Exercises

Exercise 1. (*Sum of Integers*) Implement the function `_sumOfInts()` in `sum_of_ints.py` that takes an integer n as argument and returns the sum $S(n) = 1 + 2 + 3 + \cdots + n$, computed recursively using the recurrence equation

$$S(n) = \begin{cases} 1 & \text{if } n = 1, \\ n + S(n-1) & \text{if } n > 1. \end{cases}$$

```
>_ ~/workspace/project5
$ python3 sum_of_ints.py 100
5050
```

Exercise 2. (*Bit Counts*) Implement the functions `_zeros()` and `_ones()` in `bits.py` that take a bit string (ie, a string of zeros and ones) s as argument and return the number of zeros and ones in s, each computed recursively. The *number of zeros* in a bit string is 1 or 0 (if the first character is `'0'` or `'1'`) plus the *number of zeros* in the rest of the string; *number of zeros* in an empty string is 0 (base case). The *number of ones* in a bit string can be defined analogously.

```
>_ ~/workspace/project5
$ python3 bits.py 1010010010011110001011111
zeros = 11, ones = 14, total = 25
```

Exercise 3. (*String Reversal*) Implement the function `_reverse()` in `reverse.py` that takes a string s as argument and returns the reverse of the string, computed recursively. The *reverse* of a string is the last character concatenated with the *reverse* of the string up to the last character; the *reverse* of an empty string is an empty string (base case).

```
>_ ~/workspace/project5
$ python3 reverse.py bolton
notlob
```

Exercise 4. (*Palindrome*) Implement the function `_isPalindrome()` in `palindrome.py`, using recursion, such that it returns `True` if the argument s is a palindrome (ie, reads the same forwards and backwards), and `False` otherwise. You may assume that s is all lower case and doesn't include any whitespace characters. A string is a *palindrome* if the first character is the same as the last *and* the rest of the string is a *palindrome*; an empty string is a *palindrome* (base case).

```
>_ ~/workspace/project5
$ python3 palindrome.py bolton
False
$ python3 palindrome.py madam
True
```

Exercise 5. (*Password Checker*) Implement the function `_isValid()` in `password_checker.py` that returns `True` if the given password string meets the following requirements, and `False` otherwise:

- Is at least eight characters long
- Contains at least one digit (0-9)
- Contains at least one uppercase letter
- Contains at least one lowercase letter
- Contains at least one character that is neither a letter nor a number

```
>_ ~/workspace/project5
$ python3 password_checker.py Abcde1fg
False
$ python3 password_checker.py Abcde1@g
True
```

Hint: use the `str` methods `isdigit()`, `isupper()`, `islower()`, and `isalnum()`.

Exercise 6. (*2D Point*) Define a data type called `Point` in `point.py` that represents a point in 2D. The data type must support the following API:

point.Point	
`Point(x, y)`	constructs a point `p` from the given `x` and `y` values
`p.distanceTo(q)`	returns the Euclidean distance between `p` and `q`
`str(p)`	returns a string representation of `p` as `'(x, y)'`

```
>_ ~/workspace/project5
$ python3 point.py 0 1 1 0
p1         = (0.0, 1.0)
p2         = (1.0, 0.0)
d(p1, p2) = 1.4142135623730951
```

Exercise 7. (*1D Interval*) Define a data type called `Interval` in `interval.py` that represents a closed 1D interval. The data type must support the following API:

interval.Interval	
`Interval(lbound, rbound)`	constructs an interval `i` given its lower and upper bounds
`i.lower()`	returns the lower bound of `i`
`i.upper()`	returns the upper bound of `i`
`i.contains(x)`	returns `True` if `i` contains the value `x`, and `False` otherwise
`i.intersects(j)`	returns `True` if `i` intersects interval `j`, and `False` otherwise
`str(i)`	returns a string representation of `i` as `'[lbound, rbound]'`

```
>_ ~/workspace/project5
$ python3 interval.py 3.14
0 1 0.5 1.5 1 2 1.5 2.5 2.5 3.5 3 4
[2.5, 3.5] contains 3.140000
[3.0, 4.0] contains 3.140000
[0.0, 1.0] intersects [0.5, 1.5]
[0.0, 1.0] intersects [1.0, 2.0]
[0.5, 1.5] intersects [1.0, 2.0]
[0.5, 1.5] intersects [1.5, 2.5]
[1.0, 2.0] intersects [1.5, 2.5]
[1.5, 2.5] intersects [2.5, 3.5]
[2.5, 3.5] intersects [3.0, 4.0]
```

