These labs introduce Python. Through companion exercises, we will also become acquainted with Arduino Processing (a variant of C), C#, and C++. All of these languages are expressive and powerful programming tools, and they are relatively easy to learn. Through lectures and practice, you will become, over the course of the semester, an increasingly competent programmer. In these labs you will also be introduced some of the foundations of computer science, including what have come to be called “computational thinking” and “data science.” We hope that you will also enjoy the challenges that these labs present. Most of the labs use Python, in part because Python has repeatedly been shown to be an excellent vehicle for introducing computing, but also because Python is fast becoming the programming language of choice for scientific and engineering computation. Over your career, you will encounter many programming languages, and you will find that some languages are better than others for certain kinds of programming problems. We encourage you to take a language-agnostic view of programming, learning and using new programming languages as they offer features that you find useful.

You will quickly notice that we have borrowed from many different on-line resources in the creation of these assignments. We have done this to provide a diverse set of exercises, but also to help make you aware of the vast array of on-line resources to help you learn.

A Word to the Advanced Student - One of the problems in any course is that people come in with different amounts of background, so defining activities at the right level for everyone is hard. We expect that some of you will already know quite a bit about programming and computer science. In designing these labs, we have tried to choose activities that will be valuable to people at all levels, though in different ways. Please let us know if you feel that we are asking you to do something that does not seem worthwhile, or if some modification would make a task more useful for you.

Lab 1

Programming Exercise

Complete both of the following assignments:

1. Python is used widely in “bioinformatics,” the area of computer science that explores and manipulates the data associated with molecular biology. Rosalind is an online platform for learning bioinformatics through problem solving. Rosalind offers an engaging way to learn Python, while simultaneously exploring bioinformatics. The Rosalind site also offers several introductory Python programming exercises.
   a. Create a Rosalind account (this is free and will be used for several assignments) at [http://rosalind.info/accounts/register/](http://rosalind.info/accounts/register/).
   b. Complete Rosalind Python Village ini1 (Installing Python; do this exercise even if you already have Python installed) and ini2 (Variables and Some Arithmetic): [http://rosalind.info/problems](http://rosalind.info/problems)

Be sure to read the “click to expand” section at the top of the page.

   c. Submit both your working Python code, and a snapshot from the Rosalind web site demonstrating your successful completion of this assignment, e.g.,
2. A “recurrence relation” is a way of defining the terms of a sequence of numbers with respect to the values of previous terms. The Fibonacci numbers are a well-known recurrence relation named after Leonardo of Pisa, who was known as “Fibonacci.” Fibonacci’s 1202 Latin text *Liber Abaci* introduced what have come to be called “Fibonacci numbers” to Western European mathematics, although mathematicians in India had written of the sequence hundreds of years before Fibonacci. Fibonacci sequences are found ubiquitously in nature at microscopic (e.g., various cell structures, and cancer cell division), everyday (e.g., pine cones, chicken eggs, aloe plants, and hurricanes) and galactic (e.g., spiral galaxies) levels. Search online for images of “Fibonacci sequence in nature” for some beautiful examples.

The following recurrence relation defines Fibonacci numbers $F_n$:

$$
\begin{align*}
F_0 &= 1 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2} \quad (\text{for } n \geq 2)
\end{align*}
$$

a. Write a simple Python program that prints the first 12 Fibonacci numbers. Do not use recursion or loops (and don’t worry if you do not yet know what recursion or loops are).

**Lab Homework**

1. Read next week’s lab exercise.

2. Python has a very active user community that has collectively developed a wide variety of online resources for learning Python. Explore some of these options (taken from [https://wiki.python.org/moin/BeginnersGuide](https://wiki.python.org/moin/BeginnersGuide)):

- If you have never programmed before, see [BeginnersGuide/NonProgrammers](https://wiki.python.org/moin/BeginnersGuide/NonProgrammers) for a list of suitable tutorials.
- If you have previous programming experience, consult [BeginnersGuide/Programmers](https://wiki.python.org/moin/BeginnersGuide/Programmers), which lists more advanced tutorials.
- If English is not your first language, you might be more comfortable with a tutorial that has been translated into your language. Consult python.org’s [list of Non-English resources](https://wiki.python.org/moin/LanguageResources).
Further Exploration

All data used in computation is ultimately represented in binary form, i.e., only “bits” of ones and zeros are used to represent integers, strings, decimal numbers, fractions, etc. Explore how integers are represented and manipulated in Python by working through the examples at:

Lab 2

Programming Exercise

Complete both of the following assignments:

1. Continue your exploration of Python:
   a. Complete Rosalind Python Village ini3 (Strings and Lists):
      [http://rosalind.info/problems/ini3/](http://rosalind.info/problems/ini3/)
      Be sure to read the “click to expand” section at the top of the page.

2. This exercise will begin to explore the some of the fascinating computer science problems to be found in bioinformatics. We will take advantage of Rosalind, “a platform for learning bioinformatics through problem solving,” for this purpose. Rosalind is named for Rosalind Elsie Franklin, whose work as a chemist and X-ray crystallographer was central to the understanding of the molecular structures of DNA and RNA. In particularly, Rosalind Franklin’s images allowed James Watson, Francis Crick, and Maurice Wilkins to determine of the structure of deoxyribonucleic acid (DNA), for which they received the received the Nobel Prize in Physiology or Medicine in 1962. There is considerable debate about why Franklin did not share in this prize. The common answer is that Franklin died in 1958, at the age of 37, and the Nobel Prize is not awarded posthumously (although the rule against posthumous awards was not made “official” until 1974).

   **Your assignment:**
   a. Complete the “Counting DNA Nucleotides” problem at
      Be sure to read “A Rapid Introduction to Molecular Biology” (at the same URL) first.

As before, submit both your working Python code, and a snapshot from the Rosalind web site demonstrating your successful completion of this assignment.

**Note: Please do not use BioPython for labs in this course.** The Python Biopython package offers a powerful set of tools for manipulating biological data, but these tools abstract away many details that we want you to learn by writing code that you understand completely. So, remember that Biopython exists; but for now, do not use it.

Lab Homework

1. Read next week’s lab exercise.

2. Visit [http://lightbot.com/hour-of-code.html](http://lightbot.com/hour-of-code.html) and complete Exercises 2 (procedures) and Exercise 3 (loops).
Further Exploration

1. In addition to IDLE, there are a variety of IDE’s (Integrated Development Environments) for Python, including Atom, Visual Studio, Sublime Text, Eric, Spyder, PyCharm, Eclipse with PyDev, and Thonny, to name a few. Experiment with one or two of these other development environments and see what you think.
Lab 3

Programming Exercise

Complete both of the following assignments:

1. Complete Rosalind Python Village ini4 (Conditions and Loops):
   http://rosalind.info/problems/ini4/

2. Complete the problem: “Transcribing DNA into RNA” at http://rosalind.info/problems/rna/. Be sure to read the “Click to Expand” section (at the same URL) first.

As before, submit both your working Python code, and a snapshot from the Rosalind web site demonstrating your successful completion of this assignment. **Remember that you should not use BioPython for labs in this course.**

Lab Homework

1. Read next week’s lab exercise.

2. Python is almost unique among modern programming languages in that “white space” (spaces, tabs, and carriage return/line feed) can have syntactic meaning. Thus, Python uses indentation to identify and separate blocks of code. Most other procedural programming languages use some mechanism built into the language syntax to separate blocks of code, e.g., C-family languages use curly braces (“{“ and “}”).
   a. Using online resources such as the Python wiki, explore the rational for Python to use indentation in this way.
   b. Pick one of the programs that you have already written, and translate it into another programming language (e.g., C, C++, or C#).
   c. (Optional) demonstrate that your translated program works correctly.

Further Exploration

1. “Functions,” also called “subroutines,” are used to divide programs into distinct chunks that can be reused, allowing us to better structure our code, and make it more readable and maintainable. Functions also allow us to separate and define “interfaces,” which represent how different parts of a program interact with one another. This facilitates programmers working together on a common body of code without interfering with one another. We will learn about functions in some detail later in the course. For a sneak preview, explore how functions work in Python, as follows.
   a. If you are more comfortable with the concept of functions, visit this tutorial: https://www.tutorialspoint.com/python/python_functions
   b. If you are less comfortable with the concept of functions, visit this tutorial: https://pythonprogramming.net/functions-python-3-basics-tutorial/
Lab 4

Programming Exercise

Complete both of the following assignments:

1. Complete Rosalind Python Village ini5 (Working with Files):
   
   http://rosalind.info/problems/ini5/

   Submit both your working Python code, and a snapshot from the Rosalind web site demonstrating your successful completion of this assignment.

2. Luhn’s Algorithm (This exercise was inspired by a similar exercise in the Harvard CS50 introductory computer science course (https://cs50.harvard.edu/college/2019/fall/)):

   Since the 1970’s, credit cards have become a ubiquitous method for paying for purchases and discharging debts. All such cards have a numerical sequence of digits that uniquely identify the card and its owner. In most cases, having a credit or debit card is as good as having cash. This exercise explores how credit/debit card numbers are generated and validated. Validation (testing for being a valid credit card number), is distinct from authentication and authorization, (ensuring that the user of the card is its rightful owner, and that they are allowed to make a specific purchase). We will not consider authentication or authorization in this exercise.

   Different card companies use different length numbers. For example, American Express numbers are fifteen digits in length, and Visa and MasterCard numbers are (usually) sixteen digits in length. Credit card companies also use unique identifying prefixes in their numbers. For example, American Express numbers usually start with 34 or 37; MasterCard numbers usually start with 51, 52, 53, 54, or 55, and Visa numbers usually start with a 4. For purposes of this assignment, assume that “usually,” as used here, means “always.”

