1 Basics

Exercise 1.1: The interpreter

Open the Python interpreter. What happens when you input the following statements:

(a) `3 + 1`

(b) `3 * 3`

(c) `2 ** 3`

(d) `"Hello, world!"`

Exercise 1.2: Scripts

Now copy the above to a script, and save it as `script1.py`. What happens if you run the script? (try: `python script1.py`). Can you fix this? (hint: use the `print` function)

Exercise 1.3: More interpreter

Explain the output of the following statements if executed subsequently:

(a) `'py' + 'thon'`

(b) `'py' * 3 + 'thon'`

(c) `'py' - 'py'`

(d) `'3' + 3`

(e) `3 * '3'`

(f) `a`

(g) `a = 3`

(h) `a`

Exercise 1.4: Booleans

Explain the output of the following statements:

(a) `1 == 1`

(b) `1 == True`

(c) `0 == True`

(d) `0 == False`

(e) `3 == 1 * 3`

(f) `(3 == 1) * 3`

(g) `(3 == 3) * 4 + 3 == 1`

(h) `3**5 >= 4**4`

Exercise 1.5: Integers

Explain the output of the following statements:

(a) `5 / 3`

(b) `5 % 3`

(c) `5.0 / 3`

(d) `5 / 3.0`

(e) `5.2 % 3`

(f) `2001 ** 200`

Exercise 1.6: Floats

Explain the output of the following statements:

(a) `2000.3 ** 200` (compare with above)

(b) `1.0 + 1.0 - 1.0`

(c) `1.0 + 1.0e20 - 1.0e20`

Exercise 1.7: Variables

Write a script where the variable `name` holds a string with your name. Then, assuming for now your name is *John Doe*, have the script output: `Hello, John Doe!` (and obviously, do not use `print "Hello, John Doe!"`.

Exercise 1.8: Type casting

Very often, one wants to "cast" variables of a certain type into another type. Suppose we have variable `x = '123'`, but really we would like `x` to be an integer.

This is easy to do in Python, just use `desiredtype(x)`, e.g. `int(x)` to obtain an integer.

Try the following and explain the output

(a) `float(123)`

(b) `float('123')`

(c) `float('123.23')`

(d) `int(123.23)`

(e) `int('123.23')`

(f) `int(float('123.23'))`

(g) `str(12)`

(h) `str(12.2)`

(i) `bool('a')`

(j) `bool(0)`

(k) `bool(0.1)`

2 Control flow

Disclaimer: Some of the following problems are inspired by problems from `www.projecteuler.net`. Have a look if you are interested, there are some great challenges and Python is an excellent tool for solving them.

Exercise 2.1: Range

Type `range(5)` in the interpreter, what does the interpreter return? So what does `for i in range(5)` mean?

Let's also find out whether the interpreter can help us understand the object '`range(5)`' better. Type `type(range(5))` in the interpreter. More on this soon!

Exercise 2.2: For loops

Use a `for` loop to:

(a) Print the numbers 0 to 100

(b) Print the numbers 0 to 100 that are divisible by 7

(c) Print the numbers 1 to 100 that are divisible by 5 but not by 3

(d) Print for each of the numbers $x = 2, \ldots 20$, all numbers that divide x, excluding 1 and x. Hence, for `18`, it should print `2 3 6 9`.

Hint: see https://docs.python.org/2.7/library/functions.html#range.

Exercise 2.3: Simple while loops

Instead of using a for loop, use a while loop to:

(a) Print the numbers 0 to 100

(b) Print the numbers 0 to 100 that are divisible by 7

Exercise 2.4: While loops

Use a `while` loop to find the first 20 numbers that are divisible by 5, 7 and 11, and print them Hint: store the number found so far in a variable.

Pseudo-code:

```
number found = 0
x = 11
while number found is less than 20:
    if x is divisible by 5, 7 and 11:
        print x
        increase number found by 1
    increase x by 1
```

Exercise 2.5: More while loops

The smallest number that is divisible by 2, 3 and 4 is 12. Find the smallest number that is divisible by all integers between 1 and 10.