Exercise 8. (*Rectangle*) Define a data type called `Rectangle` in `rectangle.py` that represents a rectangle using 1D intervals (ie, `Interval` objects) to represent its x (width) and y (height) segments. The data type must support the following API:

rectangle.Rectangle	
`Rectangle(xint, yint)`	constructs a rectangle `r` given its `x` and `y` segments, each an `Interval` object
`r.area()`	returns the area of rectangle `r`
`r.perimeter()`	returns the perimeter of rectangle `r`
`r.contains(x, y)`	returns `True` if `r` contains the point `(x, y)`, and `False` otherwise
`r.intersects(s)`	returns `True` if `r` intersects rectangle `s`, and `False` otherwise
`str(r)`	returns a string representation of `r` as `'[x1, x2] x [y1, y2]'`

```
>_ ~/workspace/project5
$ python3 rectangle.py 1.01 1.34
0 1 0 1 0.7 1.2 .9 1.5
Area([0.0, 1.0] x [0.0, 1.0]) = 1.000000
Perimeter([0.0, 1.0] x [0.0, 1.0]) = 4.000000
Area([0.7, 1.2] x [0.9, 1.5]) = 0.300000
Perimeter([0.7, 1.2] x [0.9, 1.5]) = 2.200000
[0.7, 1.2] x [0.9, 1.5] contains (1.010000, 1.340000)
[0.0, 1.0] x [0.0, 1.0] intersects [0.7, 1.2] x [0.9, 1.5]
```

Problems

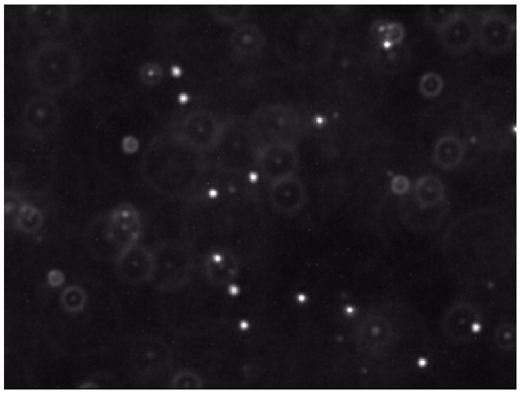
Goal The purpose of this project is to re-affirm the atomic nature of matter by tracking the motion of particles undergoing Brownian motion, fitting this data to Einstein's model, and estimating Avogadro's constant.

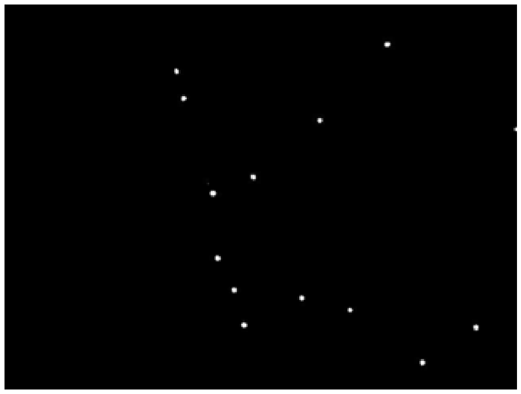
Background The atom played a central role in 20th century physics and chemistry, but prior to 1908 the reality of atoms and molecules was not universally accepted. In 1827, the botanist Robert Brown observed the random erratic motion of wildflower pollen grains immersed in water using a microscope. This motion would later become known as *Brownian motion*. Einstein hypothesized that this Brownian motion was the result of millions of tiny water molecules colliding with the larger pollen grain particles.

In one of his "miracle year" (1905) papers, Einstein formulated a quantitative theory of Brownian motion in an attempt to justify the "existence of atoms of definite finite size." His theory provided experimentalists with a method to count molecules with an ordinary microscope by observing their collective effect on a larger immersed particle. In 1908 Jean Baptiste Perrin used the recently invented ultramicroscope to experimentally validate Einstein's kinetic theory of Brownian motion, thereby providing the first direct evidence supporting the atomic nature of matter. His experiment also provided one of the earliest estimates of Avogadro's constant. For this work, Perrin won the 1926 Nobel Prize in physics.