   In addition to the identifying initial digits, all credit card numbers use a built-in “checksum,” which is a defined relationship between the digits of the credit card number. The checksum algorithm used by most credit card companies (and by several governments and many non-financial industries) is called the “Luhn Algorithm,” named for its inventor, an IBM scientist named Hans Peter Luhn. Luhn patented this algorithm (or rather, an invention that implemented this algorithm in a handheld mechanical device intended to be used by merchants) in 1960 (U.S. Patent No. 2,950,048, Computer for Verifying Numbers; filed in 1954), but the patent has since lapsed into the public domain. The Luhn algorithm is intended only to protect against accidental errors, i.e., mistyped numbers; it offers little protection from malicious cryptographic attack.

   Luhn’s algorithm for credit card number validity can be paraphrased as follows:

   1. Starting with the second to last digit (from the right) of the number, multiply every other digit by 2, recording each answer separately.
2. Sum all of the individual digits of all the products (not the products themselves) from Step 1.

3. To the sum obtained in Step 2, add all of the digits that were not multiplied by 2 in Step 1.

4. If the last digit of the resulting sum is zero, (i.e., the total modulo 10 is equal to 0), the number is valid.

Let’s try an example using a possible number that we created for this purpose:
4735672384163216

First, if valid, we know this is a Visa card because it starts with a “4”.

To make things easier, let’s underline every other digit, starting with the number’s second-to-last digit (the other digits are highlighted in yellow):

4935672384163216

Now multiply each of the underlined digits by 2, as shown here:

1\times2 ; 3\times2 ; 1\times2 ; 8\times2 ; 2\times2 ; 6\times2 ; 3\times2 ; 4\times2

That gives us:

2 ; 6 ; 2 ; 16 ; 4 ; 12 ; 6 ; 8

Now add those products’ digits (not the products themselves) together (the braces show how we have separated the digits of the two-digit numbers):

\[2 + 6 + 2 + [1 + 6] + 4 + [1 + 2] + 6 + 8 = 38\]

Now add this sum (38) to the sum of the digits that were not multiplied by 2 (the ones highlighted in yellow), starting from the right and working left:

\[38 + 6 + 2 + 6 + 4 + 3 + 7 + 5 + 9 = 80\]

The last digit in the resulting sum (80) is a zero, so this card number is valid (please do not try to use it)!

Your task:

a. (If you are less comfortable with this assignment) Write a Python program that asks for a credit card number (input as a string without hyphens or spaces, not as a number), and reports the card type (American Express, MasterCard, or Visa), and whether the card is valid. So, for the example above (4735672384163216), your program should output “4735672384163216--------VISA: VALID”, i.e., the number as a string, 8 hyphens, and “VISA: VALID.” If the entered number is anything other than a valid CC number, your program should output the number as a string, eight hyphens, and “INVALID NUMBER.” In addition to your working code, you should submit a plain text file with one answer per line, for each
of the following twenty CC numbers (which include valid and invalid credit card numbers of each type):

4735672384163216    5386622283419322
5125131221485655    4225298661212573
5451115051163662    5528322935717637
349232661988728     5420050412758076
4464766795523282    5345677634360280
5549984527236360    4581275409198311
341714386384157     4126427705118053
5365926823264290     377734275639344
5213376389751262    5141463348206102
375313685074727     5236758703360477

You may manually enter these numbers one at a time, or (better) read them from a file that you create.

b. **(If you are more comfortable with this assignment)** Write a Python program that reads a text file of candidate CC numbers, one number (as a string) per line, and outputs a file with the results described above for each CC number, one per line. The CC numbers the file (of about 700 numbers) may be of any type in any order. Test your program on the file `testCCNums.txt`, and submit the results with your working code.

**Lab Homework**

1. Read next week’s lab exercise.

2. Write a program that takes an invalid credit card number (but with a valid cc type identifier) and makes it valid.

   **Hint #1**: What is the minimum number of digits that need to be changed to make an invalid number (of the correct length and with the right prefix) valid?

   **Hint #2**: Do you think it might matter which digits are changed, depending upon card type?

**Further Exploration**

1. Are there number transpositions in otherwise valid CC numbers that Luhn’s algorithm will not detect?

2. (Optional) Write a program that generates ten valid numbers of each credit card type. DO NOT try to use these numbers to purchase anything. Prove that your numbers are valid by running them against your validation test program.
IWKS 2300 Section 2 Labs

Lab 5

Programming Exercise

Complete both of the following assignments:

1. Complete Rosalind Python Village ini6 (Dictionaries): http://rosalind.info/problems/inii6/. Be sure to read the “Click to Expand” section (at the same URL) first.

2. Complete the problem: “Complementing a Strand of DNA” at http://rosalind.info/problems/revc/. Be sure to read the “Click to Expand” section (at the same URL) first.

Turn in both your working Python code, and a snapshot from the Rosalind web site demonstrating your successful completion of these assignments. **Do not use BioPython for labs in this course.**

Lab Homework

1. Read next week’s lab exercise.

Further Exploration

Read about the history of the Human Genome Project. Start here: https://www.genome.gov/human-genome-project. Can you identify the strong connections between this project and the University of Colorado?
1. Legend has it that Julius Caesar used a simple code to encrypt his private correspondence. The “Caesar Cipher,” as it is known, is a type of substitution cipher in which each letter in the plaintext message is replaced by a letter some fixed number of positions from that letter in the alphabet. For example, with a left shift of three (the “key” purportedly used by Caesar), D would be replaced by G, E would become H, etc. This transformation can be represented by aligning two alphabets; the cipher alphabet is the plaintext alphabet rotated left or right by some number of positions. For instance, here is a Caesar cipher using a left shift (actually a rotation) of three letters, equivalent to a right shift of 23 (the shift parameter is used as the key):

<table>
<thead>
<tr>
<th>Plaintext Alphabet</th>
<th>Cipher Alphabet</th>
</tr>
</thead>
</table>

To encrypt a message, we replace each plain alphabet letter of the plaintext message with the cipher alphabet letter corresponding to the plain letter. Here is an example:

Plaintext:  \textbf{WE WILL ATTACK AT DAWN UNLESS ITS RAINING}

Ciphertext: \textbf{ZH ZLOO DWDFN DW GDZQ XQOHHV LWV UDLQLQJ}

To decrypt the message, we apply the opposite shift.

The effectiveness of the Caesar cipher was probably enhanced by the fact that illiteracy was high among most of Rome’s enemies. The “Vigenère Cipher” improves upon the Caesar cipher by applying it differently for each letter, using the letters of a “keyword” to determine the specific Caesar shift to use for each letter. The Vigenère cipher is named for Blaise de Vigenère, who wrote of it, but the cipher was actually invented by Giovan Battista Bellaso in 1553, who was building upon the work of others, most notably, Johannes Trithemius, who created the tabula recta (shown below), on which the Vigenère cipher is based.
The tabula recta has the alphabet written out 26 times in different rows, each alphabet shifted cyclically to the left one letter, relative to the previous alphabet. Each of these rows represents one of the 26 possible Caesar ciphers. The Vigenère cipher uses a “Key” to select which Caesar cipher to use for each letter of the plaintext message. The Key is used repeatedly if the message is longer than the key (which will usually be the case).

Each row in the tabula recta starts with a key letter. The rest of the row displays the letters A to Z (in shifted order). Although there are 26 key rows, a Vigenère cipher will use only as many Caesar cipher keys as there are unique letters in the Key. If the message is longer than the key, the letters of the Key will be reused in the same order, and each plaintext message letter will be encrypted using its corresponding Key row. For example, if the first letter of the plaintext message is W, and the first letter of the Key is C, row C and column W of the tabula recta yields a “Y.” So, to encrypt a message, we replace each plain alphabet letter of the Plaintext message with the Ciphertext letter found in the tabula recta at the location determined by the Key. Here is an example:

Plaintext: WE WILL ATTACK AT DAWN UNLESS ITS RAINING

KEY: COMPUTERSCIENCEISAWESOME

Ciphertext: YS IXFE EKLCKO NV HION QRDSEW KHE GUBRZFI
To decrypt the message, we simply do things in reverse.

Your task:

a. **If you are less comfortable with this assignment** Write a Python program that implements the Caesar cipher, allowing the user to specify the plaintext and the shift (for encryption), or the ciphertext and the shift (for decryption). You can have a single program that encrypts and decrypts, or you can separate these functions into two programs. Make sure you handle both upper case and lower case letters, but numbers and other characters (like spaces) should be passed through unmodified.

   **Hint 1:** The Python function `ord()` returns the ordinal ASCII number of a character, e.g., `ord('A') = 65`. Similarly, the Python function `chr()` is the inverse of `ord()`, e.g., `chr(122) = 'z'`.

   **Hint 2:** The Python character methods `isupper()` and `islower()` return `True` if the character is uppercase or lower case, respectively. Running this code:
   ```python
   lchar = 'a'
   uchar = 'G'
   if lchar.islower():
       print (lchar + ' is a lowercase letter')
   if uchar.isupper():
       print (uchar + ' is an uppercase letter')
   ```
   will produce the following output:
   
   a is a lowercase letter
   G is an uppercase letter

   **Hint 3:** Remember what the mod (%) operator does.

b. **If you are more comfortable with this assignment** Write a Python program that implements the Vigenère cipher, allowing the user to specify the plaintext and the key (for encryption), or the ciphertext and the key (for decryption) Make sure you handle both upper case and lower case letters, but numbers and other characters (like spaces) should be passed through unmodified. See Hints 1, 2, and 3 above.