Exercise 2.6: Collatz sequence

A Collatz sequence is formed as follows: We start with some number x_0, and we find the next number in the sequence by

$$x_{i+1} = \begin{cases} x_i/2 & \text{if } x_i \text{ is even} \\ 3x_i + 1 & \text{if } x_i \text{ is odd} \end{cases}$$

If $x_i = 1$, we stop iterating and have found the full sequence.

For example, if we start with $x_0 = 5$, we obtain the sequence:

```
5 16 8 4 2 1
```

It is conjectured, though not proven, that every chain eventually ends at 1.

Print the Collatz sequence starting at $x_0 = 103$.

3 Functions

Exercise 3.1: Hello

(a) Write a function `hello_world` that prints `'Hello, world!'`

(b) Write a function `hello_name(name)` that prints `'Hello, name!'` where `name` is a string.

(c) Explain the difference between the `print` and `return` keywords. What would change if instead of `print` you would use `return`?

Exercise 3.2: Polynomial
Write a function that evaluates the polynomial $3x^2 - x + 2$.

Exercise 3.3: Maximum
Write a function `my_max(x,y)` that returns the maximum of x and y. Do not use the `max` function, but use `if` instead in following two ways:

(a) Use both `if` and `else`.

(b) Use `if` but not `else` (nor `elif`).

Exercise 3.4: Primes

(a) Write a function `is_prime(n)` that returns `True` only if n is prime.

(b) Note that apart from 2 and 3, all primes are of the form $6k \pm 1$ (though not all numbers of the form $6k \pm 1$ are prime of course). Using this, we can improve the computation time by a factor 3. Update your function to use this.

(c) Write a function that returns all primes up to n.

(d) Write a function that returns the first n primes.

Exercise 3.5: Root finding
Suppose f is a continuous function and $f(a) < 0$ and $f(b) > 0$ for some known a and b. For simplicity, assume $a < b$. Then, there must exist some c such that $f(c) = 0$.

(a) Write a function `root(f, a, b)` that takes a function `f` and two floats `a` and `b` and returns the root `c`. Hint: check the sign at the midpoint of the interval.

(b) Remove the assumption that $a < b$, and that $f(a) < 0$ and $f(b) > 0$, if your current code relies on them.

(c) Add a check that prints
`'function evals have same sign'`
if $f(a) > 0$ and $f(b) > 0$ or if $f(a) < 0$ and $f(b) < 0$.

4 Lists

Exercise 4.1: Short questions

(a) Write a function that prints the elements of a list

(b) Write a function that prints the elements of a list in reverse

(c) Write your own implementation of the `len` function that returns the number of elements in a list.

Exercise 4.2: Copying lists

(a) Create a list `a` with some entries.

(b) Now set `b = a`

(c) Change `b[1]`

(d) What happened to `a`?

(e) Now set `c = a[:]`

(f) Change `c[2]`

(g) What happened to `a`?

Now create a function `set_first_elem_to_zero(l)` that takes a list, sets its first entry to zero, and returns the list.

What happens to the original list?

Exercise 4.3: Lists of lists

What is the difference between `a` and `b`:

```
a = [[]] * 3
```

```
b = [[] for _ in xrange(3)]
```

Exercise 4.4: Lists and functions

Write a function that takes a list and an index, and sets the value of the list at the given index to 0.

Exercise 4.5: Primes

In Section 3 you wrote a function that prints all primes up to n, and a function that prints the first n primes. Update these functions such that they return lists instead.

Exercise 4.6: List comprehensions

Let $i, j = 1, \ldots, n$

(a) Generate a list with elements `[i,j]`.

(b) Generate a list with elements `[i,j]` with $i < j$

(c) Generate a list with elements `i + j` with both i and j prime and $i > j$.

(d) Write a function that evaluates an arbitrary polynomial $a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ using a list comprehension, where you are given `x` and a list with coefficients `coefs` (hint: use enumerate)

Exercise 4.7: Filter

In lecture we have seen how to implement `map` using list comprehensions. Implement `filter` using list comprehensions. Name your functions `myfilter` so you can compare with Python's standard filter.

Exercise 4.8: Flatten a list of lists

Consider having a list with lists as elements, e.g. `[[1,3], [3,6]]`.