The Problem In this project, you will redo a version of Perrin's experiment. Your job is greatly simplified because with modern video and computer technology (in conjunction with your programming skills), it is possible to accurately measure and track the motion of an immersed particle undergoing Brownian motion. We supply video microscopy data of polystyrene spheres ("beads") suspended in water, undergoing Brownian motion. Your task is to write a program to analyze this data, determine how much each bead moves between observations, fit this data to Einstein's model, and estimate Avogadro's constant.

Here is a movie (avi, mov) of several beads undergoing Brownian motion. Below is a typical raw image (left) and a cleaned up version (right) using thresholding, as described below.





Each image shows a two-dimensional cross section of a microscope slide. The beads move in and out of the microscope's field of view (the x- and y-directions). Beads also move in the z-direction, so they can move in and out of the microscope's depth of focus; this results in halos, and it can also result in beads completely disappearing from the image.

Problem 1. (*Particle Representation*) Define a data type called `Blob` in `blob.py` to represent a particle (aka blob). The data type must support the following API:

Blob	
`Blob()`	constructs an empty blob `b`
`b.add(x, y)`	adds a pixel `(x, y)` to *b*
`b.mass()`	returns the mass of `b`, ie, the number of pixels in it
`b.distanceTo(c)`	returns the Euclidean distance between the center of mass of `b` and the center of mass of blob `c`
`str(b)`	returns a string representation of `b`

```
>_ ~/workspace/project5
$ python3 blob.py
1 0 0 1 -1 0 0 -1
<ctrl-d>
a           = 1 (0.0000, 0.0000)
b           = 4 (0.0000, 0.0000)
dist(a, b) = 0.0
```

Directions:

- `Blob`
  - Instance variables:
    - x-coordinate of center of mass, `_x` (float).
    - y-coordinate of center of mass, `_y` (float).
    - Number of pixels, `_pixels` (int).
  - `Blob()`
    - Initialize the instance variables appropriately.
  - `b.add(x, y)`
    - Use the idea of *running average*[1] to update the center of mass of blob `b`.
    - Increment the number of pixels in blob `b` by 1.
  - `b.mass()`
    - Return the number of pixels in the blob `b`.
  - `b.distanceTo(c)`
    - Return the Euclidean distance between the center of mass of blob `b` and the center of mass of blob `c`.

Problem 2. (*Particle Identification*) The first challenge is to identify the beads amidst the noisy data. Each image is 640-by-480 pixels, and each pixel is represented by a `Color` object which needs to be converted to a luminance value ranging from 0.0 (black) to 255.0 (white). Whiter pixels correspond to beads (foreground) and blacker pixels to water (background). We break the problem into three pieces:

1. *Read the image.* Use the `Picture` data type to read in the image.
2. *Classify the pixels as foreground or background.* We use a simple, but effective, technique known as *thresholding* to separate the pixels into foreground and background components: all pixels with monochrome luminance values strictly below some threshold τ (tau) are considered background, and all others are considered foreground. The two pictures in figure above illustrate the original frame (left) and the same frame after thresholding (right), using $\tau = 180.0$. This value of τ results in an effective cutoff for the supplied data.

[1] If $\bar{x}_{n-1}$ is the average value of $n-1$ points $x_1, x_2, \ldots, x_{n-1}$, then the average value $\bar{x}_n$ of n points $x_1, x_2, \ldots, x_{n-1}, x_n$ is $\frac{\bar{x}_{n-1}\cdot(n-1)+x_n}{n}$.

3. *Find the blobs.* A polystyrene bead is typically represented by a disc-like shape of at least some minimum number *pixels* (typically 25) of connected foreground pixels. A blob or connected component is a maximal set of connected foreground pixels, regardless of its shape or size. We will refer to any blob containing at least *pixels* number of pixels as a bead. The center-of-mass of a blob (or bead) is the average of the x- and y-coordinates of its constituent pixels.

Define a data type called `BlobFinder` in `blob_finder.py` that supports the following API. Use depth-first search to efficiently identify the blobs.

