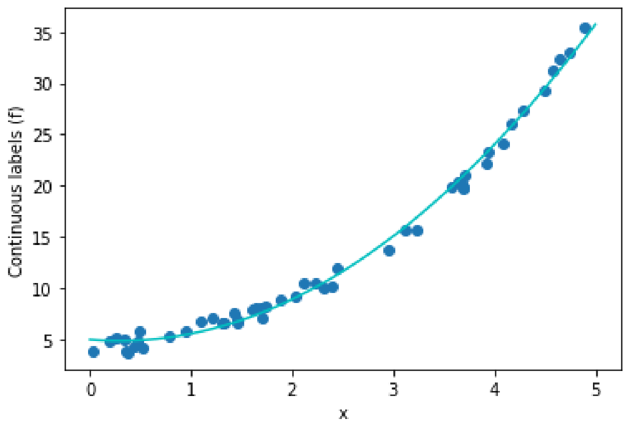
and the a's are found by the above formula.

[13]:
```
x = 5 * rng.random(50)
f = 3 * np.exp(x/2) + 2 * rng.random(50)
plt.scatter(x, f); plt.xlabel('x'); plt.ylabel('Continuous labels (f)');
```



[14]:
```
phi0 = np.ones(len(x))
phi1 = x
phi2 = x**2

X = np.array([phi0, phi1, phi2]).T
a = np.linalg.inv(X.T @ X) @ X.T @ f
```

[15]:
```
xcurve_predict = np.linspace(0, 5, num=500)
phi0 = np.ones(len(xcurve_predict))
phi1 = xcurve_predict
phi2 = xcurve_predict**2

fcurve_predict = a[0] * phi0  + a[1] * phi1  + a[2] * phi2
plt.scatter(x, f)
plt.xlabel('x'); plt.ylabel('Continuous labels (f)');
plt.plot(xcurve_predict, fcurve_predict, 'c');
```



If we had attempted to fit a straight-line through the data, then we would not have gotten such a close fit. Another way of saying this in the prevalent terminology is that linear features *underfit* this data, or that the linear model has *high bias.* Saving this example also for later, we continue.

```
[16]: curve_example = {'data': [x, f], 'model': a, 'type': 'quadratic'}
```

XV.4 The module `scikit-learn`

All the regression computations we did above can be done using the module `scikit-learn`. Of course, the formulas above were simple and easy to implement. The power of `scikit-learn` is not in its linear regression implementation, but rather, in the vast range of many other ready-made facilities it provides under a unified user interface. When faced with a package that attempts to do so many things, it's a good entry strategy to confirm that it behaves as we expect in situations we know. This was our purpose in using the simple regression as an entry point into `scikit-learn`.

Let's check if our first-principles computation of regression solutions match what `scikit-learn` produces.

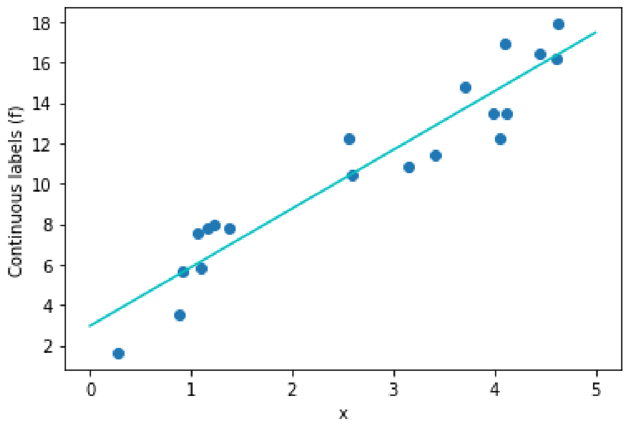
```
[17]: from sklearn.linear_model import LinearRegression
      model = LinearRegression(fit_intercept=True)
```

We can now fit data to this `model` using the `fit` method. Let's fit the same data we used in the first example of this activity.

```
[18]: x, f = linear_example['data']    # Recall the saved data from the first␣
       ↪example
      model.fit(x[:, np.newaxis], f)   # Training step
```

```
[18]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,␣
       ↪normalize=False)
```

```
[19]: xfit = np.linspace(0, 5, num=100)
      ffit = model.predict(xfit[:, np.newaxis])    # Prediction step
      plt.scatter(x, f);
      plt.xlabel('x'); plt.ylabel('Continuous labels (f)');
      plt.plot(xfit, ffit, 'c');
```



Clearly we seem to be getting the same result. We can confirm the results are *exactly* the same by digging into the solution components within the `model` object, as seen below. (Recall that in $f(x) = a_0 + a_1 x$, the coefficient a_0 is called the *intercept*.)

```
[20]: model.intercept_, model.coef_
```

```
[20]: (2.9548137487468367, array([2.90310325]))
```

This is exactly the same as the values we solved for previously:

```
[21]: linear_example['model']
```

```
[21]: array([2.95481375, 2.90310325])
```

Higher dimensions The fitting process in `scikit-learn` is similar in higher dimensions.

```
[22]: x12, f = planar_example['data']
      model.fit(x12, f)
```

```
[22]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,␣
       ↪normalize=False)
```

```
[23]: model.intercept_, model.coef_
```

```
[23]: (9.167204926561409, array([-3.03592026, -2.03048875]))
```

This matches our previously computed results:

```
[24]: planar_example['model']
```

```
[24]: array([ 9.16720493, -3.03592026, -2.03048875])
```

XV.5 More terminology

Of course, regression for curve fitting is also possible in `scikit-learn`. The difference now is that here you begin to see how things would get easier if you learn their language.

Scikit-learn uses the word *estimator* for models in machine learning. In the module, estimators are python objects that implement the methods `fit` and `predict`. We have already seen both methods in the context of the above regression examples. Additional terminology we should know include *transformer* (objects which can map/transform data into some other form) and *pipeline* (a sequence of transformers followed by an estimator).

The term *feature* is probably the most difficult one to pin down as it is used for too many things: data attributes, elements of a data row, columns of a data array, the range of a function mapping some data values, etc. When a data set is being fitted with some basis functions, linear or not, the word feature is used to refer to the basis. In fact, the process of selecting such basis functions is an example of *feature engineering*. More generally, feature engineering is any process by which raw information (data) is converted into numbers or other mathematical objects, things inside a *feature matrix.* Tidy data in a feature matrix has each *variable/feature in a column* and each *observation/sample in a row*.

```
[25]: from sklearn.pipeline import make_pipeline
      from sklearn.preprocessing import PolynomialFeatures
```

Using polynomial *feature*s, we create quadratic basis functions.

```
[26]: q = PolynomialFeatures(3, include_bias=False)
```

Here is an example of a *transform*(er):

```
[27]: data = np.array([5, 7, 9])[:, np.newaxis]
      q.fit_transform(data)
```

```
[27]: array([[  5.,   25., 125.],
             [  7.,   49., 343.],
             [  9.,   81., 729.]])
```

As you can see the feature q performed the data transformation

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_N \end{bmatrix} \longrightarrow X = \begin{bmatrix} \phi_0(\vec{x}_0) & \phi_1(\vec{x}_0) & \cdots & \phi_m(\vec{x}_0) \\ \phi_0(\vec{x}_1) & \phi_1(\vec{x}_1) & \cdots & \phi_m(\vec{x}_1) \\ \vdots & \vdots & & \\ \phi_0(\vec{x}_N) & \phi_1(\vec{x}_N) & \cdots & \phi_m(\vec{x}_N) \end{bmatrix}$$

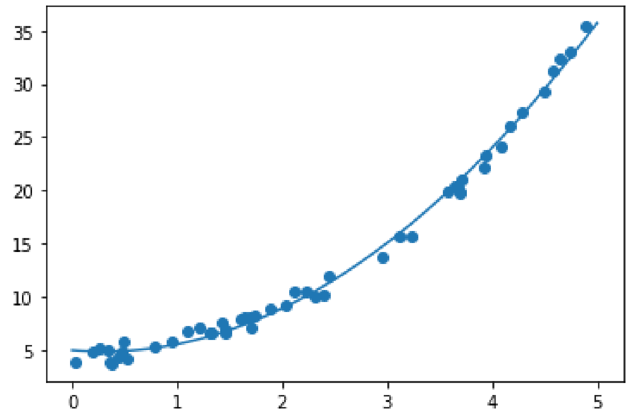
for $\{\phi_i(x)\} = \{x, x^2, x^3\}$.

Curve fitting The point of view taken by scikit-learn for curve fitting is that it is a process obtained by applying the linear regression formula after applying the above transformer. Therefore, one can implement it using a pipeline object where this transformer is chained to a linear regression estimator. Here is how this idea plays out for the curve-fitting example we saw previously.

```
[28]: x, y = curve_example['data']   # load data from the prior example

      # make model/pipeline and fit the data to it:
      quadratic_model = make_pipeline(PolynomialFeatures(2), LinearRegression())
      quadratic_model.fit(x[:, np.newaxis], y)
```

```
[28]: Pipeline(memory=None,
               steps=[('polynomialfeatures',
                       PolynomialFeatures(degree=2, include_bias=True,
                                          interaction_only=False, order='C')),
                      ('linearregression',
                       LinearRegression(copy_X=True, fit_intercept=True,␣
       ↪n_jobs=None,
                                        normalize=False))],
               verbose=False)
```

[29]:
```
yfit = quadratic_model.predict(xfit[:, np.newaxis])
plt.scatter(x, y)
plt.plot(xfit, yfit);
```



We can cross-check that the model parameters are exactly the same after fitting by examining the `LinearRegression` object in the quadratic model pipeline:

[30]:
```
quadratic_model.named_steps
```

[30]:
```
{'polynomialfeatures': PolynomialFeatures(degree=2, include_bias=True,
interaction_only=False,
                     order='C'),
 'linearregression': LinearRegression(copy_X=True, fit_intercept=True,
n_jobs=None, normalize=False)}
```

[31]:
```
quadratic_model.named_steps['linearregression'].intercept_
```

[31]:
```
4.963796378670274
```

[32]:
```
quadratic_model.named_steps['linearregression'].coef_
```

[32]:
```
array([ 0.        , -0.82755299,  1.39570211])
```

These match our previously computed results for quadratic fit:

[33]:
```
curve_example['model']  # previously saved results from first principles
```

[33]:
```
array([ 4.96379638, -0.82755299,  1.39570211])
```

To conclude, we have built some confidence in scikit-learn's abilities under the hood. There is plenty of material online, including [JV-H], on how to use scikit-learn and other machine learning packages, and on important pitfalls such as overfitting. However, it may be a bit harder to find out the mathematics behind each software facility: the documentation is designed for quick users in a rapidly changing field, and therefore understandably does not get into the math. This may not be comforting to you as students of mathematics, so my focus here and in the next few lectures is to connect these software tools with the mathematics you know.

XVI Unsupervised learning by PCA

May 27, 2020

Recall from the previous lecture that *unsupervised learning* refers to machine learning models that identify structure in unlabeled data. In this activity, we study **Principal Component Analysis (PCA)** which is a commonly used technique in unsupervised learning, often used for discovering structure in high-dimensional data, and for dimensionality reduction.

In this activity, I will extensively draw upon what you studied in some earlier activities. In particular, I will try to detail the connections between PCA and SVD, the differences in the jargon, highlight the distinctions between PCA and regression, and illustrate how unsupervised machine learning is different from supervised machine learning.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib.colors as colors
from matplotlib.collections import LineCollection
import matplotlib.cm as cm
from sklearn.decomposition import PCA
from scipy.linalg import svd
from numpy.linalg import norm
rng = np.random.default_rng(13)
```

XVI.1 Definitions

- Given a *one*-dimensional data vector $x = [x_1, x_2, \ldots, x_m]^t$, its *mean*, or **sample mean** is
$$\bar{x} = \frac{1}{m} \sum_{i=1}^{m} x_i.$$

- Consider a *multi*-dimensional $m \times n$ data array X representing

  m samples/observations/rows for n variables/features/columns.

  The jth column of X, denoted by X_j, represents a number of samples of a single variable. We say that such an X represents **centered data** if the sample mean of X_j is zero for every column j. Let R_i denote the ith row of the data matrix X. We use R_i to define the principal components of any centered data, as follows.

- The **first principal component** of any centered data X is defined as a *unit* vector $v_1 \in \mathbb{R}^n$ that maximizes
$$\sum_{i=1}^{m} (v_1 \cdot R_i)^2.$$
- The **second principal component** of any centered data X (defined when $n \geq 2$) is a unit vector $v_2 \in \mathbb{R}^n$ that is orthogonal to the first principal component v_1 and maximizes
$$\sum_{i=1}^{m} (v_2 \cdot R_i)^2, \qquad \text{subject to } v_1 \cdot v_2 = 0.$$
- The third principal component of X (defined when $n \geq 3$) is a unit vector $v_3 \in \mathbb{R}^n$ that is orthogonal to both v_1 and v_2 while maximizing $\sum_{i=1}^{m} (v_3 \cdot R_i)^2$.

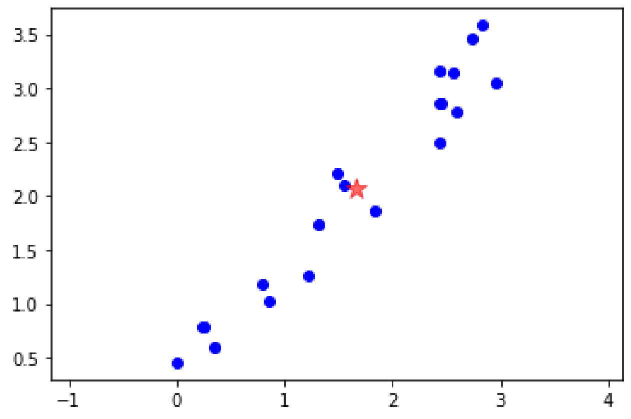
You should now see the pattern to define any number of further principal components. Note that if v is a principal component vector, then $-v$ is also one. Note also that principal components are also often referred to as *principal axes* or *principal directions.*

To understand why these principal components reveal structure in the data, first recall that the dot product of two vectors a and b is maximal when the vectors are collinear: remember that $|a \cdot b| = \|a\| \|b\| |\cos(\theta)|$ where θ is the angle between a and b, and $|\cos(\theta)|$ is maximal when θ is 0 or integer multiples of π. Hence the first principal component v_1 may be interpreted as the vector that is "most collinear" with all the rows/observations/samples R_i. A dependency between multiple variables/features/columns hidden inside the many samples/observations in X can thus be brought out using v_1. While v_1 gives the dominant dependency, the later principal components reveal further dependencies in spaces orthogonal to the previous principal components. You should now begin to see why PCA might be able to automatically discover hidden structures in data, one of the primary objectives in *unsupervised machine learning.*

XVI.2 Two-dimensional example

Let's consider a small two-dimensional example where we can graphically visualize all aspects.

[2]:
```
x = 3 * rng.random(20)
y = x + 0.75* rng.random(20)
fig = plt.figure(); ax = plt.gca()
ax.scatter(x, y, color='b')
ax.scatter(x.mean(), y.mean(),  color='r', marker='*', s=150, alpha=0.6);␣
↪ax.axis('equal');
```



We put the data in the form of *m samples/observations/rows* for *n variables/features/columns.*

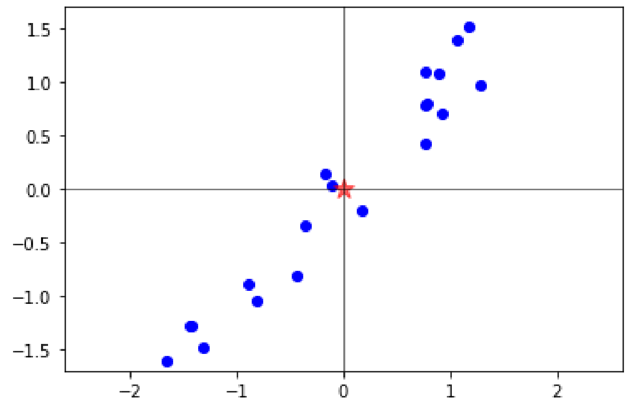
```
[3]: XX = np.array([x, y]).T
     m, n = XX.shape
```

Next, we need to *center the data.* This just means subtracting the mean of each feature/variable. Note that the mean is the marked (as the red star) in above figure.