   **Hint 4:** You do not have to build a *tabula recta* table to do the encryption/decryption.

   **Hint 5:** Get the Caesar cipher working first, then extend it to Vigenère.

**Lab Homework**

1. Read next week’s lab exercise.
Further Exploration (optional, and targeted to the mathematically inclined)

Public Key Encryption

The problem with encryption techniques that use a secret key is that the secrecy of the key is critical to the effectiveness of the cipher, yet the key must be shared for the cipher to work. “Public Key Encryption” techniques address this problem.

Public key encryption uses a pairs of keys: a public key that can be shared freely, and a private key that is known only to its owner. Here is a somewhat simplified description of how this works:

• Everyone has two keys, public (literally) and secret.
• These keys are inverses of each other (\( P \) is the public key, \( S \) is the Secret key and \( M \) is the plaintext message) i.e.,

\[
\{\{M\}\}^P \}^S = M = \{\{M\}\}^S \}^P
\]

• Knowing the public key does not help you guess the secret key.
• To send you a message, I simply encrypt it using your public key (\( C \) is the resulting ciphertext).

\[
\{M\}\}^{P(you)} = C
\]

• Only you can decrypt it, because only you know the inverse -- your secret key.

\[
\{C\}\}^{S(you)} = \{\{M\}\}^{P(you)} \}^{S(you)} = M
\]

Thus, any person can encrypt a message using the receiver's public key, but that encrypted message can only be decrypted with the receiver's private key. The effectiveness of public key encryption depends upon keeping the private key private. So how do we pick the keys? One of the best-known ways is to use the work of Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman, who in 1978 published the seminal paper *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, describing what came to be called the “RSA” algorithm. They had patented the algorithm the year before (US Patent 4,405,829). Simply put, the secret key is a pair of large prime numbers, and the public key is the even larger composite number that is the product of these primes. The actual RSA encryption/decryption functions are somewhat more complex, involving modular arithmetic, and a bunch of number theory derived from Euler's theorem (a generalization of Fermat's little theorem).

The security of RSA generally depends upon a strongly-held (but unproven) belief that it is computationally very hard to factor products of large prime numbers (the security of RSA also depends upon the difficulty associated with finding \( e \)th roots modulo a composite number \( N \) whose factors are not known, (the “RSA problem”)). This belief is based upon decades of work in number theory and the theory of computation. However, if we could find the prime factors of the public key, we could break the cipher, and at present, no one has proven that it is necessarily very hard to factor products of large
primes. In addition, Peter Shor published an algorithm in 1994 showing that a quantum computer could in principle perform the factorization needed to break RSA in polynomial time. (A program whose running time is a polynomial function of the size of its input is not considered computationally hard; for more information on this subject visit the Stanford Encyclopedia of Complexity at https://plato.stanford.edu/entries/computational-complexity/.)

Your (optional) task: download the original RSA paper here, and work through as much of it as you can.
Lab 7

Programming Exercise

Complete both of the following assignments:

1. Complete the problem: “Rabbits and Recurrence Relations” at http://rosalind.info/problems/fib/. Be sure to read the “Click to Expand” section (at the same URL) first.


Be sure to read the “Click to Expand” section (at the same URL) first.

Turn in both your working Python code, and a snapshot from the Rosalind web site demonstrating your successful completion of this assignment. Do not use BioPython for labs in this course.

Lab Homework

1. Read next week’s lab exercise.

Further Exploration

Lab 8

Programming Exercise

Complete the following assignment:

Data Visualization and Analysis

We will begin by exploring how to represent simple data sets visually. Python offers some excellent resources for this purpose, among them, Matplotlib and Plotly. Install these packages by typing the following commands into an OS (Windows or Mac) command window (you will likely need admin privileges to do this).

```
pip install matplotlib
pip install plotly
```

While we are at it, let’s install two other packages that we will need: the Alpha Vantage stock data access libraries, as well as the Pandas data analysis libraries. Do this as before, by typing the following commands into an OS (Windows or Mac) command window (with admin privileges).

```
pip install alpha_vantage
pip install pandas
```

We will first examine some of the basic “data visualization” capabilities of Matplotlib. Data visualization is a technique for creating visual representations of data that help us understand their meaning by helping to identify trends and patterns in data. Data visualization is used in most areas of science and engineering, and as we will see, finance.

The graph types that Matplotlib can draw (in two or three axes) include, among others, scatter plots, bar graphs, histograms, and pie charts. Consider a simple example in which we create two sets of coordinates, and some labels, before displaying the plot:

```python
## A simple program to demonstrate matplotlib
import matplotlib.pyplot as plt

## initialize x and y data sets
x = [0,1,2,3,4,5,6,7,8,9,10]
y = [1,2,4,8,16,32,64,128,256,512,1024]

## Configure plot parameters
plt.scatter(x,y,label='Powers of Two',color='r')
plt.xlabel('n')
plt.ylabel('2 to the n')
plt.title('Scatter Plot Example')
plt.legend()

## Display the plot
plt.show()
```

Try this example in IDLE. You should see an exponential curve that looks like this:
If we wanted to make the $y$-axis display in log scale, we could insert `pyplot.yscale('log')` before the call to `show()`; and just for fun, let’s label our data points, as follows:

```python
# A simple program to demonstrate matplotlib
import matplotlib.pyplot as pyplot

# initialize x and y data sets
x = [0,1,2,3,4,5,6,7,8,9,10]
y = [1,2,4,8,16,32,64,128,256,512,1024]

# Configure plot parameters
pyplot.scatter(x,y,label='Powers of Two',color='r')
pyplot.xlabel('n')
pyplot.ylabel('2 to the n')
pyplot.title('Scatter Plot Example')
pyplot.legend()

# Annotate each point with y data value
for i in range(0, len(x)):
    pyplot.annotate(xy=[x[i],y[i]], s=y[i])

# Make the y axis log scale
pyplot.yscale('log')

# Display the plot
pyplot.show()
```

(Try this code). Now we get a nice linear plot with labeled data:
Explore some of the other capabilities of Matplotlib by working through the short examples at https://www.tutorialkart.com/matplotlib-tutorial/.

Visualizing Stock Market Data

Shares of stock (called “securities”) generally represent fractional ownership in a corporation in proportion to the total number of shares. Like other kinds of property, shares of stock can be bought and sold. Stock “exchanges” were created to facilitate this process. A potential buyer of stock bids a specific price for the stock of a specific company, and a potential seller asks a specific price for the same stock. When the bid and ask prices match, a sale takes place, on a first-come, first-served basis, if there are multiple bidders at a given price. The purpose of a stock exchange is to facilitate the exchange of securities between buyers and sellers, thus providing a marketplace. Importantly, stock exchanges generally provide real-time trading information on the securities listed on their exchange, thus facilitating price discovery. In this part of the assignment, we will learn how to access and display this information. Props to the good folks at Alpha Vantage for making this possible (and easy).

You will need an “API Key” to access the Alpha Advantage feed. You can get one (it’s free) at: https://www.alphavantage.co/support/#api-key. Save this key; with it, you can make up to five API requests per minute, and up to 500 requests per day. Let’s make a few using the Alpha Vantage Python interface, starting with the intra-minute value for some stock. First, we need the “ticker name” for the stock of the company in which we are interested. A convenient place to find the ticker name online is at https://www.marketwatch.com/tools/quotes/lookup.asp. Enter “Apple Inc.,” and we see that ‘AAPL’ is the ticker name. Enter “Microsoft” and we get ‘MSFT’.

Create a new Python file in IDLE, and enter the following code (replacing “YOUR_API_KEY” with the Alpha Vantage API Key obtained previously):
from alpha_vantage.timeseries import TimeSeries
import matplotlib.pyplot as plt

ts = TimeSeries(key='YOUR_API_KEY', output_format='pandas')
data, meta_data = ts.get_intraday(symbol='MSFT', interval='1min', outputsize='full')
data['4. close'].plot()
plt.title('Intraday Times Series for the MSFT stock (1 min)')
## The following line flips the x axis to read left to right
plt.gca().invert_xaxis()
plt.show()

Here is what we get (do this):

Now let's plot the Bollinger Bands® for this stock. These bands plot two lines representing two standard deviations (positive and negative) away from a simple moving average (we will learn more about moving averages later) of the security's price, but can be adjusted to user preferences. Bollinger Bands® were developed and copyrighted by technical trader John Bollinger. Because standard deviation is a measure of volatility, when the market becomes more volatile, the bands widen; during less volatile periods, the bands contract. Here is the code, and the result of running this code (do this):

from alpha_vantage.techindicators import TechIndicators
import matplotlib.pyplot as plt

ti = TechIndicators(key='YOUR_API_KEY', output_format='pandas')
data, meta_data = ti.get_bbands(symbol='MSFT', interval='60min', time_period=60)
data.plot()
plt.title('BBbands indicator for MSFT stock (60 min)')
plt.gca().invert_xaxis()
plt.show()
We can also obtain real-time performance data for different market sectors. Here is the code, and the result of running this code (Alpha Vantage has made this interface unavailable unless one subscribes to their “High Usage” program. The code below will not run using the free API Key):

```python
from alpha_vantage.sectorperformance import SectorPerformances
import matplotlib.pyplot as plt

sp = SectorPerformances(key='YOUR_API_KEY', output_format='pandas')
data, meta_data = sp.get_sector()
data['Rank A: Real-Time Performance'].plot(kind='bar')
plt.title('Real Time Performance (%) per Sector')
plt.tight_layout()
plt.grid()
plt.show()
```
Finally, let’s look at the performance of Bitcoin, but in this case, we will look at the data we are retrieving a little more closely. We can obtain recent performance data of Bitcoin in a text file formatted in comma separated values (CSV) by browsing to the following URL (do this, and save to your desktop in a file named “BTC_CNY.csv”):

https://www.alphavantage.co/query?function=DIGITAL_CURRENCY_DAILY&symbol=BTC&market=CNY&apikey=demo&datatype=csv