Write a function that takes such a list, and returns a list with as elements the elements of the sublists, e.g. `[1, 3, 3, 6]`.

Exercise 4.9: Finding the longest word

Write a function that returns the longest word in a variable `text` that contains a sentence. While `text` may contain punctuation, these should not be taken into account. What happens with ties?

As an example, consider: "Hello, how was the football match earlier today???"

Exercise 4.10: Collatz sequence, part 2

Recall the Collatz sequence problem from Section 6. Our goal is to find the number $n < 1,000,000$ that leads to the longest Collatz sequence.

(a) Write a function that for any n, returns its Collatz sequence as a list

(b) Write a function that finds the integer x that leads to the longest Collatz sequence with $x < n$.

Exercise 4.11: Pivots

Write a function that takes a value `x` and a list `ys`, and returns a list that contains the value `x` and all elements of `ys` such that all values `y` in `ys` that are smaller than `x` come first, then we element `x` and then the rest of the values in `ys`

For example, the output of `f(3, [6, 4, 1, 7])` should be `[1, 3, 6, 4, 7]`

Exercise 4.12: Prime challenge

Write the function `primes(n)` that return a list with all prime numbers up to `n` using three (or less) lines of code.

Hint 1: Use lambda functions and list comprehensions.

Hint 2: Use the first two lines to define two helper (lambda) functions.

5 Tuples

Exercise 5.1: Swapping two values

Suppose you have two variables: `a` and `b`. Now you want to set `a` equal to the value of `b` and at the same time set `b` equal to the value of `a`.

The following obviously does not work

```
a = b
b = a
```

so in some languages, you need to define a third variable like this

```
t = a
a = b
b = t
```

However, in Python you don't need to do this. How can you swap `a` and `b` in one line?

Exercise 5.2: Zip

Suppose we have two lists, `x` and `y` that give the x and y coordinates of a set of points. Create a list with the coordinates (x,y) as a tuple. Hint: Find out about the `zip` function.

You have decided that actually, you need the two seperate lists, but unfortunately, you have thrown them away. How can we use zip to *unzip* the list of tuples to get two lists again?

Exercise 5.3: Distances

Suppose we have two vectors, `x` and `y`, stored as tuples with n elements. Implement functions that compute the l_1 and l_2 distances between `x` and `y`. Note that n is not explicitly given.

6 Dictionaries

Exercise 6.1: Printing a dictionary

Write a function that prints key-value pairs of a dictionary.

Exercise 6.2: Histogram

Write a function that takes a list, and returns a dictionary with keys the elements of the list and as value the number of occurances of that element in the list.

After you are done, look up 'python collections counter' in Google. Could you use a counter instead?

Exercise 6.3: Get method

Dictionaries have a `get` method, which takes a key and a default value. If the key is in the dictionary, it returns the value, otherwise, it returns the default value.

Rewrite your code from the previous problem to make use of this `get` method.

Exercise 6.4: Random text generator

In this question we will start implementing a random text generator. The generated phrases somewhat resemble English, but are usually nonsense. Next week, after we learn about file I/O, we are ready to complete the code.

To generate sentences, we first construct a so-called Markov chain based on actual text data. This is a very basic language model. We can then sample paths from this Markov chain to create new phrases. This is actually easier than it sounds:

A Markov chain consists of states, and transition probabilities between states: i.e. when I am in state A, what is the probability that I'll go to state B?

We focus on the simple case where the state will be the current word. Consider the example sentence 'the fire and the wind.' Then, the states that we move through are

$$\text{BEGIN} \to \text{the} \to \text{fire} \to \text{and} \to \text{the} \to \text{wind.} \to \text{END}$$

where 'BEGIN' and 'END' are special states for the beginning and the end of the sentence. To find the transition probabilities, we go over a large body of text and record current word and the next word. In the above example, 'BEGIN' is followed by 'the', and 'the' is followed by 'fire' and 'wind'.