```
[4]: X = XX - XX.mean(axis=0)
```

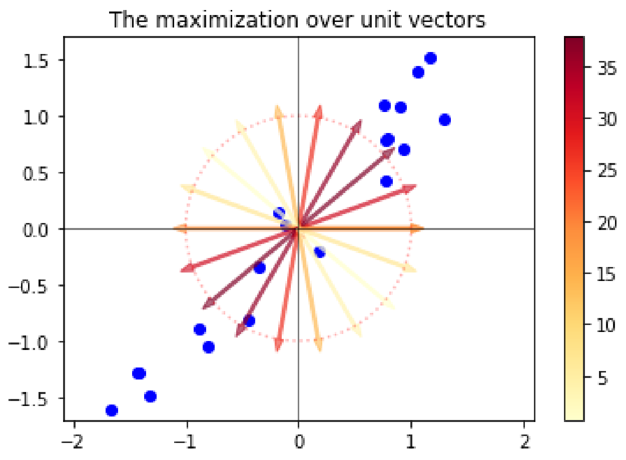
For the visual thinker, centering the data just means moving the origin to the mean (the red star), as illustrated in the next figure.

```
[5]: def plotX(X, ax=None):
         if ax is None: fig = plt.figure(); ax = plt.gca()
         ax.scatter(X[:, 0], X[:, 1], color='b')
         t = np.linspace(-3, 3, 100); o = np.zeros_like(t)
         ax.plot(t, o, 'k', o, t, 'k', linewidth=0.5);
         ax.scatter(0, 0,  color='r', marker='*', s=150, alpha=0.6);
         ax.axis('equal');
         ax.set(xlim=(-1.5,1.5), ylim=(-1.7,1.7));
     plotX(X);
```



Now comes the part that's harder to see, namely the graphical meaning of the maximization problem that defines the first principal component. Consider the figure below where a number of unit vectors are drawn colored.

```
[6]: fig = plt.figure(); ax = plt.gca()
     ax.set_title('The maximization over unit vectors')
     theta = np.linspace(0, 2*np.pi, num=100)      # draw unit circle
     ax.plot(np.cos(theta), np.sin(theta), ':r', alpha=0.3)

     theta = np.linspace(0, 2*np.pi, num=19)
     v = np.array([np.cos(theta), np.sin(theta)]) # unit vectors
     f = ((X @ v) ** 2).sum(axis=0)               # function to maximize over v

     nrm  = colors.Normalize(vmin=np.min(f), vmax=np.max(f))
     sm = cm.ScalarMappable(norm=nrm, cmap='YlOrRd')
     for i in range(v.shape[1]):      # color vectors based on f value
         ax.arrow(0, 0, v[0, i], v[1, i], width=0.025, color=sm.to_rgba(f[i]),␣
      ↪alpha=0.6)
     plt.colorbar(sm); plotX(X, ax)
```



Here, the arrows represent unit vectors v, and they are colored according to the value of the following function of the vectors v:

$$f(v) = \sum_{i=1}^{m} (v \cdot R_i)^2.$$

From the figure, there is no doubt that the vectors v for which this function takes the largest values indicate the "dominant" direction of the data points. Once we find the first maximal vector, then we can restrict to the orthogonal complement of that vector and repeat the same maximization to compute further principal components. (In two dimensions, this becomes trivial, so we proceed ignoring further components.)

Statistical literature usually considers the maximization of the function

$$g(v) = \frac{1}{m-1} \sum_{i=1}^{m} (v \cdot R_i)^2$$

instead of the above f. Of course, the maximizers of f and g are the same. The function g represents the *variance* of the data R_i projected onto v, which is the statistical quantity that the first principal component maximizes.

How do we solve the maximization problem? The answer is given in the next theorem.

XVI.3 PCA and SVD

The key mathematical device for PCA is a tool we have studied in a prior lecture, the SVD.

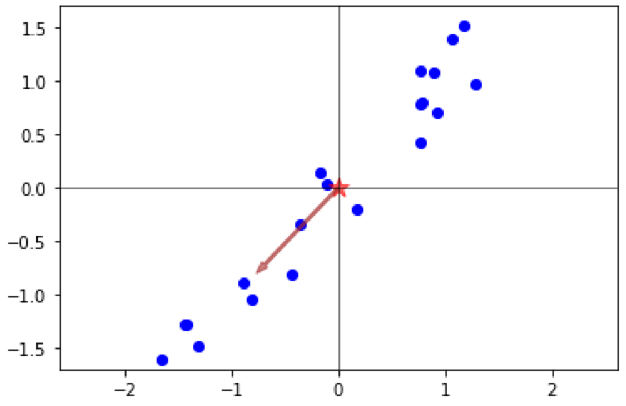
Theorem 1. Let $X = U\Sigma V^t$ be an SVD of $X \in \mathbb{R}^{m \times n}$ and let $V = [v_1, v_2, \ldots, v_n]$. If X represents centered data, then its ith principal component vector equals (up to a sign) the ith right singular vector v_i of the SVD of X.

For an example, we return to the previous two-dimensional centered dataset `X` and compute its SVD.

[7]:
```
u, s, vt = svd(X)
```

Plotting the first right singular vector as an arrow through the centered data immediately illustrates the theorem's claim. We find that the first right singular vector is in one of the two directions where we expected the maximizer of f, in view of the previous figure.

[8]:
```
fig = plt.figure(); ax = plt.gca()
ax.arrow(0, 0, vt[0, 0], vt[0, 1], width=0.025, color='brown', alpha=0.6)
plotX(X, ax)
```



The second singular vector is of course orthogonal to the one shown. (Recall that the columns of a unitary matrix are orthonormal.)

You might now be thinking that this figure is beginning to look like the linear regression figure of the previous lecture, especially if one draws a line through that arrow, and compare it with the regression line. Let me check that thinking right away.

XVI.4 PCA is different from regression

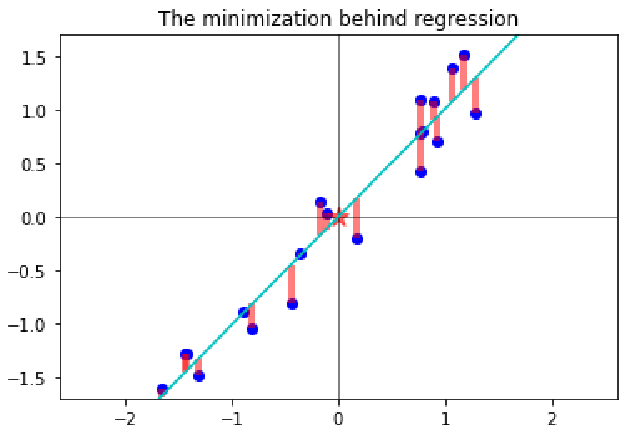
PCA and linear regression are fundamentally different. Note these differences:

- In supervised learning by regression, the data points were expressed as (x_i, f_i) to indicate that the labels f_i were *dependent* on the data x_i.
- In contrast, now the data is viewed as just points on the plane (without any labels) so we express the same points as (x_i, y_i). We do not start with an assumption that one data component depends on the other.
- In supervised learning by regression, the task was to predict values of the label f for new values of x. In PCA, the task is to discover what relationship exists, if any, between the x and y values.

So, in spite of these philosophical differences between linear regression and PCA, why is it producing similar-looking pictures in this two-dimensional example?

Actually, the pictures are not quite identical. Let us compute and plot the line obtained with linear regression applied to the same points, now viewing one of the variables (the second) as dependent on the other (the first).

[9]:
```
def plot_reg(X, ax):

    # hypothesizing that f depends on x, perform regression
    x = X[:, 0]; f = X[:, 1]

    X1 = np.array([np.ones(X.shape[0]), x]).T
    a = np.linalg.inv(X1.T @ X1) @ X1.T @ f
    x_predict = np.linspace(-2, 2, num=100)
    f_predict = a[0]  + a[1] * x_predict
    plotX(X, ax)
    ax.plot(x_predict, f_predict, 'c');
    fp = X1 @ a
    lc = LineCollection([[(x[i], f[i]), (x[i], fp[i])]
                          for i in range(len(x))],
                         color='r', linewidth=4, alpha=0.5)
    ax.add_collection(lc)
    ax.set_title('The minimization behind regression');
fig = plt.figure(); ax = plt.gca()
plot_reg(X, ax)
```



Recall that the line in this linear regression is arrived at by minimizing the sum of the squares of the lengths of the (red) vertical line segments.

In PCA, a different quantity is minimized. Although we defined the principal components using a maximization, we can transform it to a minimization as follows. Recall from linear algebra that any vector can be decomposed into its projection along a given vector and a component in the orthogonal complement. In particular, for the above two dimensional data, the vector R_i can be decomposed into its projection along v, $(v \cdot R_i)v$ plus the component of R_i in the orthogonal complement of v, which using a unit vector $v^\perp$ perpendicular to v, may be expressed as $(v^\perp \cdot R_i)v^\perp$, i.e.,

$$R_i = (v \cdot R_i)\, v + (v^\perp \cdot R_i)\, v^\perp.$$

By Pythagoras theorem,

$$\|R_i\|^2 = (v \cdot R_i)^2 + (v^\perp \cdot R_i)^2.$$

Since the left hand side is fixed by the data, maximizing $(v \cdot R_i)^2$ over all v is equivalent to minimizing $(v^\perp \cdot R_i)^2$ over the perpendicular $v^\perp$. Thus we arrive at the conclusion that the first principal component v that maximizes

$$\sum_{i=1}^{m} (v \cdot R_i)^2$$

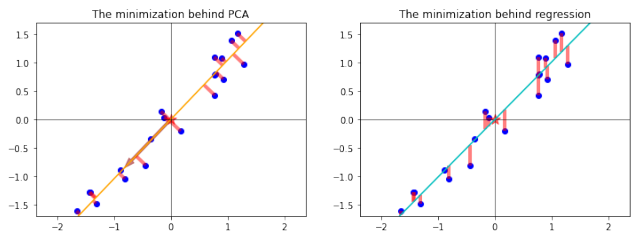
is also the same vector whose $v^\perp$ minimizes

$$\sum_{i=1}^{m} (v^\perp \cdot R_i)^2.$$

Below is the graphical illustration of this minimization behind the PCA (left plot). We draw little orange line segments from each data point R_i in the direction $v^\perp$ such that its length equals $(v^\perp \cdot R_i)^2$. Please compare it with the previous figure for linear regression, also reproduced aside below (right plot).

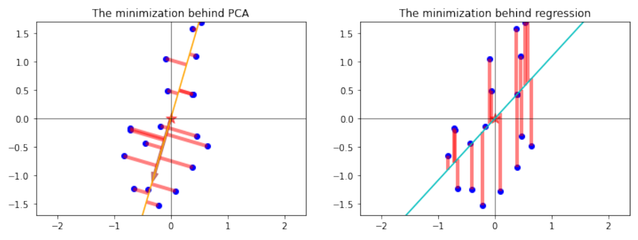
```
[10]: def plot_pca(X, ax):
          u, s, vt = svd(X)
          t = np.linspace(-3, 3, 100); v1 = vt[0, :]
          ax.plot(t*v1[0], t*v1[1], color='orange')
```

```
    ax.arrow(0, 0, v1[0], v1[1], width=0.04, color='brown', alpha=0.6)
    Xp = v1[:, np.newaxis] * (X @ v1)
    lc = LineCollection([[(X[i, 0], X[i, 1]), (Xp[0, i], Xp[1, i])]
                         for i in range(X.shape[0])],
                        color='r', linewidth=4, alpha=0.5)
    ax.add_collection(lc)
    plotX(X, ax)
    ax.set_title('The minimization behind PCA');
fig = plt.figure(figsize=(12, 4))
axl, axr = fig.subplots(1, 2)
plot_pca(X, axl); plot_reg(X, axr)
```



Clearly the two minimizations are different. The result of the different minimizations happened to be close for the above example. But this need not happen always. The results can indeed be quite different, as the quick example below shows.

[11]:
```
rng = np.random.default_rng(13)
z0 = 1.5 * rng.random(20); z1 = z0 + 2.7 * rng.random(20)
ZZ = np.array([z0, z1 ]).T
Z = ZZ - ZZ.mean(axis=0)
fig = plt.figure(figsize=(12, 4))
axl, axr = fig.subplots(1, 2)
plot_pca(Z, axl); plot_reg(Z, axr)
```



What distinguishes PCA is not simply this difference in the associated minimization problems. As you proceed with this lecture, you will understand that the power of PCA lies in its ability to find patterns in data, i.e., to find feature sets or basis sets in which data can be efficiently represented.

XVI.5 PCA in scikit-learn

Instead of getting the principal components from first principles using the SVD, as we have done above, you may just use scikit-learn's PCA facility to get the same result.

```
[12]: from sklearn.decomposition import PCA
```

To use it, one constructs a `PCA` object using some hypothesized `n_components` which can be less than the data dimensions m and n. To draw the analogies with the previous computation, let's apply PCA to the previous data setting `n_components=2` (noting that $\min(m,n) = 2$ in this example).

```
[13]: pca = PCA(n_components=2)
```

You can directly give PCA a data set that is not centered. It will do the centering behind the scenes.

```
[14]: pca.fit(XX);    # fit with raw (uncentered) data
```

Now, you may ask for the principal components of the data:

```
[15]: pca.components_
```

```
[15]: array([[-0.69180966, -0.72207991],
             [ 0.72207991, -0.69180966]])
```

This matches the principal components we computed using the SVD, reproduced below.

```
[16]: vt
```

```
[16]: array([[-0.69180966, -0.72207991],
             [ 0.72207991, -0.69180966]])
```

(Note that since principal components are defined only up to a sign, the vectors need only match up to a sign, in general.)

XVI.6 Mapping PCA and SVD jargon

To expand on the above seen relationships, let's consider a larger data set (one that we will examine in more detail in the next section), to bring out the correspondences between what `PCA` provides and what `svd` provides. This will help us understand the concepts from different viewpoints.

```
[17]: from sklearn.datasets import load_digits
      digits = load_digits()
      XX = digits.data
      X = XX - XX.mean(axis=0)
      m, n = XX.shape; m, n
```

```
[17]: (1797, 64)
```

The following two lines computes PCA (using scikit-learn) and SVD (using scipy). We will use the resulting outputs to establish correspondences between them so we can be fluent in both languages.