We could have entered our API Key in this string, but “demo” works fine. Open this file in Excel (just double-click on the file) and you should see something like this (make the column width 12 to see the entire heading, and rename the file after saving to the default):

Now let’s work with this file in Python. We will first obtain the header row information using the following short program:

```python
from alpha_vantage.cryptocurrencies import CryptoCurrencies
import matplotlib.pyplot as plt
import pandas as pd

c = CryptoCurrencies(key='YOUR_API_KEY', output_format='pandas')
data, meta_data = c.get_digital_currency_daily(symbol='BTC', market='CNY')
with pd.option_context('display.max_rows', None, 'display.max_columns', None):
    print(data.describe(), file=open('file.txt', 'w'))
```

There are a couple of non-intuitive constructs here. First, the call to `CryptoCurrencies(...)` takes two parameters: your API Key, and the output format of the data (in this case, Pandas). The function used to obtain the data and meta data is `get_digital_currency_daily(...)` , which in turn takes arguments specifying the ticker symbol and market (Bitcoin and China Yuan Renminbi (CNY) in this case). The call to `option_context(...)` on the Pandas object (`pd`) removes the default limits on the number of rows and columns so our data will not be truncated. Finally the line
print(data.describe(), file=open('file.txt', 'w')) opens (or creates) the file “file.txt” for writing, and the embedded call to data.describe() causes a full description of the available data to be written to the file. Executing this program (do this) produces the following results (open the file.txt file in an editor to see this):

From these data, we see that the name of the data column containing the closing value in US Dollars is “4b. close (USD)”. Now we have the information we need to plot the data, as follows:

from alpha_vantage.cryptocurrencies import CryptoCurrencies
import matplotlib.pyplot as plt

cc = CryptoCurrencies(key='YOUR_API_KEY', output_format='pandas')
data, meta_data = cc.get_digital_currency_daily(symbol='BTC', market='CNY')
data['4b. close (USD)'].plot()
plt.tight_layout()
plt.title('Intraday Value for Bitcoin (BTC)')
plt.grid()
plt.gca().invert_xaxis()
plt.show()

Run this code and you should get something like the following:
You can see why investing in Bitcoin is not for the faint of heart. Now let’s extract data directly from the CSV file that we created. Most online data sources offer their data in CSV format, so this is a useful thing to know how to do. **Create a new program in IDLE and enter the following code** (which we have annotated):

```python
## Allow us to easily work with CSV files
import csv
## Allow us to parse date and time information
from datetime import datetime
## Include plotting libraries
from matplotlib import pyplot as plt
filename = 'BTC_CNY.csv'
with open(filename) as f:
    ## The file is read only
    reader = csv.reader(f)
    ## read the headers row
    header_row = next(reader)
    ## Get dates and prices from this file
    dates, prices = [], []
    for row in reader:
        ## Row 0 contains date information
        current_date = datetime.strptime(row[0], '%m/%d/%Y')
        dates.append(current_date)
        ## Row 8 contains closing price information
        price = float(row[8])
        prices.append(price)
    ## Plot prices vs. date
    plt.style.use('seaborn')
    fig, ax = plt.subplots()
    ax.plot(dates, prices, c='red')
    ## Format plot.
    plt.title("BTC Daily Closing Price (USD)", fontsize=24)
    plt.xlabel('', fontsize=16)
    fig.autofmt_xdate()
    plt.ylabel("US Dollars", fontsize=16)
```

![Graph showing the daily closing price of Bitcoin (USD) from 2013 to 2023](image.png)
plt.tick_params(axis='both', which='major', labelsize=16)
plt.show()
f.close()

Run this code, and you should see the following:

Finally, if we had just wanted to see the header information in the CSV file, we could have written the following:

```python
import csv
filename = 'BTC_CNY.csv'
with open(filename) as f:
    reader = csv.reader(f)
    ## read the headers row
    header_row = next(reader)
    for index, column_header in enumerate(header_row):
        print(index, column_header)

f.close()
```

Run this code, and you should see a list of the header rows:

```
0 timestamp
1 open (CNY)
2 high (CNY)
3 low (CNY)
4 close (CNY)
5 open (USD)
6 high (USD)
7 low (USD)
8 close (USD)
9 volume
10 market cap (USD)
```
This would have been another way to know that timestamps were in column 0 and that the USD close data were in column 8.

**Your task:**

Choose a company of interest to you. Analyze that company’s stock value over the last year. Create a trading algorithm that, if applied over the last year to this stock, would have resulted in a net gain. Write a Python program that demonstrates that your algorithm works (for this company’s stock).

**Hint:** See “Further Explanation” below.

**Lab Homework**

1. Read next week’s lab exercise.

**Further Exploration**

**Algorithmic Trading**

For most of its existence, the stock market relied upon human buyers and sellers, who made buy and sell decisions based upon their knowledge, experience, and perception. You have probably seen pictures or movies depicting frenzied individuals screaming buy and sell orders on the floor of the exchange. With the advent of computers, it became possible to write algorithms intended to mimic the best instincts of a successful stock trader. Algorithmic trading (sometimes called “algo-trading”) is the name given to these efforts. The basic idea behind algorithmic trading is for the trading program to examine current and historical data related to various listed stocks, and to place buy and sell orders based upon some form of predictive modeling, i.e., buy stocks that are expected to increase in value, and sell stocks that are expected to decrease in value. Most algorithmic trading is “high-frequency trading,” meaning that the trading program works within a pre-programmed set of parameters to place a large number of orders in rapid succession across multiple markets using multiple decision parameters. If you were thinking that this would be a bad place to have a software glitch, you would be correct.

Algorithmic trading programs use “backtesting” (applying the algorithm to historical stock-market performance to see if using the program would have been profitable) to tune their behavior. There are host of interesting techniques underlying algorithmic trading, resulting in a wide variety of approaches. A few examples include:

**Percentage of Volume (POV)** – These techniques send partial orders according to programmed strategies until the entire order is filled. Long-term investors like pension funds, mutual funds, and insurance companies use POV to purchase large quantities of stock a little bit at a time. This allows them to get lower prices, and reduces the risk that a large purchase will influence the stock price of the purchased security.

**Trend Following** – These programs follow trends in moving averages, price level movements, and related technical indicators. Trades (buys or sells) are initiated when stock technical indicators suggest
that the stock price is about to increase or decrease in value. An example of trend following is the use moving averages to predict future stock performance.

Trading Range Strategies – These programs attempt to identify a price range, and allow trades to be placed automatically when the price of a security breaks in and out of this range. “Mean reversion” is a trading range strategy based on the belief that the high and low prices of a stock are a temporary phenomenon, and that the stock price will over time tend to revert to its mean.

Trading Algorithms that Detect Other Trading Algorithms – These techniques attempt to “sniff out” the actions of other trading algorithms to identify opportunities for buys or sells that exploit what the other program is doing.

Your tasks:

a. Consider how you would implement and test a low-frequency algorithmic trading program, i.e., one in which a human places trades based upon program recommendations.

b. How might a trading program go about detecting another algorithmic agent acting in the market?
Lab 9

Programming Exercise

Complete both of the following assignments:

1. Complete the problem: “Mendel's First Law” at http://rosalind.info/problems/iprb/. Be sure to read the “Click to Expand” section (at the same URL) first.

2. Complete the problem: “Translating RNA into Protein” at http://rosalind.info/problems/prot/. Be sure to read the “Click to Expand” section (at the same URL) first.

Turn in both your working Python code, and a snapshot from the Rosalind web site demonstrating your successful completion of this assignment. Do not use BioPython for labs in this course.

Lab Homework

1. Read next week’s lab exercise.

Further Exploration

Lab 10

Programming Exercise

Complete the following assignment:

Filtering Data Using a Running Average

Real world data is rarely “clean.” Small perturbations caused by momentary ambient electrical, magnetic, mechanical, and other kinds of disturbances are often found in observed data. In order to see the underlying trends in these data as clearly as possible, we typically filter or smooth the raw data in some way. How we do this depends on the nature of the data. If changes in the data occur slowly over time, we can employ techniques such as a “moving average,” (also called “rolling average” or “running average”). The basic idea is to take an average of some number of data points, and use that average to represent the value of the graph at the current point in consideration. We must choose the width of the running average window (how many data points are averaged each time), and the location of the current point in the window. For example, we might choose to place the current point in the center of the window (we average an even number of points on either side of the current point with the current point, and replace the current point with this average value), or we might place the average at the front of the window (we average some number of points prior to the current point with the current point, and replace the current point with this average value). Let’s see how this works.