We won't be computing actual probabilities. Instead, we create a dictionary that, for every word, contains all the words that follow it. To generate a phrase, we start at the 'BEGIN' state, and pick randomly one word from the list of words that follows the 'BEGIN' state. Then we look up which words follow that word, and again pick one word at random, until we hit the 'END' state, which signals that we are done.

Before you get started, download the starter code by using
`$ git clone https://github.com/schmit/Markov-chain-startercode.git` (if you have git / use Cloud9) or dowload the code directly:
https://github.com/schmit/Markov-chain-startercode/archive/master.zip

Then

- Implement the `process_line` function, which takes a line as input, and returns a list with tuples with the current state, and the next state.
- Implement the `process_textfile` function, which loops over text, calls `process_line` to extract the transitions, and adds these to a dictionary. For now, do not worry about reading data from a file, the lines are given as elements of the list `f`.
- Implement the `generate_line` function, which generates random phrases based on a dictionary with transitions.

See `markov.py` for a more detailed description of each function.

To run the code, use `python markov.py <filename>`. Since we are not using any files yet, replace `<filename>` with a random word.

Exercise 6.5: Vector functions

Let's implement some vector functions. There are two types of vectors, normal or dense vectors, which we can represent using lists. For sparse vectors, where many of the elements are zero, this is inefficient. Instead, we use a dictionary with keys the indices of non-zero values, and then the value corresponding to the key is the value of the vector at that index. Hence, the vector $[1, 2, 4]$ can be stored as a list: `[1, 2, 4]` or as a dictionary `{0:1, 1: 2, 2: 4}`.

(a) Write a function that adds two (dense) vectors

(b) Write a function that multiplies (i.e. inner product) two (dense) vectors

(c) Write a function that adds two sparse vectors

(d) Write a function that multiplies two sparse vectors

(e) Write a function that adds a sparse vector and a dense vector

(f) Write a function that multiplies a sparse vector and a dense vector

Exercise 6.6: Reverse look-up

Dictionaries are made to look up values by keys. Suppose however, we want to find the key that is associated with some value. Write a function that takes a dictionary and a value, and returns the key associated with this value.

What challenges do you face? How would you deal with those challenges?

7 File I/O

Exercise 7.1: Open a file

Write a function that opens a file (input: filename), and prints the file line by line.

Exercise 7.2: Wordcount

On the course website you can find a text file containing the complete works of William Shapespeare.

(a) Find the 20 most common words

(b) How many unique words are used?

(c) How many words are used at least 5 times?

(d) Write the 200 most common words, and their counts, to a file.

Exercise 7.3: Random text generator II

In this exercise, we will finish the implementation of our random text generator from Question 6.4.

Change the `process_textfile` function such that it reads a file line by line and processes it, instead of using the hardcoded sample. This should require only a few changes to the code.

Generate some random sentences based on the books provided in the `data/` folder.

As you will notice, most phrases are very random. To make the phrases seem a little more realistic, we should use not just the last word, but the two last words. Going back to our previous example: 'the fire and the wind.' we would have the (state, new_word) pairs: ('the fire', 'and'), ('fire and', 'the'), ('and the', 'wind') etc. Update your code to use states consisting of two words instead of one.

Bonus: Using more than 2 words is infeasible unless we have massive amounts of text data because the number of states gets increasingly large. Change your code such that you can specify `k`, the number of words per state.

Exercise 7.4: Sum of lists

Before you start coding, please read the entire problem.

(a) Data generation

Write a function that takes three integers, n, a and b and a filename and writes to the file a list with n random integers between a and b.

(b) Reading the data

Write a function that can read the files as generated above and return the values.

(c) Sum problem

Write a function that given two filenames (pointing to files as generated by the above function) and an integer k, finds all u and v such that $u + v = k$, and u is an element of the first list and v is a member of the second list.

(d) Testing

Test your functions by generating 2 files with $n = 2000$, $a = 1$, $b = 10000$ and $k = 5000$ and $k = 12000$.

(e) Bonus: Efficiency

If you are up to a challenge, write a function that solves the sum problem with the restriction that you can only go over every number in the both lists once.

8 Classes

Exercise 8.1: Rational numbers

In this problem, we will write a class that can represent rational numbers, i.e. fractions $\frac{p}{q}$.