```
[18]: pca = PCA(svd_solver='full').fit(XX)
```

```
[19]: u, s, vt = svd(X)
```

Correspondences Now we make a series of observations regarding the outputs from scipy's `svd` applied to centered data and outputs from scikit-learn's `PCA`. (Note that if you send the data matrix to SVD without centering, these correspondences do not apply.)

First, the most obvious correspondence is that `pca.singular_values_` and the singular values from scipy's `svd` are the same:

```
[20]: norm(pca.singular_values_ - s)
```

```
[20]: 0.0
```

Second, the principal components returned by `pca` are equal to $\pm$ (ith right singular vector) from the SVD. Let me illustrate this using the above `pca` and `svd` outputs. To check that two vectors are equal except for the sign "$\pm$," we define a function that computes the norms of the sum and the difference of the vectors and prints them out. Only *one of them* need be zero to have a match up to $\pm$.

```
[21]: def vectors_plus_minus_diff(v1, v2):
          print('%2.1f %2.1f' %(norm(v1 - v2), norm(v1 + v2)))
```

Using this function we check if the first seven principal components equal the corresponding singular vector up to $\pm$. Note how one of the printed out norms (either that of the sum or that of the difference) is zero.

```
[22]: for i in range(7):
          vectors_plus_minus_diff(pca.components_[i, :], vt[i, :])
```

```
2.0 0.0
2.0 0.0
2.0 0.0
2.0 0.0
2.0 0.0
0.0 2.0
0.0 2.0
```

Third, projections of the original data onto the principal axes can be obtained by `transform` (or the `fit_transform`) method of PCA. Of course, such projections are just V-components of the rows of the data X, or simply XV: since $X = U\Sigma V^t$ and V has orthonormal columns, these projections are also equal to $U\Sigma$. Hence we have the following correspondence:

$$i^{\text{th}}\text{column of } \texttt{transform(XX)} = \pm\, i^{\text{th}}\text{column of } U\Sigma.$$

Here is an illustration of this correspondence for the current example.

[23]:
```
# projected data from pca (can also use pca.fit_transform(XX)):
projX = pca.transform(XX)
```

[24]:
```
# projected data from svd:
us = u[:, :len(s)] @ np.diag(s)
```

[25]:
```
# check they are same upto a sign
for i in range(7):
    vectors_plus_minus_diff(projX[:, i], us[:, i])
```

```
1134.0 0.0
1084.5 0.0
1009.3 0.0
852.2 0.0
706.7 0.0
0.0 651.6
0.0 610.5
```

Fourth, to relate to the low-rank approximation using SVD that we studied in the SVD lecture, recall that an SVD of X can be rewritten using outer products as

$$X = \sum_{j=1}^{\min(m,n)} \sigma_j u_j v_j^*$$

from which the best rank ℓ approximation to X, denoted by X_ℓ, can be extracted simply by throwing away the later summands:

$$X_\ell = \sum_{j=1}^{\ell} \sigma_j u_j v_j^*.$$

Before showing how this is done in scikit-learn, let us compute X_ℓ, say for $\ell = 5$, using the SVD. We implement the above formula, and add the means to compensate for the fact that the SVD was taken on centered data.

[26]:
```
l = 5
Xl_svd = u[:, :l] @ np.diag(s[:l]) @ vt[:l, :] + XX.mean(axis=0)
```

There is a corresponding facility in scikit-learn. First, note that we may give the `n_components` argument to PCA, which tells PCA how many principal components to compute.

[27]:
```
# The rank l approximation needs only l principal components
pcal = PCA(n_components=l, svd_solver='full').fit(XX)
```

Now, to get the best rank ℓ approximation from PCA, we use the `transform` method, which gives the components of the data projected onto the principal axes (and there are 5 principal axes now). Then, we can use the `inverse_transform` method to lift the projected components into the original data space of 64 pixels.

[28]:
```
projX = pcal.transform(XX)
projX.shape   # the shape reflects projected data sizes
```

```
[28]: (1797, 5)
```

```
[29]: Xl_pca = pcal.inverse_transform(projX)
      Xl_pca.shape  # the shape is now the shape of original data
```

```
[29]: (1797, 64)
```

The relative difference in norm between `Xl_pca` and `Xl_svd` can now be easily verified to be close to machine precision.

```
[30]: norm(Xl_pca - Xl_svd) / norm(Xl_pca)
```

```
[30]: 1.0648619605229708e-15
```

Let's summarize this correspondence as follows: The best rank ℓ approximation X_ℓ of the centered data X satisfies

$$X_\ell = \texttt{inverse\_transform}(\texttt{transform}(\texttt{XX}) - \texttt{mean\_}(\texttt{XX})$$

Just in case this `inverse_transform` lifting into data space still sounds mysterious, then perhaps this reverse engineered formula for it might make it clearer:

$$\texttt{inverse\_transform}(\texttt{proj}) = \texttt{proj@pca.components\_} + \texttt{mean\_}(\texttt{XX}),$$

This also can again immediately be verified in our example:

```
[31]: Xl_pca2 = projX @ pcal.components_ + pcal.mean_
      norm(Xl_pca2 - Xl_pca)
```

```
[31]: 0.0
```

Fifth, consider the attribute called the `explained_variance` array of the `pca` object. This represents variances explained by the principal components (see the covariance matrix discussion below for more on this terminology). The elements of this array are related to the singular values σ_i as follows.

$$\texttt{pca.explained\_variance\_}[\texttt{i}] = \frac{1}{m-1}\sigma_i^2$$

```
[32]: norm(pca.explained_variance_ - (s**2/(m-1)))
```

```
[32]: 0.0
```

Sixth, consider another attribute of the `pca` object called `explained_variance_ratio_`. It is related to singular values as follows:

$$\texttt{pca.explained\_variance\_ratio\_}[\texttt{i}] = \frac{\sigma_i^2}{\sum_j \sigma_j^2}$$

As is obvious from this definition, the sum of all the explained variance ratios should be one. Here is the verification of the formula stated above for the current example:

[33]:
```
norm(pca.explained_variance_ratio_ - (s**2)/(s**2).sum())
```

[33]:
```
6.661408213830422e-17
```

Covariance matrix To understand the origin of some of the terms used in `pca` attributes, recall how the *covariance matrix* is defined: For centered data, the covariance matrix is

$$C = \frac{1}{m-1} X^t X.$$

The "explained variances" are the eigenvalues of C. Of course, since $X = U\Sigma V^t$ is an SVD of X, the covariance matrix C may be alternately expressed as

$$C = V \frac{\Sigma^2}{m-1} V^t,$$

from which we conclude that the ith eigenvalue of C is $\sigma_i^2/(m-1)$, which matches our observation above.

This observation also tells us that the right singular vectors (the columns of V) are actually eigenvectors of C, since the above factorization of C is actually a diagonalization of C. Therefore, one can alternately compute the right singular vectors, aka, principal components, as the *eigenvectors of the covariance matrix* simply using numpy's or scipy's `eig`. Indeed, for the current example, we can immediately cross check that we get the same results:

[34]:
```
ew, ev = np.linalg.eig(X.T @ X / (m-1)) # eigenvalues & eigenvectors of C
ii = ew.argsort()[::-1]
ew = ew[ii]; ev = ev[:, ii]             # sort by descending order of␣
 ↪eigenvalues
```

[35]:
```
norm(ew - s**2 / (m-1)) # eigenvalues equal singular values squared /␣
 ↪(m-1)
```

[35]:
```
6.25037049393972e-13
```

[36]:
```
for i in range(7):      # eigenvectors equal +/- principal components
    vectors_plus_minus_diff(pca.components_[i, :], ev[:, i])
```

```
2.0 0.0
2.0 0.0
2.0 0.0
0.0 2.0
2.0 0.0
0.0 2.0
0.0 2.0
```

What is better, `eig` or `svd`? The relationship between PCA/SVD and eigenvectors of the covariance matrix discussed above raises a natural question. If both give the same vectors (principal components), which one should be recommended for computations?

Even though both give the same vectors mathematically, it's better to use SVD (or scikit-learn's PCA, which uses SVD) to avoid round-off errors in the formation of $X^t X$ that arise in some circumstances. A classical example is the case of a Läuchli matrix, an $N \times (N-1)$ rectangular matrix of the form

$$X = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \epsilon & & & 0 \\ 0 & \epsilon & & \vdots \\ \vdots & & \ddots & \\ \vdots & & & \epsilon \\ 0 & & & 0 \end{bmatrix}$$

with a small number ϵ. The matrix $X^t X$ then has $1+\epsilon^2$ on the diagonal, and ones everywhere else. This matrix is very close to being singular numerically. For example, in the $N = 4$ case, the matrix

$$X^t X = \begin{bmatrix} 1+\epsilon^2 & 1 & 1 \\ 1 & 1+\epsilon^2 & 1 \\ 1 & 1 & 1+\epsilon^2 \end{bmatrix}$$

has eigenvalues $3+\epsilon^2, \epsilon^2, \epsilon^2$ by hand calculation. However, `eig` is unable to distinguish the latter from zero.

[37]:
```
N = 4
eps = 1e-8
X = np.diag(eps * np.ones(N), k=-1)
X[0, :] = 1;  X = X[:, :(N-1)]; X
```

[37]:
```
array([[1.e+00, 1.e+00, 1.e+00],
       [1.e-08, 0.e+00, 0.e+00],
       [0.e+00, 1.e-08, 0.e+00],
       [0.e+00, 0.e+00, 1.e-08],
       [0.e+00, 0.e+00, 0.e+00]])
```

[38]:
```
ew, ev = np.linalg.eig(X.T @ X)
ii = ew.argsort()[::-1]; ev = ev[:, ii]; ew = ew[ii]
ew
```

[38]:
```
array([ 3.00000000e+00,  0.00000000e+00, -2.22044605e-16])
```

The last two numbers are so close to machine precision that they are indistinguishable from 0. Covariance matrices should never have negative eigenvalues, but due to numerical difficulties, `eig` may return a small negative value as an eigenvalue. So in particular, if we attempt to compute the singular values by taking square roots of these eigenvalues, we might end up taking the square root of a negative number.

[39]:
```
np.sqrt(ew)
```

```
<ipython-input-39-6f1372a76c3a>:1: RuntimeWarning: invalid value encountered in sqrt
  np.sqrt(ew)
```

```
[39]: array([1.73205081, 0.        ,        nan])
```

In contrast, the SVD is able to output the singular values fairly close to the exact ones $\sqrt{3+\epsilon^2}, \epsilon, \epsilon$ without difficulty.

```
[40]: u, s, vt = svd(X)
      s
```

```
[40]: array([1.73205081e+00, 1.00000000e-08, 1.00000000e-08])
```

XVI.7 Hand-written digits dataset

Scikit-learn comes with an example dataset representing many images of hand-written digits for use as a test problem in optical character recognition. Actually, this is the same `digits` data we have been working with above. Let's take a closer look at this dataset.

```
[41]: from sklearn.datasets import load_digits
      digits = load_digits()
      digits.keys()
```

```
[41]: dict_keys(['data', 'target', 'target_names', 'images', 'DESCR'])
```

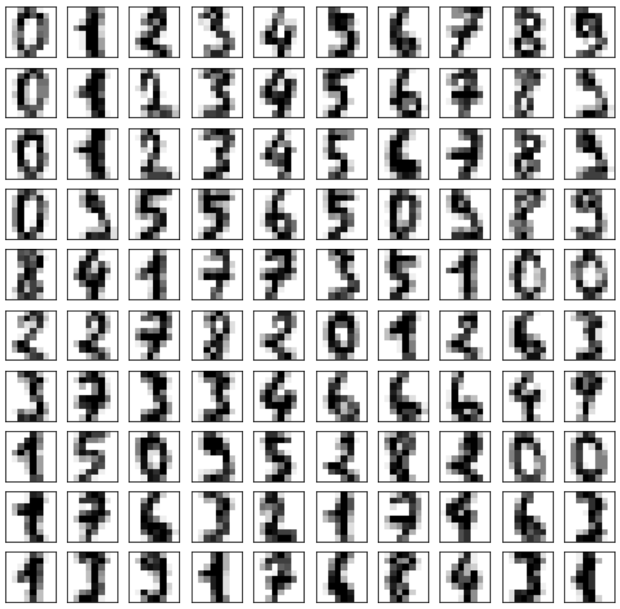
We used `digits.data` previously. The `images` key gives the images of the handwritten digits.

```
[42]: digits.images.shape, digits.data.shape
```

```
[42]: ((1797, 8, 8), (1797, 64))
```

There are 1797 images, each of 8 x 8 pixels. The flattened array versions of these images are in `digits.data` while the 8×8 image versions are in `digits.images`. Here are the first few of the 1797 images:

```
[43]: fig, axes = plt.subplots(10, 10, figsize=(8, 8), subplot_kw={'xticks':[],␣
       ↪'yticks':[]})
      for i, ax in enumerate(axes.flat):
          ax.imshow(digits.images[i], cmap='binary')
```



To apply PCA, we need to put these images into the tidy data format of m samples/observations/rows $\times$ n variables/features/columns. We set

- each *pixel* to be a *feature*/variable,
- each *image* to be a *sample*/observation.

Actually, this is the form the data is contained in `digits.data`, where each 8×8 image is one of 1797 samples of a 64-variable dataset.

```
[44]: m, n = digits.data.shape
      m, n
```

```
[44]: (1797, 64)
```

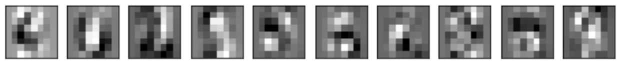
We construct a `PCA` object using this data asking specifically to retain only 10 principal components.