First, we need to generate some noisy data to filter. Run the following code (do this):

```python
import random
import matplotlib.pyplot as pyplot
x = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25]
y = []
value = 300
for i in range (0,26):
    y.append(value + random.randint(-20,+20))
    value -= 5
pyplot.scatter(x,y,label='Raw Data',color='r')
pyplot.xlabel('x')
pyplot.ylabel('Raw Data')
pyplot.title('Running Average Example: Raw Data')
pyplot.legend()
pyplot.show()
```

At this point in the semester, you should be able to figure out how this code works without further explanation. Take the time to do this. You should get a plot that looks something like this (but, since the data points are randomly generated, your plot will likely not look exactly like this):
Now we will compute a centered running average using a window of seven points, using a somewhat larger data set, as follows *(study this annotated code; then run it)*:

```python
### Example of a seven point running average
import random
import matplotlib.pyplot as pyplot
### Generate 100 x values
x = []
for i in range(0,100):
    x.append(i)
y = []
avg = []
value = 2000
### Generate 100 y values, generally decreasing, but with a fair amount (+/- 30) of noise
for j in range(0,len(x)):
    y.append(value + random.randint(-30,+30))
    value -= 2
avg = []
### Compute the running average, handling endpoint correctly
for j in range(0,len(x)):
    ### At the endpoints, we work with the points we have
    if (j==0):
    elif (j==1):
```

elif (j==2):

elif (j==len(x)-3):
    avg.append(((y[len(x)-6] + y[len(x)-5] + y[len(x)-4] + y[len(x)-3] + y[len(x)-2] + y[len(x)-1])/6))

elif (j==len(x)-2):
    avg.append(((y[len(x)-5] + y[len(x)-4] + y[len(x)-3] + y[len(x)-2] + y[len(x)-1])/5))

elif (j==len(x)-1):
    avg.append(((y[len(x)-4] + y[len(x)-3] + y[len(x)-2] + y[len(x)-1])/4))

else:
    # centered average

## Create a figure with subplots
fig = pyplot.figure()
ax = fig.add_subplot(111)  ## '111' = "1x1 grid, first subplot"
## Plot the raw data: in red, with filled circles as markers, dashed line style
ax.plot(x,y,c='r',marker='o',ls='--',label='Raw Data',fillstyle='full')
## Plot the averaged data: in green, filled arrows as markers, dotted line style
ax.plot(x,avg,c='g',marker='v',ls=':',label='Running Average (7pt)',fillstyle='full')
pyplot.xlabel('x')
pyplot.ylabel('Data Value')
pyplot.title('Running Average Example: (7 point)')
pyplot.legend()
pyplot.show()

This code should produce a plot something like the following:

![Running Average Example: (7 point)](image)

Note that the green line is much smoother than the noisy red one, but that the green line wanders toward noise values. Computing a running average is a form of linear filtering that enforces continuity. This is a good choice if our data are continuous, but what if our data are not continuous?
Median Filters

Sometimes data is inherently discontinuous. For example, consider the example of a robot following a black line on an otherwise white background. To detect the line, the robot might incorporate an IR reflective sensor that generates an infrared light, and then measures the intensity of the reflected light. White will reflect significantly more than black, so the reflection values can be used to detect a line. In the real world, sensors do not work perfectly, and ambient light sources (especially sunlight and light from incandescent lighting) may create “noise” in our sensor data stream.

Recognizing the edge of the line is critical to the success of the robot’s line-following algorithm. So how do we filter out the noise, but retain our ability to detect sharp discontinuities? Such “contrast preserving” filters are non-linear. Non-linear filter design is an advanced topic, but we can take advantage of a simple kind of contrast-preserving filter known as a “median filter.” Median filters work just like running average filters, but, instead of taking the average of a number of points, we take the median. How does this work? Consider the following simple example depicting a series of x value / y value pairs. The (red) values might represent our robot’s IR sensor data, where “8” is typical of white, and “6” is typical of black. In our example, the Y data are noisy. If we use a running average to filter this noise, we only see the line (black value < 6) for a portion of the time the sensor is detecting the line (depending on the amount of “hysteresis” we employ). This could cause our robot to act erratically. On the other hand, if we use a median filter (the blue line), the data is smoothed in a way that preserves the important edges of our data. Here is the code that generated this example (study, and then run this code):
import random
import statistics
import matplotlib.pyplot as pyplot

x = [1,2,3,4,5,6,7,8,9,10,11,12]
y = [8,9,8,6,4,6,7,6,6,8,9,8]

## compute average
avg = []
for j in range (0,len(x)):
    if (j==0):
        avg.append(statistics.mean([y[0], y[1] , y[2]]))
    elif (j==1):
    elif (j==len(x)-2):
        avg.append(statistics.mean([y[len(x)-4] , y[len(x)-3] , y[len(x)-2] , y[len(x)-1]]))
    elif (j==len(x)-1):
        avg.append(statistics.mean([y[len(x)-3] , y[len(x)-2] , y[len(x)-1]]))
    else:
        ## centered five point average

## compute median
med = []
for j in range (0,len(x)):
    if (j==0):
        med.append(statistics.median([y[0], y[1] , y[2]]))
    elif (j==1):
    elif (j==len(x)-2):
        med.append(statistics.median([y[len(x)-4] , y[len(x)-3] , y[len(x)-2] , y[len(x)-1]]))
    elif (j==len(x)-1):
        med.append(statistics.median([y[len(x)-3] , y[len(x)-2] , y[len(x)-1]]))
    else:
        ## centered five point median

fig=pyplot.figure()
ax=fig.add_subplot(111)  ## '111' = "1x1 grid, first subplot"
ax.plot(x,y,c='r',marker='o',ls='--',label='Raw Data',fillstyle='full')
ax.plot(x,avg,c='g',marker='v',ls=':',label='Running Average (5pt)',fillstyle='full')
ax.plot(x,med,c='b',marker='v',ls='-.',label='Median Filter (5pt)',fillstyle='full')
pyplot.xlabel('x')
pyplot.ylabel('Data Value')
pyplot.title('Average vs. Median Filter')
pyplot.legend()
pyplot.show()
Note the use of the “statistics.mean” and “statistics.median” functions from the Python statistic package in this example, which saves us a bit of coding. Median filters also detect gradual changes, which make them great filters for real-world data. Here are two data examples with both gradual and sharp changes in the data stream. As you can see, averaging rolls off all of the sharp edges, while the median filter preserves them.

Your task: Write a Python, C++, or C# program that demonstrates these techniques on either a data set of interest to you, or one that you create for this purpose.

Lab Homework

1. Read next week’s lab exercise.
Further Exploration (optional, targeted to the mathematically inclined)

Edge Detection using a Sobel Kernel Filter

Another application of non-linear filtering is image edge detection. An edge is where change occurs quickly. Recall from calculus that the first derivative of a continuous function is the rate of change, and the second derivative is the rate of the rate of change. The first derivative is at maximum, and the second derivative is zero, at the points of greatest change. However, edges in images (and images themselves) are discrete, not continuous. Real edges also exhibit noise (jagged edges at high resolution). To address these issues, we often use a technique called “convolution”, and a particular subset of convolution that uses kernel filters. There are a variety of kernel filters, but one example is the “Sobel Operator.” Let’s explore how these work, starting with a little mathematical background.

For a discrete one-dimensional function \( f(i) \), the first derivative can be approximated by the equation:

\[
\frac{df(i)}{d(i)} = f(i + 1) - f(i)
\]

Images are two two-dimensional objects, so what is the two-dimensional equivalent of a derivative? Just as the derivative represents the rate of change of a one-dimensional function, the “gradient” is a multi-variable generalization of the derivative. The gradient (represented by \( \nabla \) (the nabla symbol; pronounced “del”) of a multi-variable function is a vector whose direction is the direction of the most rapid change of the function, and whose magnitude is the rate of increase/decrease in that direction. (We are ignoring some details here related to the fact that the function has to have certain properties in order to be able to compute the gradient – you will learn these details when you take vector calculus.)

For a two dimensional image, the gradient vector has x and y components that can be separately derived by computing the partial derivative with respect to each variable, i.e.:

\[
\nabla f = \left[ \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right]
\]

Here are a few simple examples:

\[
\nabla f = \left[ \frac{\partial f}{\partial x}, 0 \right]
\]

\[
\nabla f = \left[ 0, \frac{\partial f}{\partial y} \right]
\]

The gradient direction is given by:

\[
\theta = \tan^{-1}\left( \frac{\partial f / \partial y}{\partial f / \partial x} \right)
\]
The edge strength is given by the gradient magnitude:

$$\| \nabla f \| = \sqrt{\left( \frac{\partial f}{\partial x} \right)^2 + \left( \frac{\partial f}{\partial y} \right)^2}$$

We can compute gradients using a technique known as “convolution.” Convolution, simply put, is an operation on two functions that produces a third function that represents a “blending” of the original two functions. (More formally, convolution involves integrating the product of the two functions after one is reversed and shifted. Again, stay tuned for more details when you take vector calculus).

Gradient edge detection is a widely used technique in which an image is convolved with two “kernels,” one estimating the gradient in the x-direction, $G_x$, the other the gradient in the y-direction, $G_y$. The gradient is the two-dimensional equivalent of the first derivative and is defined as the gradient vector:

$$\nabla f (x, y) = G = \begin{bmatrix} G_x \\ G_y \end{bmatrix}$$

The magnitude of $G$ is:

$$| G | = \sqrt{G_x^2 + G_y^2} \approx |G_x| + |G_y|$$

Calculating this formula is equivalent to convolving the function with [-1 1]. Similarly the second derivative can be estimated by convolving $f(i)$ with [1 -2 1]. When working with image data, we often use a “kernel” (sometimes called a kernel filter, convolution matrix, or mask) to perform convolution. A kernel is a small matrix used as one of the convolution functions (the other function is the image data itself). A kernel filter works by applying a kernel matrix to every pixel in the image. The kernel contains multiplication factors to be applied to the pixel and its neighbors. Once all the values have been multiplied, the pixel is replaced with the sum of the products. By choosing different kernels, different types of filtering can be applied. The convolution matrix filter applies the kernel to the matrix. The kernel used depends upon the effect desired. For each pixel of the image, the filter multiplies the value of that pixel and values of the eight surrounding pixels by the kernel corresponding value. Then it adds the results, and the initial pixel is set to the resulting value.