(a) Create a class `Rational` which is initialized by two integers, p and q, the nominator and denominator

(b) Add a method to print the rational number as p/q (the `__str__` or `__repr__` method is useful).

(c) We would like to represent $\frac{10}{20}$ by $\frac{1}{2}$ instead, hence write a function that computes the greatest common divisor, and ensure that every rational number is simplified

(d) Add a method so that we can add two rational numbers with `r1 + r2`, here the `__add__()` method is useful.

(e) Add a method to subtract two rational numbers. (`__sub__`)

(f) Add a method to multiply two rational numbers. (`__mul__`)

(g) Add a method to divide two rational numbers. (`__div__`)

(h) Add a method that compares whether two rational numbers are equal.

(i) Add a method to convert the rational number to a floating point (the `__float__()` method may be handy).

(j) Add any more functionality that you think is useful but I failed to mention.

Exercise 8.2: Rock Paper Scissors

In this problem, we will finish an implementation for Rock-Paper-Scissors. We have written some code to get you started, now it's up to you to finish the implementation.

The code consists of 2 files: game.py and agent.py. The code that implements the actual game is coded in game.py. agent.py defines several agents that can play the game. Download the starter code here using
`$ git clone https://github.com/schmit/Rock-paper-scissors-startercode.git` (if you have git / use Cloud9) or dowload the code directly:
`https://github.com/schmit/Rock-paper-scissors-startercode/archive/master.zip`

(a) Finish the implementation of game.py by implementing the compare function, updating the scores (where a win is 1 point, and a tie or loss 0 points), and finally the summary function that gives some information after the game.

(b) Implement the HumanAgent in agent.py, this agent should query the user for the next move, and ensure that the user gives valid input.

(c) Implement MyAgent, where you can implement your own strategy, try to beat the InstructorAgent consistently over 100 rounds.

Hint: have a look at the Hangman code.

Exercise 8.3: Hangman agent

Implement your own Hangman computer agent (see exercise 2.??) that is much more effective than the Agent that guesses random characters.

Make sure you create a new class rather than overwriting the existing `Agent` class. You can of course inherit from the `Agent` class

You can update the `simulate.py` script to test your implementation.

Exercise 8.4: Sparse and dense vectors

In exercise 6.5 you implemented functions for sparse and dense vector multiplications using lists and dictionaries. However, this is a bit clumsy to use in practice. Really, we would like to represent sparse and dense vectors as classes, this way we can overload operators such as `+` (`__add__`) and get sensible output. For example, using `+` on two dense vectors implemented as lists would append the second vector to the first, instead of adding the two together.

Implement sparse and dense vectors. Both classes should have the following capabilities:

(a) Print vector

(b) Add two vectors (both if other is dense and sparse)

(c) Multiply two vectors (both if other is dense and sparse)

Do re-use your code from the previous exercise.

Hint: `isinstance()` might be useful.

Exercise 8.5: Implementing the set class

Write a class `mySet` that has the same basic functionality as the Python `set` data structure. Base your implementation on a dictionary.

Exercise 8.6: Binary search tree

In this exercise, we will implement a binary search tree. See `http://en.wikipedia.org/wiki/Binary_search_tree` for an explanation.

(a) Define a class `Node`, and write the constructor, which takes one argument, `value`, and initializes the left and right children to None.

(b) Write a function to print the tree.

(c) Write a function that inserts a new value in the tree at the right location.

(d) Write a function that looks up a value in the tree.

(e) Write a function that removes a value from the tree.

Exercise 8.7: Ordinary least squares

Our goal in this exercise is to write our own least-squares solver to solve regression problems:

$$\arg\min_{\beta} \|y - X\beta\|_2$$

See for example statsmodels `ols` or `LinearRegression`. While one can, and should, use written solvers, it's a good practice exercise.

(a) Setup an OLS class with fit and predict methods, to be coded later

(b) Write the fit method using numpy's or scipy's linear algebra module.

(c) Now write the predict function, that predicts y_n given new X_n.

(d) Add a function that summarizes the model

(e) (Optional) Use Patsy and Pandas to support DataFrames and formulas, similar to `R`.