```
[45]: pca = PCA(n_components=10).fit(digits.data)
```

Would you hazard a guess that the 10 principal components are the usual 10 digits?

Well . . . here is how the 10 principal components look like:

```
[46]: fig, axes = plt.subplots(1, 10, figsize=(8, 4),
                               subplot_kw={'xticks':[], 'yticks':[]})
      for i, ax in enumerate(axes.flat):
          ax.imshow(pca.components_[i, :].reshape(8, 8), cmap='binary')
```



Obviously, these outputs don't look anything like recognizable digits. It is important to understand in what sense these garbled images represent something "principal" about the original data set. Proceed on to gain this understanding.

XVI.8 PCA is a feature finder

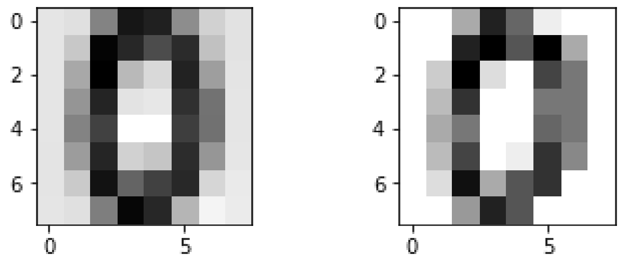
To make sense of the above garbled images as a basis, let's use the `transform` method (which, as you recall from the correspondences above, computes $U\Sigma$).

[47]:
```
projdgt = pca.transform(digits.data)
projdgt.shape
```

[47]:
```
(1797, 10)
```

Each row of `projdgt` contains 10 coefficients, which when multiplied by the 10 principal components, reveal what's going on. Of course, we must also correct for the previously subtracted mean. The first row of `dgt` then yields the following image (left). Compare it with the original first image in the data (right). Of course, this is reminiscent of the low-rank approximation of a single image that we discussed in the prior SVD lecture; the difference now is that we are applying the same process to 1797 centered images all at once (although we are only showing the first one below).

[48]:
```
fig = plt.figure(figsize=(6, 2))
axl, axr = fig.subplots(1, 2)
reconstructed_dgts = pca.inverse_transform(projdgt)
im0 = reconstructed_dgts[0, :]
# alternately and equivalently, we may set im0 by
# im0 = dgt[0, :] @ pca.components_ + pca.mean_
axl.imshow(im0.reshape(8, 8), cmap='binary')
axr.imshow(digits.images[0], cmap='binary');
```



Let's dig a bit more into this. Writing the SVD of the centered image data array `X` as

$$X = \sum_k \sigma_k u_k v_k^t,$$

we may read off the the ith row R_i, which represents the ith image in this dataset, as follows:

$$[R_i]_j = X_{ij} = \sum_k \sigma_k [u_k]_i [v_k]_j.$$

The `pca` object above computed the rank-10 best approximation by restricting the above sum to the first 10 summands. This is what is was implemented above by the line of code

```
reconstructed_dgts = pca.inverse_transform(projdgt)
```

From the previously discussed fourth correspondence's equivalent form of the `inverse_transform`, we note that the above statement may equivalently be written as

```
reconstructed_dgts = projdgt @ pca.components_ + pca.mean_
```

where the correction for the zero mean is explicit. This also makes it abundantly clear that the statement setting `reconstructed_dgts` is just an implementation of the above formula for R_i.

Viewing the ith image/row R_i as a function f of pixels, it is instructive to view the above formula for R_i as the sum

$$f(x) = a_0\phi_0(x) + a_1\phi_1(x) + \cdots a_9\phi_9(x)$$

where

$$a_k = [\sigma_k u_k]_i, \qquad \phi_k = v_k,$$

i.e., the numbers $a_k = [\sigma_k u_k]_i$ represent coefficients in a basis expansion with the **basis images** ϕ_k set by $\phi_k = v_k$, and where x represents one of the 64 pixels. In this viewpoint, what PCA has done is to fit the 10-term formula for f to a data set of 1797 images. While this is reminiscent of regression, note two important differences:

- PCA found the basis ϕ_k (while regression needs ϕ_k as input).
- The coefficients a_k change for each data row (unlike in regression where it's fixed for the whole dataset).

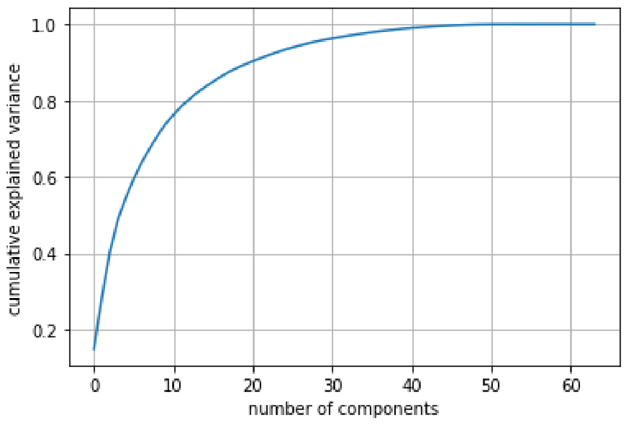
To summarize, **PCA automatically finds an efficient basis (or *feature* set) to represent the data**. (In contrast, regression needs you to provide a basis ϕ_k as input in order to output the best-fit coefficients a_k; see e.g., the curve fitting examples we have seen previously.) This exemplifies one of the differences between supervised and unsupervised learning.

XVI.9 PCA is useful for dimensionality reduction

The left (PCA) and the right (original data) images in the previous figure strongly suggests the following interpretation: the original 64-dimensional dataset might actually be well represented in a 10-dimensional space!

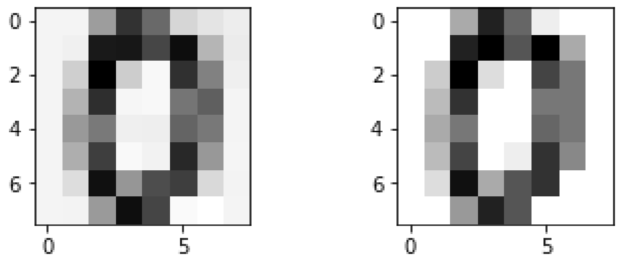
The number 10 was, of course, arbitrary, and somewhat of a red herring in a dataset of images of 10 digits. It would be better if the data itself can lead us to some number of relevant dimensions it possesses. This is where the `explained_variance_ratio` becomes useful. Let's return to the full PCA and examine this array. Recall that it is an array that sums to one, so its *cumulative sums* indicate how close we are to fully representing the data.

[49]:
```
pca = PCA().fit(digits.data)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.grid(True)
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```



Clearly, with 10 components, we are far away from the cumulative sum of 1. We are much closer to the point of diminishing returns, retaining about 95% of the variance, if we instead choose, say 30 components.

[50]:
```
pca = PCA(n_components=30).fit(digits.data)
dgt = pca.fit_transform(digits.data)
fig = plt.figure(figsize=(6, 2))
axl, axr = fig.subplots(1, 2)
im0 = dgt[0, :] @ pca.components_ + pca.mean_
axl.imshow(im0.reshape(8, 8), cmap='binary')
axr.imshow(digits.images[0], cmap='binary');
```



In other words, the 64-dimensional data set may effectively be reduced to a 30-dimensional dataset retaining 95% of the variance. (Per our discussion in the prior SVD lecture, you can, of course, also convert this statement on variances into a precise measure of the relative error in the Frobenius norm.) Summarizing, PCA is also useful as a dimensionality reduction tool.

XVII
Latent Semantic Analysis

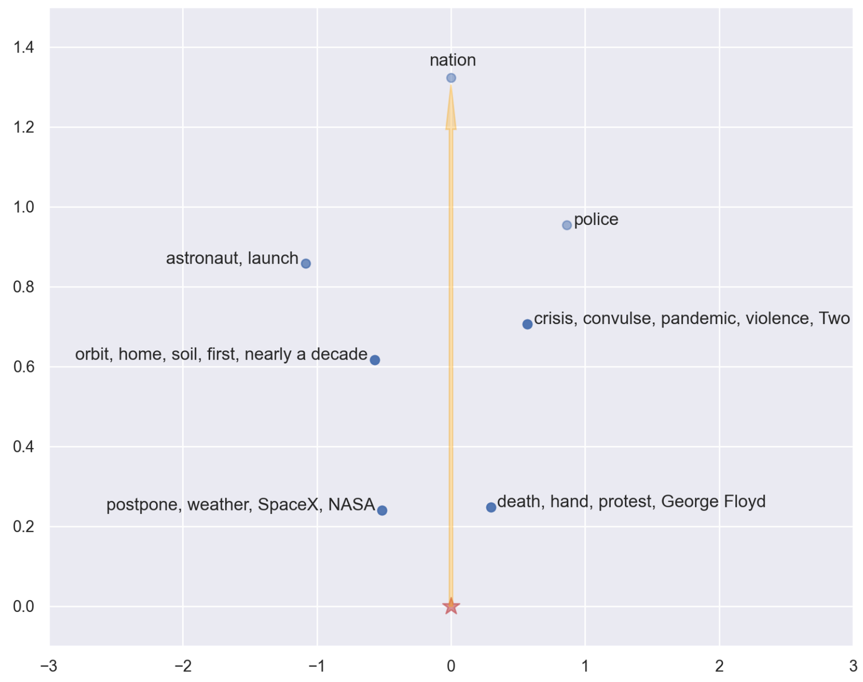
June 1, 2020

In the study of information retrieval systems, a fundamental question is how to extract documents from a large collection in response to a user query. A simplistic way is to pick out all documents which contain the query words. Is there a more "intelligent" way? Documents usually have interrelated *concepts* and if a query could be matched to a concept, perhaps the results extracted would look more intelligent. Documents are written in natural language, using copious amounts of *words*, yet the number of topics that people write about are usually much smaller than the number of words they use. Latent Semantic Analysis (LSA) is a technique to associate *concepts* in a space of much lower dimension than a space of *words* in order to help with the complex task of information retrieval.

Of course, a number of details have to be worked out. How can one associate words to a vector space? How can one identify topics in this space? How can one represent queries? It should therefore not be surprising that this is a whole field of study in itself: see e.g., [MRS]. Yet, we are able to take a peek into this machinery because the essential mathematical tool used in LSA is something you already know, namely the SVD.

I'm sure yesterday's news is very much on your mind, with the best and the worst of humanity on display. Shocking police violence and a successful astronaut launch dominated the news headlines. Having failed to get the news out of my mind, I am going to use sentences from current news for introducing LSA.

The next graph, obtained from LSA's interpretation of *four news headlines* on a two-dimensional space made in this lecture, may well be a representation of the country's current state. Today's lecture will show you how to analyze text and graphically display words and their apparent connections like those displayed below.



If this is a proxy for the country's current state, where we go from here seems critical in this moment.

XVII.1 Natural language processing

Using a few headlines, we make a *corpus* of text documents to illustrate the basics of LSA as a python dictionary, called `c` below.

```
[1]: c = {'May31':
         'Two crises convulse a nation: a pandemic and police violence',

         'May30a':
         'Nation's first astronaut launch to orbit from home soil in nearly a␣
      ↪decade',

         'May30b':
         'Death of George Floyd at the hands of police set off protests',

         'May27':
         'SpaceX launch of NASA astronauts is postponed over weather'}
```

In this corpus, `c['May31']` is a *document*, and the `c` has three more documents. Each document is composed of many words, or *terms*. We need to simplify the complexities of natural language to be able to compute anything. With my apologies to the writers among you, we proceed by taking the view that the order of words, declensions and conjugations, and often-used words like articles and prepositions, are all immaterial. Then we view *concepts* as merely associations of the remaining root words, associations marked by their joint appearances in documents. LSA is only useful under the assumption that words that are close in semantics will occur in similar documents as the corpus of documents become large.

Applying the above-mentioned language simplifications to even a small corpus is a lot of work, if you try to do it from scratch. But thankfully, there are several python modules that excel in natural language processing (NLP). Below, I will use spaCy, one of the recent additions to the python NLP tool set. (Please install it and also make sure to install their English dataset `en_core_web_sm`, say by `python3 -m spacy download en_core_web_sm`, before proceeding.)

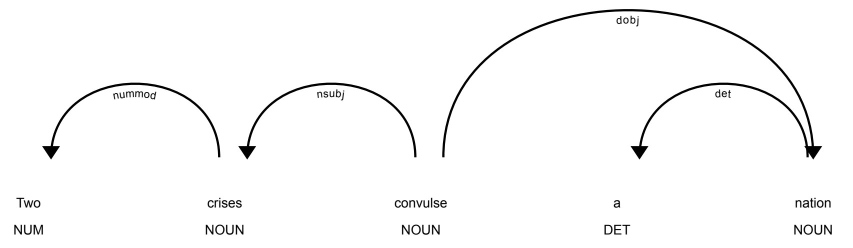
[2]:
```
import spacy
from spacy import displacy

# Install dataset: python3 -m spacy download en_core_web_sm
nlp = spacy.load('en_core_web_sm')
```

Consider the first sentence in our corpus.

[3]:
```
doc0 = nlp('Two crises convulse a nation')
```

The spacy module is able to process sentences, and identify nouns, verbs, direct objects, and their interrelationships. In the cell below, after processing a sentence, the result is saved in an SVG figure. The saved image is then displayed as an image in the next cell.

[4]:
```
svg = displacy.render(doc0, style="dep", jupyter=False)
with open('../figs/sentence0.svg', 'w') as f: f.write(svg)
```

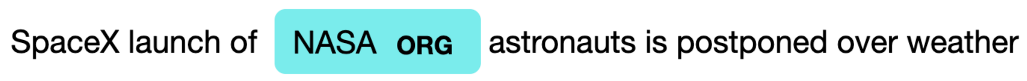


Within a Jupyter notebook, one may also directly render the resulting image (without needing to save the image into a file) by specifying `jupyter=True` instead. Here is an example.

[5]:
```
doc1 = nlp('SpaceX launch of NASA astronauts is postponed over weather')
displacy.render(doc1, style='ent', jupyter=True, options={'distance':90})
```

```
<IPython.core.display.HTML object>
```

Just in case you are not reading this in a Jupyter notebook and the image does not render on your reading device, I am reproducing the image that displacy generated:



As you can see from the annotated sentence, the module can even identify some named entities in the real world: it knows about NASA, but it still does not know about SpaceX! (We will fix this later in this lecture by adding our own named entity terms.)

We shall use the package's capabilities for **tokenization** and **lemmatization**. Tokenization is the process of dividing a sentence or a document into words via demarcation characters like white spaces. Lemmatization is the process of identifying the so-called "lemma" of a word, allowing us to group together inflected forms of the word into a single item. Here is the result of tokenization and lemmatization on the above sentence. Note how the originally found words "astronauts" and "postponed" have changed in the output.

```
[6]: [w.lemma_ for w in doc1 if not w.is_stop]
```

```
[6]: ['spacex', 'launch', 'NASA', 'astronaut', 'postpone', 'weather']
```

Here we have also removed **stop words**, a collection of the most common words in a language as previously identified and categorized by the NLP program. In the above example, the words "of", "is", and "over" have been removed. You can view spacy's collection of all stop words if you use the following import statement.

```
from spacy.lang.en.stop_words import STOP_WORDS
```

XVII.2 Term-document matrix

The important mathematical object for LSA is the **term-document** matrix, a matrix whose rows correspond to terms, whose columns correspond to documents, and whose element at position (t, d) is 1 if the document in column d contains the term in row t, and is 0 otherwise. (You will find variations on this matrix in the literature, e.g., the tranpose, ir refinements beyond 0/1 entries, are often used.) Let's make this matrix with a quick hack (where we have now also asked spacy to ignore punctuations). The matrix will be displayed as a pandas data frame to easily visualize term and document labels of rows and columns.

```
[7]: import pandas as pd
     from scipy.sparse import lil_matrix

     d = {}
     for j, dok in enumerate(c.keys()):
         tokens = [w.lemma_ for w in nlp(c[dok])
                   if not w.is_stop and w.pos_ != 'PUNCT']
         for t in tokens:
```

```
        d[t] = d.setdefault(t, [])
        d[t] += [j]
A = lil_matrix((len(d.keys()), len(c.keys())), dtype=int)
for i, t in enumerate(d.keys()):
    for j in d[t]:
        A[i, j] = 1
Adf = pd.DataFrame(A.toarray(), index=d.keys(), columns=c.keys()); Adf
```

[7]:

```
           May31  May30a  May30b  May27
crisis         1       0       0      0
convulse       1       0       0      0
nation         1       1       0      0
pandemic       1       0       0      0
police         1       0       1      0
violence       1       0       0      0
astronaut      0       1       0      1
launch         0       1       0      1
orbit          0       1       0      0
home           0       1       0      0
soil           0       1       0      0
nearly         0       1       0      0
decade         0       1       0      0
death          0       0       1      0
George         0       0       1      0
Floyd          0       0       1      0
hand           0       0       1      0
set            0       0       1      0
protest        0       0       1      0
spacex         0       0       0      1
NASA           0       0       0      1
postpone       0       0       0      1
weather        0       0       0      1
```

We might want to have a combination of first and last names treated as a single entity, but the code is not yet smart enough to do that. We'll fix that later, after introducing the idea of LSA. For the moment, note how words have been represented as row vectors and documents as column vectors. This is enough to understand the basics of LSA, as we see next.