A simple example illustrates this idea, where the numbers in the boxes on the left represent image data, and the numbers in the boxes in the middle represent the convolution filter kernel:
When the indicated convolution filter is applied, the red squared pixel become 42, as follows:

\[(40*0)+(42*1)+(46*0) + (46*0)+(50*0)+(55*0) + (52*0)+(56*0)+(58*0) = 42.\]

When applying the filter, this step is performed for every pixel of the image.

So how does all this math apply to edge detection? The “Sobel operator,” one of more commonly used edge detectors, performs a 2-D spatial gradient measurement on an image. In practice, the Sobel operator consists of a pair of 3×3 convolution kernels, as shown below. One kernel is simply the other rotated by 90°.

\[
\begin{bmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1 \\
\end{bmatrix}
\quad
\begin{bmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
1 & 2 & 1 \\
\end{bmatrix}
\]

The Sobel operator places an emphasis on pixels that are closer to the center of the mask. Often, the absolute magnitude is the only output needed --- the two components of the gradient are conveniently computed and added in a single pass over the input image using the pseudo-convolution operator shown below.

\[
|G| = P_3 = \left| (P_1 + 2*P_2 + P_3) - (P_7 + 2*P_8 + P_9) \right| + \left| (P_3 + 2*P_6 + P_9) - (P_1 + 2*P_4 + P_7) \right|
\]
The good news is that Python packages exist that perform all of this math for us. Using Sobel edge detection, we can extract the middle edge detected image from the original image on the left (the famous Rodin sculpture at Columbia: “The Thinker”); edges are shown inverted on the right (image from https://blogs.cul.columbia.edu/outdoorsculpture/).

Here is the Python code that produced these images (note that two packages we have not used before are being used here, scipy, and imageio (numpy is part of the standard Python installation); if you want to run this code (and you should, if you have read this far), you will need to install these Python packages using pip), and you will need the art:

```python
import numpy
import scipy
import imageio
from scipy import ndimage

im = imageio.imread('art3.jpg')
im = im.astype('int32')
dx = ndimage.sobel(im, 0)  # horizontal derivative
dy = ndimage.sobel(im, 1)  # vertical derivative
mag = numpy.hypot(dx, dy)  # magnitude
mag *= 255.0 / numpy.max(mag)  # normalize
imageio.imsave('sobel.jpg', mag)
invfile = imageio.imread('sobel.jpg')
inv = numpy.invert(invfile)
imageio.imsave('sobel_inv.jpg', inv)
```
Lab 11

Complete any two of the following problems (These five problems are a little harder than the Rosalind problems we have seen so far, so you might want to look at all of them before you choose):

1. “Finding a Motif in DNA” at http://rosalind.info/problems/subs/. Be sure to read the “Click to Expand” section (at the same URL) first.
2. “Computing GC Content” at http://rosalind.info/problems/gc/. Be sure to read the “Click to Expand” section (at the same URL) first.
3. “Mortal Fibonacci Rabbits” at http://rosalind.info/problems/fibd/. Be sure to read the “Click to Expand” section (at the same URL) first.
4. “Finding a Most Likely Common Ancestor” at http://rosalind.info/problems/cons/. Be sure to read the “Click to Expand” section (at the same URL) first.
5. “Calculating Expected Offspring” at http://rosalind.info/problems/iev/. Be sure to read the “Click to Expand” section (at the same URL) first.

Turn in both your working Python code, and a snapshot from the Rosalind web site demonstrating your successful completion of this assignment. Remember that you may not use BioPython for labs in this course.

Lab Homework

1. Read next week’s lab exercise.

Further Exploration

Learn more about RNA. Start here: https://www.thoughtco.com/types-of-rna-1224523.
Lab 12

Programming Exercise

Now that we have some of the basics of data visualization, let’s look at some data that has nothing to do with the stock market. In this exercise, we will explore and visualize certain data related to climate and earthquakes. Complete both Part (a) and Part (b).

1. For this part of the exercise, we will examine historical climate data in CSV format from the NOAA National Centers for Environmental Information at [https://www.ncdc.noaa.gov/cdo-web/](https://www.ncdc.noaa.gov/cdo-web/). We obtained data from this web site by submitting an electronic request that detailed the nature of the data that we sought to obtain. These requests, if valid, are responded to immediately. You should explore the process of obtaining data from this site by using the Search Tool on the landing URL, but we have already downloaded annual weather-related data for Longmont, Colorado from 1895 through 2018. These data can be found in the file `Longmont_Data.csv`. We chose Longmont because it was nearby, and its weather station has been operating longer than most stations in Colorado.

As before, we can get the header information from this file with this simple Python program:

```python
import csv
filename = 'Longmont_Data.csv'
with open(filename) as f:
    reader = csv.reader(f)
    ## read the headers row
    header_row = next(reader)
    for index, column_header in enumerate(header_row):
        print(index, column_header)
f.close()
```

Run this program **(do this)** and you will obtain a list that starts with:

```
0 STATION
1 NAME
2 DATE
... 
```

which we can probably figure out, but then our list continues with entries of the form “CDSD, CLDD, DP01, DP10, ..., HDSD, HTDD, PRCP, SNOW, TAVG, TMAX, TMIN.” What to these labels mean? Fortunately, the nice folks at NOAA have provided a key, as follows:

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDSD</td>
<td>Cooling Degree Days Season to Date</td>
</tr>
<tr>
<td>CLDD</td>
<td>Cooling Degree Days</td>
</tr>
<tr>
<td>DP01</td>
<td>Number of days with greater than or equal to 0.1 inch of precipitation</td>
</tr>
<tr>
<td>DP10</td>
<td>Number of days with greater than or equal to 1.0 inch of precipitation</td>
</tr>
<tr>
<td>DT00</td>
<td>Number days with minimum temperature less than or equal to 0.0 F</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>DSND</td>
<td>Number days with snow depth &gt; 1 inch (25.4mm) for the period.</td>
</tr>
<tr>
<td>DSNW</td>
<td>Number days with snow depth &gt; 1 inch.</td>
</tr>
<tr>
<td>DT32</td>
<td>Number days with minimum temperature less than or equal to 32.0 F</td>
</tr>
<tr>
<td>DX32</td>
<td>Number days with maximum temperature &lt; 32 F</td>
</tr>
<tr>
<td>DX70</td>
<td>Number days with maximum temperature &gt; 70 F (21.1C)</td>
</tr>
<tr>
<td>DX90</td>
<td>Number days with maximum temperature &gt; 90 F (32.2C)</td>
</tr>
<tr>
<td>EMNT</td>
<td>Extreme minimum temperature for the period.</td>
</tr>
<tr>
<td>EMSD</td>
<td>Extreme maximum snow depth for the period.</td>
</tr>
<tr>
<td>EMSN</td>
<td>Extreme maximum snowfall for the period.</td>
</tr>
<tr>
<td>EMXP</td>
<td>Extreme maximum precipitation for the period.</td>
</tr>
<tr>
<td>EMXT</td>
<td>Extreme maximum temperature for the period.</td>
</tr>
<tr>
<td>FZF0</td>
<td>First freeze &lt;= to 32F/0C of the year.</td>
</tr>
<tr>
<td>FZF1</td>
<td>First freeze &lt;= 28F/-2.2C of the year</td>
</tr>
<tr>
<td>FZF2</td>
<td>First freeze &lt;= 24F/-4.4C of the year</td>
</tr>
<tr>
<td>FZF3</td>
<td>First freeze &lt;= 20F/-6.7C of the year</td>
</tr>
<tr>
<td>FZF4</td>
<td>First freeze &lt;= 16F/-8.9C of the year</td>
</tr>
<tr>
<td>FZF5</td>
<td>Last freeze &lt;= 32F/0C of the year</td>
</tr>
<tr>
<td>FZF6</td>
<td>Last freeze &lt;= 28F/-2.2C of the year</td>
</tr>
<tr>
<td>FZF7</td>
<td>Last freeze &lt;= 24F/-4.4C of the year</td>
</tr>
<tr>
<td>FZF8</td>
<td>Last freeze &lt;= 20F/-6.7C of the year</td>
</tr>
<tr>
<td>FZF9</td>
<td>Last freeze &lt;= 16F/-8.9C of the year</td>
</tr>
<tr>
<td>HDSD</td>
<td>Heating Degree Days Season to Date</td>
</tr>
<tr>
<td>HTDD</td>
<td>Heating degree days</td>
</tr>
<tr>
<td>PRCP</td>
<td>Precipitation</td>
</tr>
<tr>
<td>SNOW</td>
<td>Snowfall</td>
</tr>
<tr>
<td>TAVG</td>
<td>Average Temperature</td>
</tr>
<tr>
<td>TMAX</td>
<td>Maximum temperature</td>
</tr>
<tr>
<td>TMIN</td>
<td>Minimum temperature</td>
</tr>
</tbody>
</table>

More detailed explanations of these codes can be found at: [https://www1.ncdc.noaa.gov/pub/data/cdo/documentation/gsom-gsoy_documentation.pdf](https://www1.ncdc.noaa.gov/pub/data/cdo/documentation/gsom-gsoy_documentation.pdf).