9 Numpy

Generate matrices A, with random Gaussian entries, B, a Toeplitz matrix, where $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times m}$, for n = 200, m = 500.

Exercise 9.1: Matrix operations
Calculate $A + A$, $AA^\top, A^\top A$ and AB. Write a function that computes $A(B - \lambda I)$ for any λ.

Exercise 9.2: Solving a linear system
Generate a vector b with m entries and solve $Bx = b$.

Exercise 9.3: Norms
Compute the Frobenius norm of A: $\|A\|_F$ and the infinity norm of B: $\|B\|_\infty$. Also find the largest and smallest singular values of B.

Exercise 9.4: Power iteration
Generate a matrix Z, $n \times n$, with Gaussian entries, and use the power iteration to find the largest eigenvalue and corresponding eigenvector of Z. How many iterations are needed till convergence?

Optional: use the `time.clock()` method to compare computation time when varying n.

Exercise 9.5: Singular values
Generate an $n \times n$ matrix, denoted by C, where each entry is 1 with probability p and 0 otherwise. Use the linear algebra library of Scipy to compute the singular values of C. What can you say about the relationship between n, p and the largest singular value?

Exercise 9.6: Nearest neighbor
Write a function that takes a value z and an array A and finds the element in A that is closest to z. The function should return the closest value, not index.

Hint: Use the built-in functionality of Numpy rather than writing code to find this value manually. In particular, use brackets and `argmin`.

10 Scipy

Exercise 10.1: Least squares
Generate matrix $A \in R^{m \times n}$ with $m > n$. Also generate some vector $b \in R^m$.

Now find $x = \arg\min_x \|Ax - b\|_2$.

Print the norm of the residual.

Exercise 10.2: Optimization
Find the maximum of the function

$$f(x) = \sin^2(x - 2)e^{-x^2}$$

Exercise 10.3: Pairwise distances
Let X be a matrix with n rows and m columns. How can you compute the pairwise distances between every two rows?

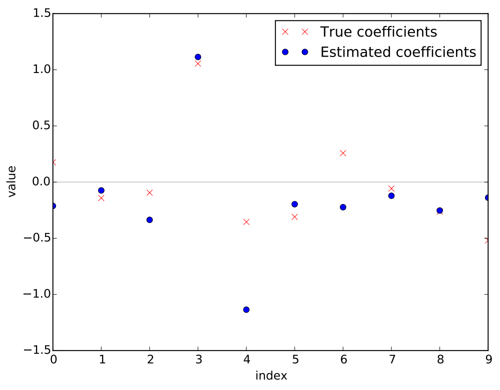

Figure 1: Parameter plot

As an example application, consider n cities, and we are given their coordinates in two columns. Now we want a nice table that tells us for each two cities, how far they are apart.

Again, make sure you make use of Scipy's functionality instead of writing your own routine.

11 Matplotlib

Exercise 11.1: Plotting a function

Plot the function

$$f(x) = \sin^2(x-2)e^{-x^2}$$

over the interval $[0, 2]$. Add proper axis labels, a title, etc.

Exercise 11.2: Data

Create a data matrix X with 20 observations of 10 variables. Generate a vector b with parameters Then generate the response vector $y = Xb+z$ where z is a vector with standard normally distributed variables.

Now (by only using y and X), find an estimator for b, by solving

$$\hat{b} = \arg\min_b \|Xb - y\|_2$$

Plot the true parameters b and estimated parameters $\hat{b}$. See Figure 1 for an example plot.

Exercise 11.3: Histogram and density estimation

Generate a vector z of 10000 observations from your favorite exotic distribution. Then make a plot that shows a histogram of z (with 25 bins), along with an estimate for the density, using a Gaussian kernel density estimator (see scipy.stats). See Figure 2 for an example plot.

12 Pandas

Exercise 12.1: Exploring historical stock prices

Using either the pandas-datareader package (introduced in Lecture 6 `https://github.com/pydata/pandas-datareader` or a CSV of stock prices of your choice, complete the steps below. I am also hosting a CSV at `stanford.edu/~bmj/ex/data/AAPL_2008-2017.csv` if you would rather use that.