XVII.3 The idea of LSA

The idea is to perform an SVD of the term-document matrix and use its low-rank approximation, with a rank k much less than the number of words. The dominant singular vectors may then be expected to capture patterns in the association of words. Of course, this is not an exact technique, but it does give us something numerical to work with for analysis of large amounts of textual data. For our example of the 4-document corpus, we shall use the best rank-2 approximation (as discussed in the SVD lecture), the difference now being that we don't actually need the low-rank matrix, but rather the SVD components that go into it.

```
[8]: import numpy as np
     import matplotlib.pyplot as plt
     import seaborn; seaborn.set();
     from numpy.linalg import norm
     from scipy.linalg import svd
```

```
[9]: u, s, vt = svd(A.toarray())
```

Here is the first important step in creating mathematical objects to represent documents. Using the best rank k approximation, the first k right singular vectors are used to *represent each document as a k vector.*

```
[10]: k = 2                                # Limit to rank k
      Vt = vt[:k, :]
      pd.DataFrame(Vt, columns=c.keys())   # Documents as k-vectors
```

```
[10]:      May31    May30a    May30b     May27
      0 -0.269907 -0.829243 -0.109002 -0.477101
      1  0.458490 -0.149538  0.854138 -0.194611
```

The second important step is to represent words (or terms) as mathematical objects in the same space. Unlike documents, the words/terms are represented by the first k left singular vectors, weighted by the associated singular values. The first five word tokens are displayed below as vectors.

```
[11]: US = u[:, :k] @ np.diag(s[:k])
      usp = pd.DataFrame(US, index=d.keys()) # Words as k-vectors
      usp.head()
```

```
[11]:                  0         1
      crisis   -0.269907  0.458490
      convulse -0.269907  0.458490
      nation   -1.099150  0.308952
      pandemic -0.269907  0.458490
      police   -0.378909  1.312628
```

Many words are mapped to the same point in such a small example. In other words, there is not enough data in our small corpus to distinguish between such words.

Nonetheless, even in our very small dataset, it is very interesting to see the associations between words in terms of how different the word vectors are. Ignoring the magnitude of word vectors, one may measure the difference between two word vectors (both drawn from the origin) using a device different from the norm. When magnitude is ignored, the difference between vectors is captured by the *angle* the word vectors make with each other, or by the **cosine** of the angle. Two vectors of the same magnitude are farther apart if the cosine of their angle is smaller. Remember that it's very easy to compute the cosine of the angle between two unit vectors, since it is equal to their dot product.

```
[12]: astronaut = usp.loc['astronaut', :].to_numpy()
      crisis    = usp.loc['crisis', :].to_numpy()
      police    = usp.loc['police', :].to_numpy()
```

Here is an example of an uncanny association the program has made:

The word `crisis` is closer to `police` than to `astronaut`! This conclusion follows from the two cosine computations below.

```
[13]: crisis.dot(police) / norm(police) / norm(crisis)
```

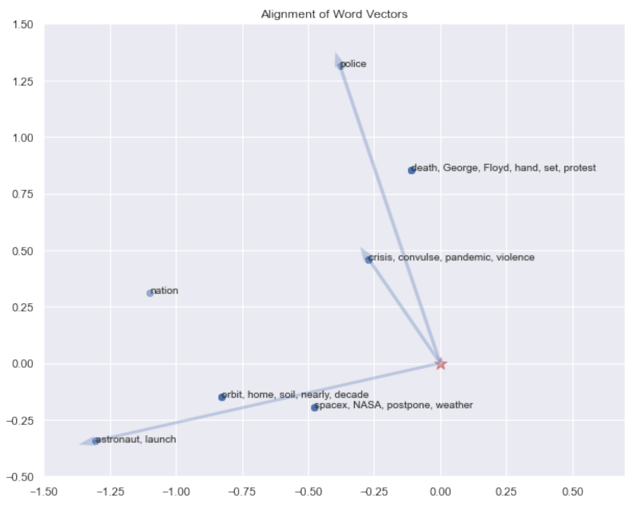
```
[13]: 0.9686558216875333
```

```
[14]: crisis.dot(astronaut) / norm(astronaut) / norm(crisis)
```

```
[14]: 0.27103529721595343
```

Let's dig into this a bit more. In our small example, since words are two-dimensional vectors, we can plot them to see how they are dispersed in terms of angles measured from the origin. Below, the origin is marked as a red star, and points representing the terminal point of word vectors are annotated with the word.

```
[15]: w = {}; us = np.round(US, 8) # w[(x,y)] = list of words at that point
      usr = list(set([tuple(us[i, :]) for i in range(us.shape[0])]))
      for i in range(len(usr)):
          w[usr[i]] = []
          for j in range(usp.shape[0]):
              if norm(usp.iloc[j, :] - usr[i]) < 1e-6:
                  w[usr[i]] += [usp.index[j]]
      fig = plt.figure(figsize=(10, 8)); ax = fig.gca()
      ax.arrow(0, 0, crisis[0], crisis[1],       width=0.015, alpha=0.3)
      ax.arrow(0, 0, police[0], police[1],       width=0.015, alpha=0.3)
      ax.arrow(0, 0, astronaut[0], astronaut[1], width=0.015, alpha=0.3)
      ax.scatter(US[: , 0], US[: ,1], alpha=0.5)
      ax.scatter(0, 0,  color='r', marker='*', s=150, alpha=0.6);
      for i, key in enumerate(w.keys()):
          ax.annotate(', '.join(w[key]), (key[0], key[1]))
      ax.set_xlim((-1.5, 0.7)); ax.set_ylim((-0.5, 1.5));
      ax.set_title('Alignment of Word Vectors');
```



The first takeaway from this figure is that the angles the word vectors make is clearly in accordance with the previous cosine computation.

The second is more enigmatic. In our small corpus of four sentences, there were two categories of news, one of violence, and one of exploration. While we as humans can instinctively make that categorization, it is uncanny that some mathematics and a few simple lines of code can separate the words associated to the two categories into different areas of a “word space”. The word that appears somewhat in the middle of the two categories is `nation`, as it ought to. (The same figure, after a rotation, modification of arrows, and cleaned up positioning, is what I presented at the beginning of the lecture.) You should now have an idea of why LSA can be useful when applied to a large corpus with many more words, documents, and hidden associations (or latent semantics).

XVII.4 Language is complex

Let me return to the news headlines. During this entire spring term, bad news have been accumulating, of how the pandemic and its repercussions are battering our country, highlighting and amplifying many of our systemic problems, and finally even more bad news of yet another police violence. All this made the few glorious moments last weekend especially precious. When SpaceX lifted NASA astronauts Bob Behnken and Doug Hurley into orbit on a reusable rocket that returned to an autonomous droneship, it was a moment of reassurance that our science, industry, and innovation remain peerless. Let me now focus on this bit of positive news and add more sentences on these exciting developments to our text corpus.

[16]:
```
c.update(
{
 'May30Launch':
 'Go NASA! Go SpaceX! Godspeed, Bob and Doug!',

 'NYTimes':
 'NASA and SpaceX officials more often than not ' +
 'just call the pilots of this historic mission Bob and Doug.',
 'May30NASAblog':
 'The first stage of the SpaceX rocket has landed ' +
 'successfully on the droneship, Of Course I Still Love You.',

 'May31NYTimes':
 'After a 19 hour trip, NASA astronauts Bob and Doug ' +
 'successfully docked their capsule and entered the space station.',
})
```

Do you see the complexities of dealing with real examples of natural language?

The ocean droneship, controlled by an autonomous robot to help the rocket land, has a curious name: "Of Course I Still Love You". Standard tokenization would simply split it into component words. It would be better to keep it as a single entity. We will do so below with spacy's facilities. But, before that, just in case you don't know, that curious name for the ship is taken from the novel *The Player of Games* by Iain M. Banks. Elon Musk gave his droneship that name in tribute to Banks. Let me add a sentence from Musk and another from Bank's novel to our text corpus.

[17]:
```
c.update(
{
 '2015Musk':
 'West Coast droneship under construction will ' +
 'be named Of Course I Still Love You',

 'IainBanks':
 'These friends of yours are ships. ' +
 'Yes, both of them. ' +
 'What are they called? ' +
 'Of Course I Still Love You and Just Read The Instructions. ' +
 'They are not warships? ' +
 'With names like that?'
})
```

To deal with text items like the droneship name, we need to use the **phrase matching** capabilities of spacy. Three examples of terms to match are added to a `TerminologyList` below. Spacy also does some default phrase matching, e.g., it identifies the phrase "nearly a decade" as a temporal unit. It is up to us whether we want to use the entire phrase as a token or not. Below, we will modify the tokenization step to keep all phrases as tokens with _ in place of white space so we can recognize them easily.

[18]:
```
from spacy.matcher import PhraseMatcher

terms = ['SpaceX',
         'Of Course I Still Love You',
         'Just Read The Instructions']
```

```
patterns = [nlp.make_doc(text) for text in terms]

matcher = PhraseMatcher(nlp.vocab)
matcher.add('TerminologyList', None, *patterns)
```

Next, we use a slicing feature (called `Span`) of spacy to capture the matched phrases as tokens. We also use the `ents` attribute provided by spacy to add *named entities* (a real-world object with a name) to the document object.

[19]:
```
from spacy.tokens import Span

def tokensfromdoc(doc):
    d = nlp(doc)
    matches = matcher(d)
    for match_id, start, end in matches:
        term = Span(d, start, end, label='myterms')
        d.ents = list(d.ents) + [term]
    tokens = [w.lemma_ for w in d
              # no pronouns
              if w.pos_ != 'PRON'   \
              # no punctuations
              and w.pos_ != 'PUNCT' \
              # not Beginning of a named entity
              and w.ent_iob_ != 'B' \
              # not Inside a named entity
              and w.ent_iob_ != 'I' \
              # not a stop word
              and not w.is_stop]
    tokens += [de.string.rstrip().replace(' ', '_') for de in d.ents]
    return tokens

def dictokens(corpora):
    d = {}
    for j, dok in enumerate(corpora.keys()):
        for t in tokensfromdoc(corpora[dok]):
            d[t] = d.setdefault(t, [])
            d[t] += [j]
    return d
```

The above function `dictokens` makes a dictionary with lemmatized words as keys and document numbers as values. This can be used to make the term-document matrix as we did for the initial example.

[20]:
```
def tdmatrix(d, corpora):
    A = lil_matrix((len(d.keys()), len(corpora.keys())), dtype=int)
    for i, t in enumerate(d.keys()):
        for j in d[t]:
            A[i, j] = 1
    return A
```

[21]:
```
d = dictokens(c)
```

[22]:
```
d = dictokens(c)
A = tdmatrix(d, c)
Adf = pd.DataFrame(A.toarray(), index=d.keys(), columns=c.keys())
```

This array is now a bit too big to meaningfully display here, but here are a few elements of one row, which now displays the droneship name as a single token.

[23]:
```
Adf.loc[['Of_Course_I_Still_Love_You'], 'NYTimes':].T
```

[23]:
```
               Of_Course_I_Still_Love_You
NYTimes                                 0
May30NASAblog                           1
May31NYTimes                            0
2015Musk                                1
IainBanks                               1
```

XVII.5 Queries and retrieval

Returning to the question of information retrieval posed at the beginning of the lecture, let's consider how to handle queries. *Free text query*, is a form of query popular on internet searches, where query terms are typed in without any connecting operators. Query terms can be any collection of words extracted from the corpus. A query vector can be made by taking the mean of these query word vectors and normalizing it to a unit vector. (Again this is not a foolproof strategy, but it is a simple prescription that often works well.) The cosine separation between the query vector and each document vector is then computed. The most relevant documents are considered to be the ones that make the smallest angle with the query vector, so they are returned first in the output list. Here is a quick implementation suitable for small datatsets.

[24]:
```
def retrieve(querytokns, W, Vt, c):

    """Given a list of query word token numbers "querytokns",
    all words vectors "W"  and all document vectors "Vt.T"
    extracted from a corpus c, retrieve the documents
    relevant to the query. """

    q = W[querytokns, :].mean(axis=0)
    nrm = norm(q)
    q /= nrm
    idx = np.argsort(Vt.T @ q)[::-1]
    kl = list(c.keys())
    keys = [kl[i] for i in idx]
    docs = [c[k] for k in keys]
    return docs, keys, idx
```

To use this on our current corpus example, let's make the word and document vectors first.

[25]:
```
uu, ss, vvt = svd(A.toarray())    # SVD & rank k approximation
k = 4
U = uu[:, :k]; S = ss[:k];
Vt = vvt[:k, :]                   # Document vectors
W = uu[:, :k] @ np.diag(ss[:k])   # Word vectors
```

Here is an example of a query with two words, `astronaut` and `first`, and the first three matching documents generated by the above strategy.

[26]:
```
myquery = np.where((Adf.index=='astronaut') | (Adf.index=='first'))[0]
docs, keys, idx = retrieve(myquery, W, Vt, c)
docs[:3]
```

[26]:
```
['Nation's first astronaut launch to orbit from home soil in nearly a␣
 ↪decade',
 'SpaceX launch of NASA astronauts is postponed over weather',
 'The first stage of the SpaceX rocket has landed successfully on the␣
 ↪droneship,
Of Course I Still Love You.']
```

The first result has both search words, while the other two has one of the two search words. Below is another example, where somewhat surprisingly, a document without the query word (but certainly what we would consider a relevant document) is listed within the top three matches.

[27]:
```
myquery = np.where(Adf.index=='droneship')[0]
docs, keys, idx = retrieve(myquery, W, Vt, c)
docs[:3]
```

[27]:
```
['The first stage of the SpaceX rocket has landed successfully on the␣
 ↪droneship,
Of Course I Still Love You.',
 'These friends of yours are ships. Yes, both of them. What are they␣
 ↪called? Of
Course I Still Love You and Just Read The Instructions. They are not␣
 ↪warships?
With names like that?',
 'West Coast droneship under construction will be named Of Course I Still␣
 ↪Love
You']
```

Let me conclude this introduction to the subject of text analysis and information retrieval by noting that the concept of mapping words to vectors is finding increasingly significant uses, such as in automatic translation. I have tried to present ideas in minimal examples, but you should be aware that there are many extensions in the literature. Some extensions are easy to see as emerging from computational experience. An example is a generalization that we will see in an exercise that modifies the term-document matrix to account for the number of times a term occurs in a document. The resulting matrix will have frequency-weighted entries, not just 0 and 1 as above. This is built into scikit-learn's text analysis facilities, which we shall use in the exercise.