So, what data would well represent changes over time? Let’s start with the ones highlighted in yellow above (column numbers from the header row information are shown below in parentheses):

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DX32</td>
<td>Number days with maximum temperature &lt; 32 F.</td>
<td>(Column 11)</td>
</tr>
<tr>
<td>DX90</td>
<td>Number days with maximum temperature &gt; 90 F (32.2C)</td>
<td>(Column 13)</td>
</tr>
<tr>
<td>EMNT</td>
<td>Extreme minimum temperature for the period.</td>
<td>(Column 14)</td>
</tr>
<tr>
<td>EMXT</td>
<td>Extreme maximum temperature for the period.</td>
<td>(Column 18)</td>
</tr>
<tr>
<td>TAVG</td>
<td>Average Temperature.</td>
<td>(Column 33)</td>
</tr>
</tbody>
</table>

We can display these data using plotting techniques with which we are already familiar, but with a few embellishments. Consider the following code:
import csv
from numpy.polynomial.polynomial import polyfit
import pandas as pd
import matplotlib.pyplot as pyplot

filename = 'Longmont_Data.csv'
year = []
tavg = []
with open(filename) as f:
    reader = csv.reader(f)
    ## read the header row
    header_row = next(reader)
    ## now the other rows (data in some rows for some years may be empty
    for row in reader:
        try:
            yr = float(row[2])  ## Date in Row 2 (casting as float for polyfit)
            tmp = float(row[33])  ## TAVG in Row 33
        except ValueError:
            print(f"Missing temp for year {row[2]}")
        else:
            year.append(yr)
            tavg.append(tmp)

    ## compute running (rolling) average
    df = pd.DataFrame(tavg)
    ravg = df.rolling(window=5).mean()

    ## Compute line of best fit
    b, m = polyfit(year, tavg, 1)
    ## y = mx + b
    poly = []
    for i in range(0, len(year)):
        poly.append((m * year[i]) + b)

    fig = pyplot.figure()
    ax = fig.add_subplot(111)  ## '111' = "1x1 grid, first subplot"
    ax.scatter(year, tavg, c='r', marker='o', ls='--', label='TAVG')
    ax.plot(year, ravg, c='g', marker='v', ls=':', label='Running Average (5pt)', fillstyle='full')
    ax.plot(year, poly, c='b', ls=':', label='Line of Best Fit')
    pyplot.xlabel('Year')
    pyplot.ylabel('Annual Average Temperature')
    pyplot.title('Longmont Annual Average Temperature, 1895-2018')
    pyplot.legend()
    pyplot.show()
    f.close()
There are three new constructs in this code. First, instead of manually computing the running average ourselves (which was kind of a pain), we use a feature of the Python Pandas package to do this for us. The call “df.rolling(window=5).mean()” computes a five point running average. How cool is that? What do you think would happen if we replaced “mean” with “median” in this call?

The second new construct is the call to “polyfit,” which requires a bit of explanation. Scientists frequently encounter datasets from which they seek to derive meaning. One approach that is effective for data expressed in (x, y) value pairs is a technique known as “curve fitting.” The process of curve fitting seeks to determine a function that best represents the observed data, i.e., has the best fit to a series of observed data points. When looking for trends in certain kinds of data, scientists often look for a “line of best fit,” meaning that they want to determine the slope (m) and the “y” intercept (b) for the line equation:

\[ y = mx + b \]

Curve fitting can be much more complex, involving functions for higher order polynomials. We will stick with lines, using a technique called the “Method of Least Squares.” The mathematics for actually computing the fit using this technique are complex, involving the simultaneous solution to a system of partial differential equations. You can find a good explanation of the math behind curve fitting using the Method of Least Squares at: [https://www.engr.uidaho.edu/thompson/courses/ME330/lecture/least_squares.html](https://www.engr.uidaho.edu/thompson/courses/ME330/lecture/least_squares.html). For now, we can celebrate the fact that the Python Numpy package can perform this computation for us. You should also be aware that the polyfit function exhibits some mathematical instability for higher order polynomials, so make sure that the argument after the data in our call to polyfit is a “1,” as it is in the above code.

The third new construct is this code block:

```python
try:
    yr = float(row[2])  ## Date in Row 2 (casting as float for polyfit)
    tmp = float(row[33]) ## TAVG in Row 33
except ValueError:
    print(f"Missing temp for year {row[2]}")
else:
    year.append(yr)
    tavg.append(tmp)
```

Often when we write a program, we know that a particular kind of error may occur. Rather than letting our program simply halt and catch fire, we handle the error explicitly in the program. Many languages, including Python, provide a way to do this using the “try ... except” model. The idea is that the program tries to execute the code within the “try” block (in this case reading data from row 2 and row 33), and if something goes wrong (e.g., trying to make an empty cell into a floating point number, called a “ValueError”), we do something (in this case printing the year for which there is no corresponding data). On the other hand, if our “try” code executes successfully, we proceed normally (in this case appending the values that we know are...
valid). Python allows the programmer to catch a wide variety of errors (you can find the entire list of errors and warnings that are “catchable” at: https://docs.python.org/3/library/exceptions.html). If we just wanted to catch all errors, we can leave the error field blank, as in “except:”. This model for error checking, and handling exceptions, in the program itself is extremely useful, and its use is the mark of a serious programmer.

If you run the code above on the Longmont dataset (do this), you should see something like:

If we modify the code to examine the other selected variables of interest (do this), we obtain the following results:
**Your task for Part 1:** Pick a station anywhere in the United States of interest to you and download the historical climate data from NOAA for that station. Apply the analysis techniques that we have just examined to these data, choosing those data you consider interesting, and report your results.

2. CSV formatted data is convenient to manipulate, but it only offers a very simple structuring of the data: x, y, and maybe z. Alternative formats have been developed that allow data to be organized in a way that conveys structural context and meaning, for example, XML (Extended Markup Language) and JSON (JavaScript Object Notation).

JSON is a human-readable, language-independent file format that uses attribute-value pairs and arrays to represent complex data. JSON was created for use with JavaScript, but it has been widely adopted by other languages. JSON was first standardized in 2013 in [RFC 7158](https://tools.ietf.org/html/rfc7158) (which addressed certain security and interoperability issues) and [ECMA-404](https://www.ecma-international.org/publications/standards/Ecma-404.htm) (which defined JSON syntax). The most recent JSON standard is [RFC 8259](https://tools.ietf.org/html/rfc8259), published in 2017. GeoJSON is a standardized variant of JSON used to represent and structure geo-located data. The specification for GeoJSON is found in [IETF RFC 7946](https://tools.ietf.org/html/rfc7946), published in August 2016.

In this exercise, we will examine earthquake data in JSON format obtained from the USGS at [https://earthquake.usgs.gov/earthquakes/feed/](https://earthquake.usgs.gov/earthquakes/feed/), and we will map these data using the geo-location-mapping capabilities of the Plotly Python package. This exercise was inspired by a similar example found in Eric Matthes’ excellent book: *Python Crash Course, 2nd Edition: A Hands-On, Project-Based Introduction to Programming* (available at Amazon for around $25.00: [https://www.amazon.com/Python-Crash-Course-2nd-Edition/dp/1934356842](https://www.amazon.com/Python-Crash-Course-2nd-Edition/dp/1934356842)).

For this exercise, we have downloaded the file: “[4.5_month.geojson.json](https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php),” which is a compilation of all earthquakes worldwide in the last month with a magnitude greater than 4.5 (downloaded on August 4, 2019). If you open this file in a text editor, the first few lines (the file is 508 lines long) indicate its (somewhat) human readable format:
The lines of this file are far too long to print on this page, but you can see that these data are structured. We can get an easier-to-read version of these data by writing a little Python, as follows (do this):

```python
import json
filename = '4.5_month.geojson.json'
with open(filename) as f:
    ## read the JSON data into a Python dictionary
earthquake_data = json.load(f)

## create a readable JSON file
earthquake_data_readable = 'eqke_Mag4_5_30day.json'
with open(earthquake_data_readable, 'w') as f:
    json.dump(earthquake_data, f, indent=4)
```

This code reads the original file into a Python dictionary, then writes these data back out into a formatted file with indentation. If we now open the file that we just created, we have a much more readable (but longer) file that begins:

```
{
"type": "FeatureCollection",
"metadata": {
    "generator": "1564954452000",
    "url": "https://earthquake.usgs.gov/earthquakes/feud/v1.0/summary/4.5_month.geojson",
    "title": "USGS Magnitude 4.5+ Earthquakes, Past Month",
    "status": 200,
    "api": "1.8.1",
    "count": 508
},
"features": [

```
This preamble is known as the “metadata” for the rest of the file. The metadata tells us information about how the data in the file was created, and at a high level, how these data are structured. In this case, we can see that the file is a collection of “features,” and that there are 508 of them.

If we look at the first feature, we find a section with a number of “properties,” followed by sections for “geometry” and “id.”:

```
{
"type": "Feature",
"properties": {
    "mag": 5.3,
    "place": "30km N of Do Gonbadan, Iran",
    "time": "1564949281181",
    "updated": "15649544528622",
    "tz": 210,
    "url": "https://earthquake.usgs.gov/earthquakes/eventpage/us6000501k",
    "detail": "https://earthquake.usgs.gov/earthquakes/feud/v1.0/detail/us6000501k.geojson",
    "felt": 17,
    "cdi": 7.4,
```

...
The key data that we are interested in for each earthquake are its location and magnitude. We might also want a descriptive title if we wanted to provide basic information on some form of interactive map (which we do). The location appears in the “geometry” section, under “coordinates,” and the magnitude and description are found in the “properties” section. The coordinates provide traditional latitude and longitude, but with a twist; longitude appears first, then latitude. This is standard for the “GeoJSON” format in which the USGS provides these files, but be careful; not all geo-spatial datasets follow this convention.