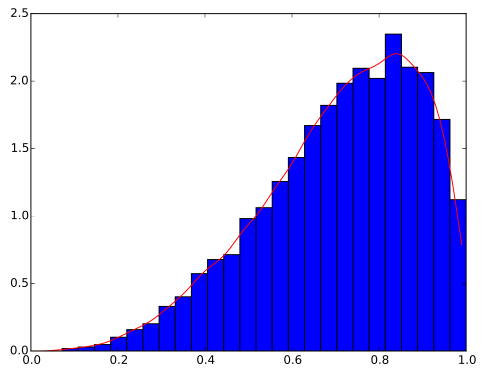

Figure 2: Histogram

(a) Plot the stock price of your stock over the past year. (Hint: Use a DateTimeIndex on your frame)

(b) Append a column to your frame called 'Return' which has the daily return of the stock. This daily return at time t can be expressed as $\frac{P_t - P_{t-1}}{P_{t-1}}$ where P_t is the price of the stock.

(c) What is the volatility of the returns over the past 6M? Volatility is a measure of fluctuation away from the mean - you can use the standard deviation here. Please provide as single scalar.

(d) Which day had the best return in the past year? Worst return? Please provide two dates.

13 Recursion

Exercise 13.1: Power
Write a recursive function that computes a^b for given a and b, where b is an integer. Do not use `**`.

Exercise 13.2: Purify
Write two functions, one that uses iteration (say a for loop), and the other using recursion, that achieve the following: The input of the function is a list with integers. The functions return a (new) list with only the even integers in the list.

Exercise 13.3: Product
Write two functions, one that uses iteration, and the other using recursion, that achieve the following: The input of the function is a list with numbers. The functions return the product of the numbers in the list.

Exercise 13.4: Factorial
Write a recursive function to compute $n! = n \times (n-1) \times \ldots \times 1$. Note that 0! is defined to equal 1.

Exercise 13.5: Fibonacci sequence
The Fibonacci sequence $\{F_i\}_i = 0^\infty$ starts with $F_0 = 0, F_1 = 1$. Every subsequent value in the sequence is the sum of the last elements in the sequence:

$$F_n = F_{n-1} + F_{n-2}$$

(a) Implement a non-recursive function that computes the F_n

(b) Implement a recursive function that computes F_n

(c) Compare the runtime for computing F_{35} recursively versus non-recursively, and explain the difference.

(d) This does not mean a recursion is not feasible for this problem, only that the naive implementation isn't the best. We can get a better version using either of the following

  (a) Store values already calculated, say in a dictionary, so you can look them up instead of redoing the calculation.

  (b) Generalizing the Fibonacci sequence to an additive sequence with arbitrary starting points t_0 and t_1, and finding a recursive algorithm to find the nth term in such a sequence. Note that finding the nth term in a sequence started from t_0 and t_1 is the same as finding the $n-1$th term in a sequence started from t_1 and $t_0 + t_1$.

Implement one of the above (or both).

Exercise 13.6: Palindromes

Given a string `t`, we are interested in finding the largest palindrome in `t`, where we are allowed to remove characters from `t`. For example, consider the string `abcdba`, then the function should return `abcba`. Before you start coding, figure out the recursion on paper first.

Extra: you will notice that if you are not careful, this will take a very long time to compute for longer inputs. However, a simple modification can speed up your code so that it runs in $\mathcal{O}(n^2)$, where n is the length of the input string. Find and implement this modification.[1]

14 Exception handling

Exercise 14.1: Rational numbers

Edit your code that implements the `Rational` class such that it raises an exception when the denominator is 0.

Exercise 14.2: Wordcount

Recall the exercise of finding the 20 most common words in the Complete works of Shakespeare.

Write a script, reusing as much of your code as you can, such that you can specify the filename and k for the k most common words at the command line and that the script will print the k most common words, and their counts, from the file.

Make sure you handle errors gracefully, such as a misspecified filename.

Exercise 14.3: Historical stock data

Add an exception to 12.1 to catch an exception and fail gracefully if the file you try to read is not available.

[1]This exercise is inspired by an exercise in cs221.