A Exercises

A.1 Exercise: Sum up integer powers

Task: Write a code to compute the value of

$$\sum_{n=1}^{N} n^i$$

for any integers i and N. (Solution codes will be ranked in terms of correctness, readability, and brevity.)

How do you know your answer is correct? When writing code it is important to check for correctness. Llementary mathematics tells us that

$$\sum_{n=1}^{N} n^2 = \frac{N}{6}(N+1)(2N+1).$$

(If you don't know this prove it!) So you can easily check that your code gives the correct answer, at least for $i = 2$. In fact, even for a general power i, power sums have been studied very well and expressions connecting them to the Riemann zeta function are well known, so for this task, there are indeed many sources to double check our code results.

Python has many styling guidelines for writing good code. You may want to peruse PEP 8 at your leisure. And take time to behold an easter egg (one of several) within the language:

[1]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

A.2 Exercise: Graphing functions

This exercise checks that you have learnt the basic usage of numpy and matplotlib.

```
[1]: import matplotlib.pyplot as plt
     import numpy as np
     %matplotlib inline
```

Task 1: Graph a function of one variable Plot the graph of $\sin(x)$ for x in the interval $[0, 10]$.

Task 2: Graph a function of two variables Plot the graph of $\cos(\sqrt{x^2 + y^2})$ for $(x, y) \in [-5, 5] \times [-5, 5]$.

A.3 Exercise: Passing function arguments

When programming mathematical algorithms, it is important to know if unnecessary copying is being done by your program. Copying large matrices is expensive. Consider this simple function, where `w` could be a matrix or a vector or a list or a string etc.

```
[1]: def twice(w):
         """Replace w by 2*w"""
         w *= 2
```

Consider a scenario where you make an object `v` and then send it to this function, like in this example:

```
[2]: v = [2, 5, 1]
     twice(v)
```

Task: Your task is to determine if `v` is being copied when you call `twice(v)` for some `v`. In other words, is a deep copy of `v` being implicitly made by python when you send `v` as an argument to `twice`? Answer this for at least two cases:

- `v` is a numpy array
- `v` is a string

Explain your observations.

A.4 Exercise: Piecewise functions

Task: Write an efficient numpy-based code for computing the values of

$$f(x) = \begin{cases} x\sin(x^2) & \text{if } x < 0, \\ \cos(x) & \text{if } x \geq 0 \end{cases}$$

at 1000001 uniformly spaced points in the interval $[-5, 5]$. Time it and then plot the function.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

A.5 Exercise: Row swap

Task: Consider the following simple python code to interchange rows `i` and `j` of a numpy array `A`.

[1]:
```
def swaprow(i, j, A):
    tmp = A[i, :]
    A[i, :] = A[j, :]
    A[j, :] = tmp
```

If there are problems with this (correctness? efficiency? elegance? brevity?), explain them, and produce a better function. (Please do check for correctness before you check anything else.)

A.6 Exercise: Averaging matrix

Task: Make an $n \times n$ matrix whose entries on the diagonal, k superdiagonals, and k subdiagonals, are one, and whose remaining entries are zero. E.g., for $k = 2$ and $n = 6$, the matrix looks like

```
[[1., 1., 1., 0., 0., 0.],
 [1., 1., 1., 1., 0., 0.],
 [1., 1., 1., 1., 1., 0.],
 [0., 1., 1., 1., 1., 1.],
 [0., 0., 1., 1., 1., 1.],
 [0., 0., 0., 1., 1., 1.]]
```

Now modify the unit entries in each row to another constant such that all the row sums of the matrix equal one. We shall call the resulting matrix an averaging operator A. After making the A matrix, do the following tasks: - Apply A to a vector x whose entries are $x_j = (-1)^j$, say for $n = 20$, and $k = 2$. and comment on the resulting vector.

- Apply A to a vector whose entries are values of $f(x) = x^2 + 2\sin(10x)$, say at $n = 1000$ equally spaced points in the interval [0, 10] and $k = 100$.

In both cases investigate the effect of varying k, report any edge effects, and discuss your observations.

A.7 Exercise: Differentiation matrix

Task: Make an $(n-1) \times n$ matrix D with the property that when it is applied to a vector $f \in \mathbb{R}^n$ we get $Df \in \mathbb{R}^{n-1}$ whose entries are

$$[Df]_i = f_{i+1} - f_i, \qquad i = 0, 1, \ldots, n-1.$$

Use the `diag` facility of `numpy` to make `D` fast. Put $x_j = jh$ for some positive grid spacing h. If f_j equals $f(x_j)$ for some differentiable function f, then $h^{-1}Df$ produces approximations to the derivative of f, so we shall call $h^{-1}D$ the differentiation matrix.

[1]:
```
import numpy as np
from numpy import diag
```

- Apply D to obtain an approximation of the derivative of $f(x) = \sin(x)$, plot the result, and verify that you get what you expect.
- Apply D to obtain an approximation of the derivative of $f(x) = x^2 + 2\sin(10x)$ for a thousand or more equally spaced values of x. Plot the result. Experiment with what happens if you add the averaging operator from the previous exercise into the mix.

Optional Extra Task: Install `scipy` if you don't have it already. Then make `D` as a sparse matrix (that doesn't store zeros) using the following facility in `scipy.sparse` module:

[2]:
```
from scipy.sparse import diags
```

Apply the sparse differentiation matrix to the functions described above and note if there are any performance gains.

A.8 Exercise: Pairwise differences

Task: Given a 1D numpy array x, produce the 2D numpy array D whose entries are

$$D_{ij} = x_i - x_j$$

in one line of code.

[1]:
```
import numpy as np
x = np.random.rand(5)
```

A.9 Exercise: Hausdorff distance

Task: Given two collections P and Q of points in the plane, compute the Hausdorff distance between sets P and Q. The Hausdorff distance between P and Q, denoted here by $H(P, Q)$, is defined as follows. Let

$$h(P, Q) = \max_{p \in P} \min_{q \in Q} \|p - q\|$$

where, for any $p \in \mathbb{R}^2$, the notation $\|p\|$ denotes the Euclidean distance $\sqrt{p \cdot p}$. Using this, the Hausdorff distance is defined by

$$H(P, Q) = \max\left[h(P, Q),\ h(Q, P)\right].$$

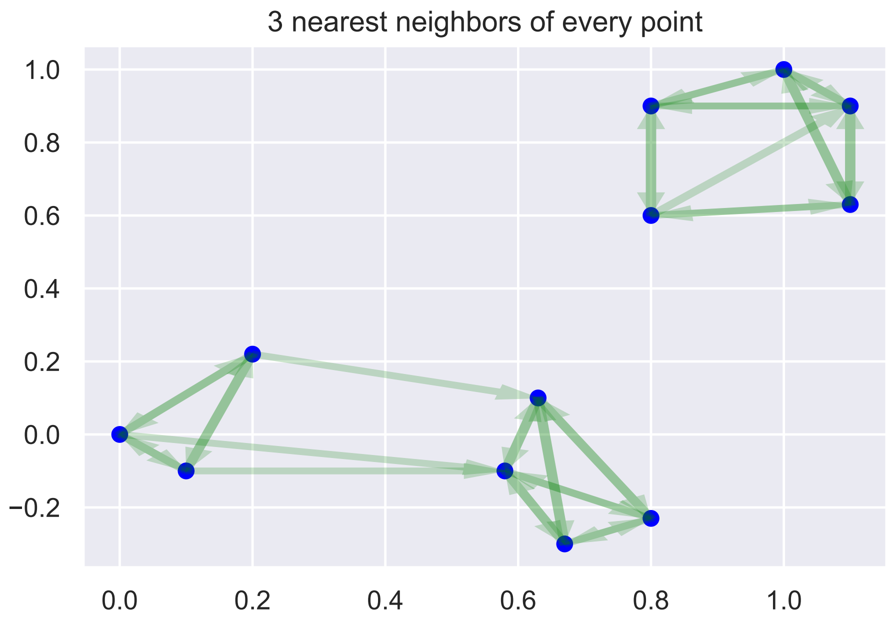
[1]:
```
import numpy as np
P = np.random.rand(5, 2)
Q = np.random.rand(7, 2)
```

A.10 Exercise: k Nearest Neighbors

Task 1: Write a function that finds, given a finite set of points in the plane and an integer k, the k nearest neighbors of each point in the set using numpy's vectorized facilities.

Task 2: Apply your function to this set of points with $k = 3$. Plot an arrow from each point to its k-nearest neighbors.

[1]:
```
import numpy as np

P = np.array([[0,0], [0.2, 0.22], [0.1, -0.1],
              [1,1], [1.1, 0.9], [0.8, 0.9], [1.1, 0.63],
              [0.58, -0.1], [0.63, 0.1], [0.67, -0.3], [0.8,-0.23],
              [0.8, 0.6]])
k = 3
```



A.11 Exercise: Predator-prey model

Suppose the populations of rabbits (denoted by $r(t)$) and foxes (denoted by $x(t)$) at time t in a jungle are modeled by the ODE system

$$\frac{dr}{dt} = \alpha r - \beta r x$$
$$\frac{dx}{dt} = \delta r x - \gamma x$$

where $\alpha = 1.1$, $\beta = 0.4$, $\delta = 0.1$, and $\gamma = 0.1$.

Task 1: Given initial conditions $r(0) = 5$ and $x(0) = 2$, solve for $r(t)$ and $x(t)$ and plot the solution for $0 \le t \le 70$.

Task 2: The phase plot of the solution consists of points $(x(t), r(t))$ for various t values. Prepare a figure (*phase portrait*) with phase plots of, say, 10 solutions, one each for randomly chosen initial values $r(0)$ and $x(0)$ between 1 and 9.

Task 3: This system has two equilibria. Solve for them and mark them in your phase portrait.

A.12 Exercise: Column space

Task 1: You have seen that the SVD of an $m \times n$ matrix A gives, among other things, a basis for the range (column space). Compute this for the given matrix.

Another way to obtain a basis for the range is using the QR factorization, also implemented in scipy. Carefully go through the linked QR documentation page. Then compute a basis for the column space of a given A using QR, and then using the SVD.

Task 2: Check that the column *spaces* (not the bases) you obtained in the two ways are the same. (How would you check that two given bases span the same space?)

Task 3: For a 500×500 random matrix, which method is faster?

[1]:
```
import numpy as np
from scipy.linalg import svd, qr

A = np.array([[1, -2, 3, -3], [2, -4, 9, -2], [-3, 6, -9, 9]])
```

A.13 Exercise: Null space

Task 1: Find the null space of the given matrix A using SVD.

Task 2: Find the null space of the same matrix A using the QR factorization. Use the linear algebra theorem that tells us that the null space of A is equal to the orthogonal complement of the range of the transpose A^t. (How would you extract the orthogonal complement from a full QR factorization?)

Task 3: Check that both answers above span the same space.

```
[1]: import numpy as np
     from scipy.linalg import svd, qr

     A = np.array([[1, -2, 9, 5, 4,], [1, -1, 6, 5, -3], [-2, 0, -6, 1, -2],␣
      ↪[4, 1, 9, 1, -9]])
     A
```

```
[1]: array([[ 1, -2,  9,  5,  4],
            [ 1, -1,  6,  5, -3],
            [-2,  0, -6,  1, -2],
            [ 4,  1,  9,  1, -9]])
```

A.14 Exercise: Pandas from dictionaries

```
[1]: import pandas as pd
```

While numpy arrays have an *implicitly defined* integer index used to access the values, the pandas objects have an *explicitly defined* index associated with the values, a feature shared with the python dictionary construct. To begin our pandas exercises, start by converting dictionaries to pandas objects.

Tasks:

1. Convert d0 to a corresponding pandas object pd0.

```
[2]: d0 = {2:'a', 1:'b', 3:'c'}
```

2. Sort indices of pd0.

(Note: Newer versions of pandas do not sort dictionary keys.)

3. Convert d1, d2 (together) to a pandas object dd.

```
[3]: d1 = {'a': 1, 'b': 2}
     d2 = {'b': 3, 'c': 4}
```

4. Give examples of indexing and slicing on dd.

In pandas, *indexing* refers to accessing columns by their names, in a syntax reminiscent of dictionary access by keys, while *slicing* refers to row access like in numpy.

5. Give examples of implicit and explicit indexing on dd.
6. Forward fill and backfill missing values using various axis.

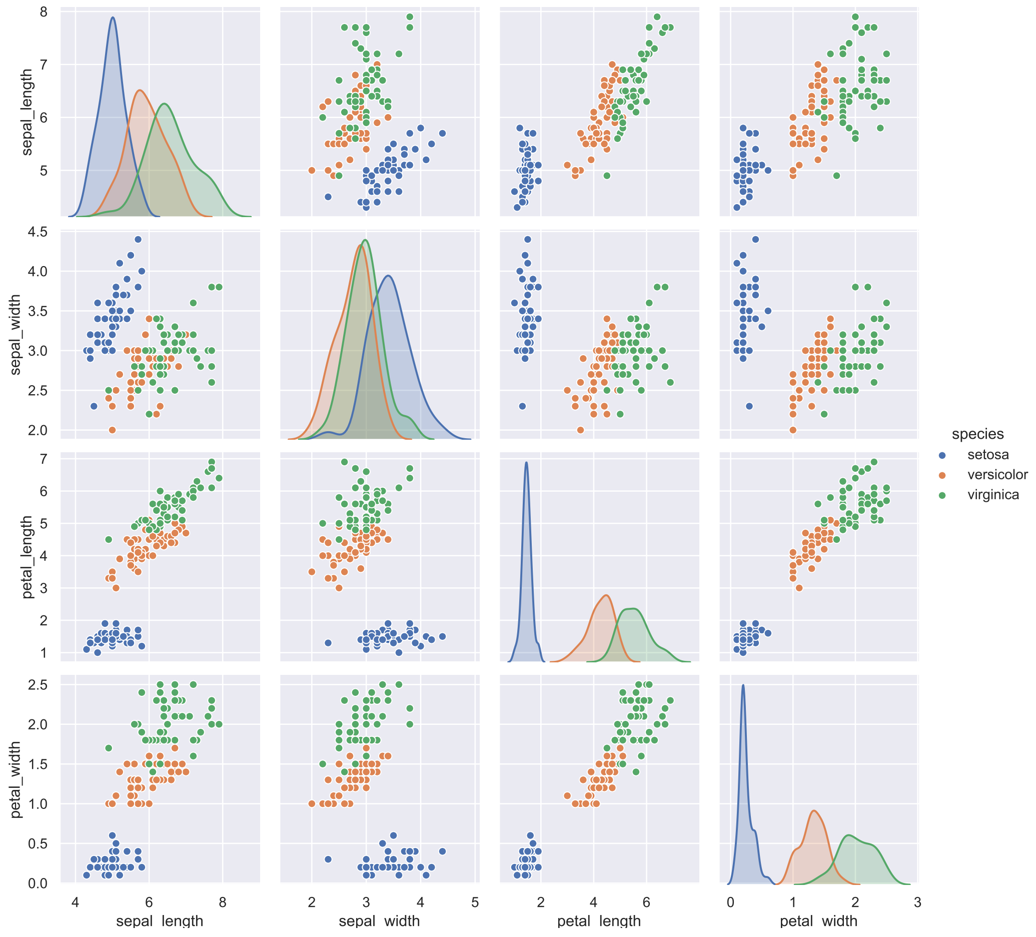
A.15 Exercise: Iris flower dataset

The statistical visualization package seaborn comes with the famous Iris Flower Dataset. Your exercise is to give one-line codes to address the two tasks mentioned after the figure below.