Now we will extract these data and present them on a world map. Consider the following program:

```python
import json
from plotly.graph_objs import Scattergeo, Layout
from plotly import offline

filename = '4.5_month.geojson.json'
with open(filename) as f:
    earthquake_data = json.load(f)  ## read data into Python dictionary
eqke_list = earthquake_data['features']  ## Extract into list of "features"
## Extract the magnitude of each feature
magnitudes = []
for eqke in eqke_list:
    mag = eqke ['properties']['mag']
    magnitudes.append(mag)
## Extract latitude and longitude of each feature
longitudes = []
latitudes = []
for eqke in eqke_list:
    long = eqke ['geometry']['coordinates'][0]
    lat = eqke ['geometry']['coordinates'][1]
    longitudes.append(long)
    latitudes.append(lat)
## Extract earthquake description of each feature
descriptions = []
for eqke in eqke_list:
    title = eqke ['properties']['title']
    descriptions.append(title)
```
## Create a world map depicting these data

## Define the scattergeo data presentation format

data = [{
    'type': 'scattergeo',
    'lon': longitudes,
    'lat': latitudes,
    'text': descriptions,
    'marker': {
        'size': [((mag**2)/2) for mag in magnitudes],
        'color': magnitudes,
        'colorscale': 'Portland',
        'reversescale': False,
        'colorbar': {'title': 'Magnitude'}
    }
}]

## Plot the data in an interactive map

my_layout = Layout(title='Earthquakes in Last 30 Days > Magnitude 4.5')
fig = {'data': data, 'layout': my_layout}
offline.plot(fig, filename='global_4_5_earthquakes.html')
f.close()

Let’s study this code a little bit at a time. In addition to the using the JSON data manipulation tools of the Python json package, we will use three capabilities of the Plotly interactive graphing package for Python. Plotly offers a rich array of capabilities for displaying data of all kinds. Here, we will only scratch the surface of these capabilities. You should spend some time examining the descriptions of what Plotly can do at: [https://plot.ly/python/](https://plot.ly/python/).

Scattergeo, as its name implies, is a tool for creating a scatter plot of geo-located data. Layout allows us to specify attributes that apply to the entire plot, like the title, axis details, and format. Offline is tool for creating standalone html plots, which can be interactive.

The next block of code after the import section reads the original JSON file, calls json.load() to parse this file into a Python dictionary, and creates a Python list of structured data for each feature (earthquake) in the file.

The next three blocks of code extract data from this dictionary. These three blocks were separated to make the code easier to read, but could easily be combined into a single for loop. The first block creates an empty list of magnitudes, and then populates this list by extracting the magnitude from each element of our structured list of earthquakes. For example, the term

    eqke ['properties'] ['mag']

navigates the feature structure to “properties,” and then to the “mag” element within properties. Longitude, latitude, and title information are similarly extracted.
The next block of code:

```python
data = [{
'type': 'scattergeo',
'lon': longitudes,
'lat': latitudes,
'text': descriptions,
'marker': {
'size': [((mag**2)/2) for mag in magnitudes],
'color': magnitudes,
'colorscale': 'Portland',
'reversescale': False,
'colorbar': {'title': 'Magnitude'}
}
}]
```

is used to define what data we want to plot, and how we want these data to be displayed. Most of the interesting information here is in the “marker” definition. For each earthquake, this code defines the marker that will be used to represent that earthquake on a map. The size of the marker is made to be proportional to the magnitude of the earthquake, in this case, the magnitude is squared, and the result is divided by two. This particular factor was used because we observed that the majority of magnitudes were similar in size, and squaring the magnitude allowed us to discriminate magnitudes more readily.

Our code uses the “Portland” color scale, which ranges from blues to reds. There are many other possible color scales including 'Blackbody', 'Bluered', 'Blues', 'Earth', 'Electric', 'Greens', 'Greys', 'Hot', 'Jet', 'Picnic', 'Portland', 'Rainbow', 'RdBu', 'Reds', 'Viridis', 'YlGnBu', and 'YlOrRd'. These scales can be reversed by setting ‘reversescale’ to True.

The last four lines of our code:

```python
my_layout = Layout(title='Earthquakes in Last 30 Days > Magnitude 4.5')
fig = {'data': data, 'layout': my_layout}
offline.plot(fig, filename='global_4_5_earthquakes.html')
f.close()
```

is the first call `Layout` to define a title (this is among the simplest things we can do with `Layout`; see https://plot.ly/python/reference/ for more information about Plotly and `Layout`). The code then identifies the data that will be plotted (that we have previously defined), and calls `offline.plot` to create an interactive html document containing our plot. More information about `offline` (and other plotting techniques) can be found at https://plot.ly/python/renderers/.

Now that we understand this code, let’s run it (do this). You should, after a bit of computation time, get an interactive html map of the earthquake data that supports pan, scroll, zoom, and rollover (experiment with this). Here is a non-interactive snapshot of the result of running the above code:
Your task for Part 2 (choose one of the following):

1. More flexible earthquake analysis
   a. Go to the USGS realtime earthquake data feed and download JSON data for all earthquakes globally in the last 30 days.
   b. Write a Python, C++, or C# program that:
      i. Prompts the user for a minimum magnitude to consider.
      ii. Ensures that the data for magnitude, latitude, and longitude are valid, and handles the case where they are not.
      iii. Using the techniques that we have just learned, creates an interactive map that displays these data with appropriate title and labels.
   c. Report your results.

2. Choose your own adventure
   a. Find a large GeoJSON data feed of interest and download it.
   b. Write a Python, C++, or C# program that:
      i. Prompts the user to select at least one feature for analysis.
      ii. Ensures that the selected data are valid, and handles cases where they are not.
      iii. Using the techniques that we have just learned, creates an interactive map that displays these data with appropriate title and labels.
   c. Report your results.
Lab Homework

1. Suppose you were a climate science denier and wanted to use the data we have examined in this week’s exercise to make the case that earth’s climate has not undergone significant changes in the last 100 years. What data would you present and how would you present these data?

2. On the other hand, suppose you wanted to make the strongest possible case that that earth’s climate has undergone significant changes in the last 100 years. What data would you present and how would you present these data?

3. Explore the structure and purpose of XML, another example of a (sort-of) human-readable language to structure data. Start here: https://www.w3.org/TR/REC-xml/.

Further Exploration (optional)

Protocol Buffers

The fact that JSON is a human readable format makes it inherently inefficient, since humans need a lot of contextual information in order to interpret data. JSON and other text-based schema, e.g., XML (Extended Markup Language), all share this limitation. If we did not care about human readability, we could serialize data far more efficiently. One such technique is Google’s Protocol Buffers, or “protobufs.” Google’s developer web site contains the following definition for protocol buffers:

“Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.” https://developers.google.com/protocol-buffers/

Protocol buffers require that the structure of the information to be serialized be defined in a “.proto” file. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. Each message type has one or more uniquely numbered fields, and each field has a name and a value type, where value types can be numbers (integer or floating-point), booleans, strings, raw bytes, or other protocol buffer message types, allowing users to structure data hierarchically. Fields can be optional or required. Much more information about writing .proto files can be found in the Protocol Buffer Language Guide: https://developers.google.com/protocol-buffers/docs/proto.

Once a protocol buffer message format is defined, developers run a language-specific protocol buffer compiler on the .proto file to generate data access functions. These functions provide simple mechanisms for accessing message data in a structured way, and for serializing and deserializing the messages themselves. Many producers of content in protocol buffer form offer pre-compiled access methods for several languages (including Python, C++, and C#) in order to simplify user access.
Google claims that protocol buffers are superior to XML for serializing structured data because they are “simpler,” “3 to 10 times smaller,” “20 to 100 times faster,” “less ambiguous,” and “easier to use programatically.” [https://developers.google.com/protocol-buffers/docs/overview](https://developers.google.com/protocol-buffers/docs/overview)

As a local example, Denver’s RTD system uses protocol buffers to serialize and make available its transit-related information. RTD schedule data is available in a subset of the protocol buffer schema known as the “General Transit Feed Specification” (GTFS; see [https://developers.google.com/transit/gtfs/](https://developers.google.com/transit/gtfs/)) for use in developing applications and other mobile tools for RTD riders. These data are freely available to developers and educators (access to real-time data requires permission that is readily obtained). You can learn more at [http://rtd-denver.com/gtfs-developer-guide.shtml](http://rtd-denver.com/gtfs-developer-guide.shtml).


2. (Optional, harder) Write a Python, C++, or C# program that accesses the Denver RTD database and displays static current route and time information. You might find these lecture slides helpful.

3. (Optional, harder) Write a Python, C++, or C# program that accesses the Denver RTD database and displays real-time current route and time information. You might find these lecture slides helpful.
Lab 13 – Final Project

The content of the last lab is up to you! The goal is for you to design and implement a meaningful project that will help cement your understanding of the material that we have covered this semester. Most of the parameters of the project are completely up to you, but there are a few ground rules:

- Your project should relate to both what you have learned in the course and to your major or career interests.
- Your project may extend one of the labs you have already completed in some way.
- You are encouraged to choose a project that will enrich your understanding of not only computing, but also your chosen academic path.
- You must submit a written proposal for your project by the due date listed in the course schedule. You do not need to write a book, but please provide enough detail so that your project idea can be evaluated. The purpose of this evaluation is not to limit what you can do, but rather to keep you from undertaking a project whose scope is too large.
- Your project proposal must be approved in writing by the instructor.
- You can work alone if you wish, or you may work with up to two other people. If you wish to form a group, you should include in your written proposal (all people in the proposed group should turn in the same proposal) the details of how responsibilities will be divided, and how you will resolve technical conflict, should it arise. You may not work in a group unless all members of the group wish to do so, and all members have been notified by the instructor in writing that they are authorized to do so.
- Your project should require a significant level of computer programming to complete, and should employ one of the programming languages we have studied in this course, i.e., Python, C++, C#, or Processing.

Finally, you will be expected to make a brief presentation of your project on the last day of class, and to be present at the Inworks Expo to demonstrate and discuss your project. See the course schedule for more details.