[1]:
```
import pandas as pd
%matplotlib inline
import seaborn; seaborn.set()

iris = seaborn.load_dataset('iris')
```

The following figure is generated by

```
seaborn.pairplot(iris, hue='species');
```



Task 1: Make a bar plot of the mean sepal sizes for each species

Task 2: Find the min, max and mean of petal sizes for each species

A.16 Exercise: Stock prices

```
[1]: import pandas as pd
     %matplotlib inline
```

If you have installed `pandas_datareader`, please download current stock prices using this:

```
import pandas_datareader.data as web
s = web.DataReader(['AAPL', 'GOOG', 'TSLA'], data_source='yahoo', start='2020')
```

Alternately, load from a file where data from until yesterday was saved. Download the file from D2L and move it to the right place in order for the following cell to work.

```
[2]: s = pd.read_pickle('../../data_external/stock_prices.pkl')
```

In either case, you will end up with a data frame `s` which contains three categories of prices for three tech stocks.

Tasks:

- Find out if a heirarchical indexing (`MultiIndex`) is being used in this data.
- Access and plot the closing price of `AAPL` on all days in the data.
- Print the closing price of all three stocks yesterday.
- Extract a smaller data frame with no `MultiIndex` containing only `TSLA` data.

A.17 Exercise: Passengers of the Titanic

The Titanic, with over 2000 passengers on board, including hundreds of emigrants to the US, as well as some of the world's richest, sank in 1912. The `seaborn` library provides a smaller-sized, anonymized data set of Titanic's passengers. Without identifying information, we can't tell the poor immigrant from the wealthy, yet the data manages to tell a story in other ways. Your task in this exercise is to answer a series of questions from the data, beginning with the mundane and ending with who survived.

```
[1]: import numpy as np
     import pandas as pd
     import seaborn
```

```
[2]: t = seaborn.load_dataset('titanic')
     t.head()
```

```
[2]:    survived  pclass     sex   age  sibsp  parch     fare embarked  class  \
     0         0       3    male  22.0      1      0   7.2500        S  Third
     1         1       1  female  38.0      1      0  71.2833        C  First
     2         1       3  female  26.0      0      0   7.9250        S  Third
     3         1       1  female  35.0      1      0  53.1000        S  First
     4         0       3    male  35.0      0      0   8.0500        S  Third
     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
1  woman       False    C    Cherbourg   yes  False
2  woman       False  NaN  Southampton   yes   True
3  woman       False    C  Southampton   yes  False
4    man        True  NaN  Southampton    no   True
```

Tasks: The exercise is to answer the following questions.

- How many passengers are described in the data set?
- How many distinct values are in `who` column?
- How many missing values do you find in each data column?
- Does the data contain passengers over 60 old? How many?
- What is the passenger age distribution? (Plot it.)
- What are the 3-quantiles of the passenger age distribution?

(Finite samples are divided into q subsets of nearly equal sizes by q-quantiles. The 2-quantile is the median.)

- How will you drop all passengers with no `embarked` data?
- What is the average, minimum, and maximum fares paid by the passengers?
- What are the proportions of passengers in different classes?
- What is the female to male ratio in each travel class?
- What fraction survived?

(This fraction is sometimes called the *survival rate* - although it is an improper name in the sense that there is no "rate" to speak of here; the question is to compute a dimensionless fraction.)

- Are the survival rates of male and female passengers different?
- Are the survival rates of first, second, and third class passengers different?
- How can one print a table of survival rate dependencies on class and gender?
- How can one print a table with number of survivors and average fare for each gender and cabin?

A.18 Exercise: Animation

Let x represent a point in the spatial interval $[0, 10]$, let $t > 0$ represent time, and let

$$f(x,t) = \frac{1}{2}(x+t)^2 + 2\sin(10(x-t)).$$

The following tasks are to be completed in a `.py` python file (not in a jupyter notebook).

Task 1: Use `matplotlib.animation` module to display changes in the plot of f over x with respect to time t.

Task 2: Use the `celluloid` module to perform the same task. Which is faster? Which is more convenient?

Task 3: Add a text labeling the time at each snapshot that the animation is composed of.

A.19 Exercise: Insurance company

An insurance company starts with \$1,000 dollars in its reserve. The company earns \$100 per day which is added to the reserve every day. However, the insurance company is engaged in very risky business: each day, with probability $q = 0.46$, the company may receive a claim, which will require it to pay \$200 from its reserve the day it receives the claim.

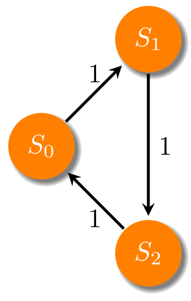
Question 1: What is the probability that the insurance company will run out of its reserve eventually and be ruined?

(Hint: The situations of the gambler G of [Gambler's Ruin] and the insurance company are not that different: at the end of each game, gambler G is up or down a chip; and at the end of each day, the insurance company reserve is up or down by \$100. When G has no chips to play, G is ruined; when the insurance company's reserve drops to \$0, it is ruined.)

Question 2: What should be the company's reserve in order to make the probability of the company's ruin less than 0.1%?

A.20 Exercise: Probabilities on small graphs

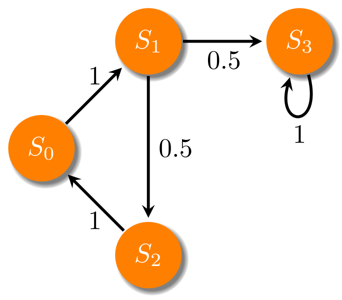
Consider random walks on the following small graphs with the indicated probabilities and answer the questions posed.



$$\begin{tikzcd}
 & S_1 \arrow[dd, "1"] \\
S_0 \arrow[ur, "1"] & \\
 & S_2 \arrow[ul, "1"]
\end{tikzcd}$$

Task 1:

What is the probability that you can hit one state from another state (in any number of steps) in this Markov chain? Answer by looking at the graph and double check that an eigenvector gives what is expected.

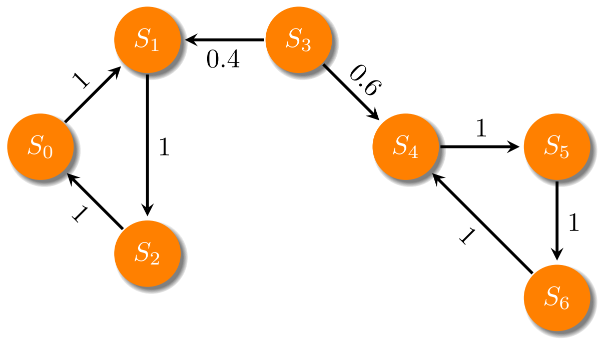


Task 2:

For each question below, guess the answer from the figure, and then verify that your computational method gives the expected answer.

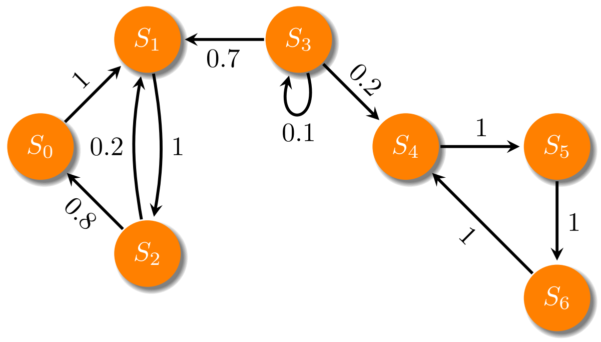
- Starting from $B = \{3\}$, what is the probability of hitting $A = \{0, 1, 2\}$ in any number of steps?
- Starting from from any state in $B = \{0, 1, 2\}$ what is the probability of hitting $A = \{3\}$ in any number of steps?
- Is this an absorbing Markov chain? Verify that the answers come out the same using the two methods you learnt.

Task 3:



- What are the hitting probabilities of $A = \{0, 1, 2\}$ from the remaining states?
- What are the hitting probabilities of $A = \{4, 5, 6\}$ from the remaining states?
- What are the hitting probabilities of $A = \{3\}$ from the remaining states?

Task 4:



- Starting from 1 what is the probability of hitting 0 in any number of steps?
- Starting from 3 what is the probability of hitting 6 in any number of steps?

A.21 Exercise: Ehrenfest thought experiment

The following thought experiment is well-known in physics. Begin with a box that is divided into two equal halves. Each half contains many air molecules. Perform the following experiment repeatedly: in each step, a molecule is chosen at random from one half and moved to the other half. If we start with unequal number of molecules in each half, what happens in the long run?

In probability texts, this experiment is conducted with "Ehrenfest Urns". There are two urns that together contain $2k$ balls. At each step, one of the balls is chosen at random and moved from its urn to the other urn.

Task 1: Model the process as a Markov chain. Choose as states the number of balls in the first urn. Write a function to make the transition matrix P for general k. Print out your transition matrix for $k = 2$ (which should be 5 x 5). Draw the directed graph of the chain for $k = 2$ case.

Task 2: Is P irreducible?

Task 3: Does P^n converge as $n \to \infty$?

Task 4: Plot the stationary distribution of this Markov chain for $k = 100$.

A.22 Exercise: Power method for large graphs

In the lecture, we used the `eig` function to compute pagerank. Since this is unsuitable for large graphs, we pursue an alternate idea, also from the lecture: the pagerank vector, being the eigenvector of the dominant eigenvalue of a positive stochastic matrix, is the limit of the sequence

$$x, \quad (P^t)x, \quad (P^t)^2x, \quad (P^t)^3x, \quad \ldots,$$

which we can computationally approach. This is an instance of the *power method*, a topic well studied in numerical analysis. This method only needs to *apply* the matrix P^t repeatedly (and has no need for other operations found in general eigensolvers that carry more memory overhead). This exercise shows you the practical problems with using a general eigensolver as the graph size increases and asks you to implement the power method to be able to solve on large graphs.

Task 1: Compute the stationary distribution of the Markov chain from Gambler's ruin with $p = 0.4, N = 10$. (Do this task with `eig`.) Do you get more than one stationary distribution?

Task 2: Consider the adjacency matrix (independent of p) of the random walk and introduce a restart probability. Using it, compute the pagerank for all states of the Markov chain with $N = 10$ and restart probabilities $r = 0.1, 0.01, 10^{-3}$, and 10^{-4}.

Task 3: Compute the pagerank of the ruin state for $r = 0.1, N = 1000$. How much farther can you go increasing N in your computing environment, while continuing to use `eig`?

Task 4: Implement the power method for a positive stochastic matrix P as a python function, with the inputs indicated below.

```python
def powerP(x, aPt, r=0.1, maxn=1000, tol=1e-10):
    """ Apply power iterations to a positive stochastic matrix P.
    INPUTS:
     - x: initial probability distribution for the power method,
     - aPt: function that returns P.T @ x given x
     - r: restart probability,
     - maxn: apply at most 'maxn' power iterations
     - tol: quit if successive iterations differ by less than 'tol'.
    OUTPUT: Vector of pageranks if converged. """
```

This function should start with a random initial probability distribution x and compute $(P^t)^n x$ for increasing n until $\|(P^t)^n x - (P^t)^{n-1} x\|$ becomes smaller than a given input tolerance. To save memory and flops, you should *not create the transition matrix P in memory*, rather, you should make a *function* that applies P^t to a vector, and pass that function as one of the arguments `aPt` to the `powerP` function.

Apply your function to compute the pagerank of the ruin state for $r = 0.1, N = 100000$.

A.23 Exercise: Google's toy graph

At SNAP (Stanford Network Analysis Project), you will find a large graph data set (with over 5 million edges) called the *Google Web Graph*. (This graph was released by google for a programming competition.) Download this graph, examine the file, and guess the format. You will need to load this data into your computer's memory to solve this exercise. Think carefully about what tools you would use so as not to run out of memory.

Task: Setting restart probability $r = 1 - 0.85$, compute the pageranks of all vertices on this graph. Reuse the power method function you wrote in Exercise: Power method for large graphs.

[1]:
```python
import os
import urllib
import shutil
import numpy as np
```

[2]:
```python
# The file is located here:
url = "https://snap.stanford.edu/data/web-Google.txt.gz"

# Download and copy it here using the code below:
f =  '../../data_external/web-Google.txt.gz'

if not os.path.exists(f):
    r = urllib.request.urlopen(url)
    fo = open(f, 'wb')
    shutil.copyfileobj(r, fo)
    fo.close()
```

A.24 Exercise: Atmospheric carbon dioxide rise

Task: Reconsider the problem of the assignment on rising atmospheric CO_2 levels. Using the data you already downloaded for the assignment, now fit a quadratic curve to the data by regression. Since the derivative of a quadratic function can be hand-computed, you would then be able to estimate the rate of change of the regression fit and address the earlier assignment task: provide a policy-maker with a "yes or no" answer on whether the *rate of increase of* CO_2 is increasing.

A.25 Exercise: Ovarian cancer data

Download and copy the datafile `ovariancancer.npy` into `data_external` folder. This file contains data of 216 patients, the first 121 of which have ovarian cancer, and the remaining 95 do not. For each patient, expressions of some biomarkers through 4000 spectroscopic measurements are provided. The original data source is ccr.cancer.gov. The data is also packaged together with Matlab® and they maintain an online documentation page on it. High-dimensional biological and genetic datasets are often highly correlated, i.e., patients can be expected to have significant overlap in genes and biomarkers. Therefore such datasets will generally benefit from PCA and dimensional reduction. In this exercise, you will work with a realistic dataset which exemplifies such a dimensional reduction.

```
[1]: import matplotlib.pyplot as plt
     from mpl_toolkits.mplot3d import Axes3D
     %matplotlib inline
     import numpy as np
     X = np.load('../../data_external/ovariancancer.npy')
     X.shape
```

```
[1]: (216, 4000)
```

Task 1: Project the 4000-variable data into its first 3 principal components and view the projections in a three-dimensional plot.

Task 2: Plot the cumulative explained variance for this dataset. What is the percentage of variance lost in restricting the data from 4000 to 3 dimensions? How many dimensions are needed to keep 95% of the variance?

A.26 Exercise: Eigenfaces

In this exercise you will apply PCA to a large library of facial images to extract dominant patterns across images. The dataset is called *Labeled Faces in the Wild*, or LFW, (source) and is popular in computer vision and facial recognition. It is made up of over a thousand 62 x 47 pixel face images from the internet, the first few of which are displayed below.

```
[1]: import matplotlib.pyplot as plt
     %matplotlib inline
     import numpy as np
     from sklearn.datasets import fetch_lfw_people      # this will download␣
      ↪images if
     faces = fetch_lfw_people(min_faces_per_person=60) # you don't already have␣
      ↪them
     fig, ax = plt.subplots(4, 7, figsize=(12, 10))
     for i, axi in enumerate(ax.flat):
         axi.imshow(faces.images[i], cmap='pink')
         axi.set(xticks=[], yticks=[], xlabel=faces.target_names[faces.
      ↪target[i]])
```

Task 1: We refer to the principal components of face image datasets as *eigenfaces*. Display the first 28 eigenfaces of this dataset. (They will have little resemblance to the first 28 images displayed above.)

Task 2: Let N be the least number of dimensions to which can you reduce the dataset without exceeding 5% relative error in the Frobenius norm. Find N. (This requires you to combine what you learnt in the SVD lecture on the Frobenius norm of the error in best low-rank approximation with what you just learnt in the PCA lecture.)

Task 3: Repeat PCA, restricting to N eigenfaces (with N as in Task 2), holding back the last seven images in the dataset. Compute the representations of these last seven images in terms of the N eigenfaces. How do they compare visually with the original seven images?

Task 4: Restricting to only images of Ariel Sharon and Hugo Chavez, represent (and plot) them as points on a three-dimensional space whose axes represent the principal axes 4, 5, and 6. Do you see the points somewhat clustered in two groups? (The principal directions 0, 1, 2, 3 are excluded in this task since they seem to reflect lighting, shadows, and generic facial features, so will likely not be useful in delineating individuals.)

A.27 Exercise: Word vectors

The following corpus contains statements by two Republican presidents, a quote from the bible, and three quotes from the internet.

[1]:
```
c = {                                                     \
'Lincoln1865':
'With malice toward none, with charity for all ...'  +
'let us strive on to finish the work we are in ... ' +
'to do all which may achieve and cherish a just and lasting peace, ' +
'among ourselves, and with all nations.',

'TrumpMay26':
'There is NO WAY (ZERO!) that Mail-In Ballots ' +
'will be anything less than substantially fraudulent.',

'Wikipedia':
'In 1998, Oregon became the first state in the US ' +
'to conduct all voting exclusively by mail.',

'FortuneMay26':
'Over the last two decades, about 0.00006% of total ' +
'vote-by-mail votes cast were fraudulent.',

'TheHillApr07':
'Trump voted by mail in the Florida primary.',

'KingJamesBible':
'Wherefore laying aside all malice, and all guile, and ' +
'hypocrisies, and envies, and all evil speakings',
}
```

Task 1: Use scikit-learn's `CountVectorizer` to make the term-document matrix, particularly noting what the rows and columns correspond to (and compare with the LSA lecture). Display it as a data frame labeled with words and document keys. Does `CountVectorizer` lemmatize the words?

Task 2: Combine `CountVectorizer` (see its doc string for help) with a tokenizer function you write using spacy's lemmatization (per what you learnt in the LSA lecture). Remake the term-document matrix. Display your answer. (Your matrix size will depend on whether you used `stop_words='english'` argument of `CountVectorizer`, and may even depend on which version of spacy you are using, since lemmatization has changed across versions.)

Task 3: Use LSA to compute three dimensional representations of all documents and words using your term-document matrix from Task 2. Print out your vector representation of `vote` (which will obviously depend on the matrix).

Task 4: Write a function to compute the cosine of the angle between the spans of two word vectors. Compute the cosine of the angle between `malice` and `vote`. Compute the cosine of the angle between `mail` and `vote`.

Task 5: In order to moderate the influence of words that appear very frequently, the TF-IDF matrix in often used instead of the term-document matrix. The term frequency-inverse document frequency (TF–IDF) matrix weights the word counts by a measure of how often they appear in the documents according to a formula found in scikit-learn user guide. Compute the TF-IDF matrix for the above corpus using `TfidfVectorizer`.

Task 6: Recompute the two cosines of Task 4, now using the TF-IDF matrix of Task 5 and compare.

[2]:
```
from sklearn.feature_extraction.text import CountVectorizer,␣
 ↪TfidfVectorizer
import pandas as pd
```

B Projects

B.1 Assignment: Bisection Method

Your task is to implement the *bisection method* for finding a solution x of the equation

$$f(x) = 0.$$

Here f is a real-valued function of a single real variable x and the solution of the above equation is called a *root* of f.

Many nonlinear algebraic equations, such as $x = 1 + \cos x$ do not admit a closed form solution. But a numerical method can find an approximate solution by finding the root of $f(x) = x - 1 - \cos x$.

Bisection is a numerical method to solve for a root of a function $f(x)$ of a single real variable x. Here is its description:

The Bisection method

1. Start with an interval $[a, b]$ in which $f(x)$ changes sign.
2. Then there must be (at least) one root in $[a, b]$.
3. Halve the interval and set the midpoint $m = (a + b)/2$.
   - Does f change sign in left half $[a, m]$?
   - If Yes: Repeat with the left interval $[a, m]$ (set $b = m$)
   - If No: Repeat with the right interval $[m, b]$ (set $a = m$)
4. At the nth step, the initial interval $[a, b]$ has been halved n times and we know that $f(x)$ must have a root inside a small subinterval of length $2^{-n}(b - a)$. Since the root is contained in this subinterval, error $\leq 2^{-n}(b - a)$.
5. Hence we may stop the subdivisions when n is such that

   $$2^{-n}(b - a) \leq \epsilon.$$

   for some user specified error tolerance ϵ, and take the midpoint m as the root.

Hints and suggestions

1. Write down your steps as a precise algorithm (before you code) in terms of `for`/`while`, `if`, `else`, etc. Use this to map out how you will write your code.
2. Write a first version of the code and make sure it is working on a test problem. Your code should be in the form of a function

```
def bisection(f, a, b, eps, niters):
    # Code goes here.
```

where `f, a, b,` and `eps` represent f, a, b and ϵ in the above description, and `niters` is the maximal number of iterations, which in this case is the maximal number of subdivisions of the initial interval you will make (whether or not the `eps` tolerance is met).

3. Test your bisection code on the function $f(x)$ whose roots you know, say $f(x) = \cos(x)$. Once you know your current code is working correctly, proceed to the next step. If you jump this step, beware that "premature optimization is the root of all evil," according to Donald Knuth.

4. Refactor/improve/optimize: When halving the interval, can you reuse a previously used value of f to make the code more efficient? (This would be important when the evaluation of f is expensive.) Also have you made sure you have included comments? Does you function have a docstring?

B.2 Assignment: Rising CO_2 levels in the atmosphere

Mauna Loa Observatory is located over 3000 meters above the sea level, on the Big Island of Hawaii. NOAA, the National Oceanic and Atmospheric Administration, runs this facility, and has been collecting data on the composition of our atmosphere for years. The rise of CO_2 in our atmosphere is one of the drivers of climate change as CO_2 is a heat-trapping gas. NOAA measures CO_2 levels at the observatory and has made its data available for all.

Task

In this assignment, your task is

1. to download this data from within python using `urllib`,
2. do the necessary data munging to get the data into arrays,
3. extract the monthly averages of measured CO_2 from a data column named `average`,
4. plot the monthly averages as a function of time,
5. estimate the rate of change of CO_2 from this data, and
6. plot your estimate of the rate of change as a function of time.

The last two items require you to experiment with imperfect techniques to estimate rate of change (imperfections that we saw in previous class activities). Please conclude your assignment with what your answer would be if a policy-maker wants a "yes or no" answer on whether the *rate of increase of CO_2* is increasing.

Your product for grading should be a jupyter notebook, written clearly (like a term paper), including code and graphs, and with explanations of all your steps to arrive at your graphs and conclusions.

Hints

- This is where the data is available for download:

ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_mm_mlo.txt

- The data has gaps (read the file header) which you can easily remove using numpy's masking facilities.

- The data may not be perfectly equispaced. Use the function `interp1d` from `scipy.interpolate` to generate values at equispaced time intervals (after installing scipy).
- You will see that although CO_2 data has a rising trend, the curve is filled with small oscillations. Use your experience from exercises or any other tools you know to calculate rate estimates of the overall trend.

B.3 Assignment: Growth of confirmed COVID-19 cases

Task

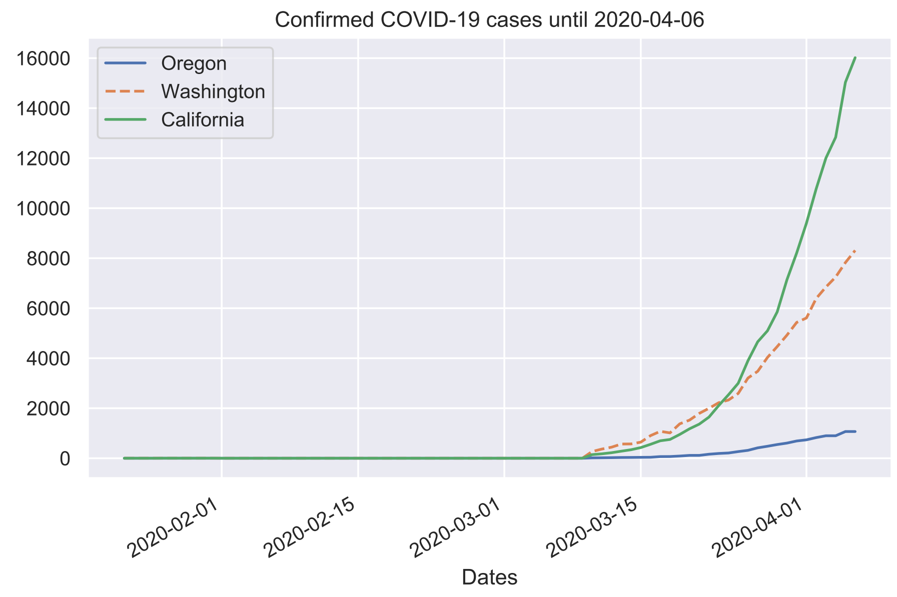
Your assignment is to plot a time series of confirmed COVID-19 cases in Oregon, Washington, and California, using data from [JHU-CSSE].

Start by copying the required code to download/update the data from the Overview lecture. Then see what files you would need to examine to get the data to accomplish your goal.

Product for grading

Turn in one `.ipynb` file containing a jupyter notebook explaining how you produced the plot. Also turn in one `.png` file of your final plot.

If you were to finish the assignment today, your plot would look like this:



But, of course, the plot you turn in should be the plot obtained that day. And none of us know yet how that would look like.

B.4 Assignment: World map of COVID-19

Task

Your task is to make a chloropleth map visualizing COVID-19 cases worldwide. The countries of the world should be displayed in the Albers equal area projection. The number of cases in each country should be indicated by a reasonable color scale (of your choice).

Please submit three files:

1. A `.png` image file of the chloropleth map using the latest available data on the submission day.
2. A `.py` python file (**not** jupyter notebook) which generated your `.png` image file.
3. A `.mp4` movie file containing an animation of the cholorопleth maps from 01/22/2020 through the latest date of the data and a `.py` python file that you used for creating the animation.

Hints

- The last task of making the `mp4` movie file is harder than the other two. Begin with the easier tasks.
- An example of a chloropleth map is in the Overview lecture.
- We have already seen the Albers equal area projection in the lecture Visualizing geospatial data.
- For animation, use your experience from the exercise on animations.
- Use the Johns Hopkins dataset.

An example solution with data until May 7, 2020, can be seen in the output of the next code cell if you are reading this in a jupyter notebook.

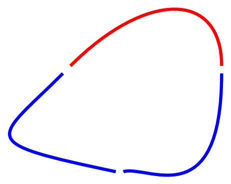
```
[1]: from IPython.display import Video
     Video("../figs/covidworldmapanim.mp4")
```

```
[1]: <IPython.core.display.Video object>
```

Alternately, the same solution video can be downloaded or visualized at this weblink.

B.5 Assignment: Neighbor's color

Task 1: Consider a simple closed curve divided into n pieces (arcs), each of which is colored in either red or blue. In each iteration, one of the arcs is chosen at random. The selected arc then chooses one of its neighbors at random and adopts that neighbor's color.



Consider $n = 3$ first (as in the figure above). Model the process as a Markov chain. The states consist of all the possible color configurations on the arcs. Answer these questions:

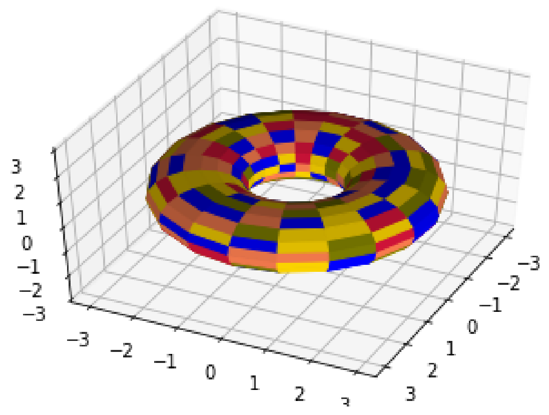
- How many states are there?
- Draw the directed graph of the Markov chain.
- What are the absorbing states?
- Is this an absorbing Markov chain?

Task 2: Now, consider a general n instead of $n = 3$.

- How many states are there?
- Write a python function to compute the probabilities of eventually hitting the absorbing states for a general n. (Use vectorized operations as much as possible.)

Task 3: Finally, consider the generalization of the above setting from a closed curve to the surface of a torus. The toroidal surface is divided into $n \times n$ rectangles, each of which has one of k colors (see the figure below for an example). The process generalizes to selecting one of these rectangles at random, the chosen rectangle then adopting a color from one of its *8 neighbors*, and then repeating.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
fig = plt.figure(); ax = fig.gca(projection='3d')
angs = np.linspace(0, 2.*np.pi, 20)
theta, phi = np.meshgrid(angs, angs)
x = (2 + np.cos(theta)) * np.cos(phi)
y = (2 + np.cos(theta)) * np.sin(phi)
z = np.sin(theta)
rng = np.random.default_rng()
randind = rng.integers(5, size=x.shape)
colors = np.array(['crimson', 'coral', 'gold', 'blue', 'olive'])[randind]
ax.plot_surface(x, y, z, facecolors=colors, linewidth=1, edgecolors='k')
ax.view_init(46, 26); ax.set_zlim(-3,3);
```



Again, model the process as a Markov chain. This Markov chain arises in population genetics. It shows you a manifestation of the *combinatorial explosion* (or the *curse of dimensionality*) that makes computation by the standard technique quickly infeasible.

- How many states are there for a given n and k?
- Is it feasible to extend the python function you wrote in Task 2 to compute the absorption probabilities for this Markov chain, say for $k = 2, n = 10$?
- Imagine cutting, unfolding, and stretching the toroidal surface to a square with an $n \times n$ grid of color cells, respecting the *boundary identifications* inherited from the torus. Any state of the Markov chain can thus be implemented as a 2D integer numpy array (each entry taking one of k values, representing the colors). Write a python function to simulate the process for a general n and k using numpy's random module. You should see colors dispersing, coalescing, migrating etc, with the process eventually terminating in an absorbing state.
- Create an animation displaying the sequence of states obtained in one call of your function, say with $k = 4, n = 10$, starting from some initial state of your choice. (Be warned that there are random sequence of states that are too long to fit in memory even for small n and k, so build in a fail-safe to avoid a computer crash.) Render the animation either on the toroidal grid or on the equivalent flat $n \times n$ grid.

An example of a solution animation can be viewed below:

```
[2]: from IPython.display import Video
     Video("../figs/diversityloss.mp4", width=500)
```

```
[2]: <IPython.core.display.Video object>
```

If the video does not render on your reading device, you may download it from this weblink.