BlobFinder	
`BlobFinder(pic, tau)`	constructs a blob finder `bf` to find blobs in the picture `pic` using a luminance threshold `tau`
`bf.getBeads(pixels)`	returns a list of all blobs with mass $\geq$ `pixels`, ie, a list of beads

```
~/workspace/project5
$ python3 blob_finder.py 25 180.0 data/run_1/frame00001.jpg
13 Beads:
29 (214.7241, 82.8276)
36 (223.6111, 116.6667)
42 (260.2381, 234.8571)
35 (266.0286, 315.7143)
31 (286.5806, 355.4516)
37 (299.0541, 399.1351)
35 (310.5143, 214.6000)
31 (370.9355, 365.4194)
28 (393.5000, 144.2143)
27 (431.2593, 380.4074)
36 (477.8611, 49.3889)
38 (521.7105, 445.8421)
35 (588.5714, 402.1143)
15 Blobs:
29 (214.7241, 82.8276)
36 (223.6111, 116.6667)
1 (254.0000, 223.0000)
42 (260.2381, 234.8571)
35 (266.0286, 315.7143)
31 (286.5806, 355.4516)
37 (299.0541, 399.1351)
35 (310.5143, 214.6000)
31 (370.9355, 365.4194)
28 (393.5000, 144.2143)
27 (431.2593, 380.4074)
36 (477.8611, 49.3889)
38 (521.7105, 445.8421)
35 (588.5714, 402.1143)
13 (638.1538, 155.0000)
```

The program identifies 15 blobs in the sample frame, 13 of which are beads. Our string representation of a blob specifies its mass (number of pixels) and its center of mass (in the 640-by-480 picture). By convention, pixels are measured from left-to-right, and from top-to-bottom (instead of bottom-to-top).

Directions:

- Instance variable:
  - Blobs identified by this blob finder, `_blobs` (list of `Blob` objects).
- `BlobFinder()`
  - Initialize `blobs` to an empty list.
  - Create a 2D list of booleans called `marked`, having the same dimensions as `pic`.
  - Enumerate the pixels of `pic`, and for each pixel `(i, j)`:
    * create a `Blob` object called `blob`;
    * call `_findBlob()` with the appropriate arguments; and
    * add `blob` to `blobs` if it has a non-zero mass.

- `bf._findBlob()`
  - Base case: return if pixel `(i, j)` is out of bounds, or if it is marked, or if its luminance (use the function `luminance.luminance()` for this) is less than `tau`.
  - Mark the pixel `(i, j)`.
  - Add the pixel `(i, j)` to the blob `blob`.
  - Recursively call `_findBlob()` on the N, E, W, and S pixels.
- `bf.getBeads(pixels)`
  - Return a list of blobs from `blobs` that have a mass $\geq$ `pixels`.

Problem 3. (*Particle Tracking*) The next step is to determine how far a bead moved from one time step t to the next $t+\Delta t$. For our data, $\Delta t = 0.5$ seconds per frame. We assume the data is such that each bead moves a relatively small amount, and that two beads do not collide. However, we must account for the possibility that the bead disappears from the frame, either by departing the microscope's field of view in the x- or y- direction, or moving out of the microscope's depth of focus in the z-direction. Thus, for each bead at time $t + \Delta t$, we calculate the closest bead at time t (in Euclidean distance) and identify these two as the same beads. However, if the distance is too large, ie, greater than Δ (delta) pixels, we assume that one of the beads has either just begun or ended its journey. We record the displacement that each bead travels in the Δt units of time.

Implement a program called `bead_tracker.py` that accepts p (int), *tau* (float), *delta* (float), and a sequence of JPEG filenames as command-line arguments; identifies the beads in each JPEG image using `BlobFinder`; and writes to standard output (one per line, formatted with 4 decimal places to the right of decimal point) the radial distance that each bead moves from one frame to the next (assuming it is no more than *delta*). Note that it is not necessary to explicitly track a bead through a sequence of frames — you only need to worry about identifying the same bead in two consecutive frames.

```
>_ ~/workspace/project5

$ python3 bead_tracker.py 25 180.0 25.0 data/run_1/frame00000.jpg data/run_1/frame00001.jpg
7.1833
4.7932
2.1693
5.5287
5.4292
4.3962
```

Directions:

- Accept command-line arguments `pixels` (int), `tau` (float), and `delta` (float).
- Construct a `BlobFinder` object for the frame `sys.argv[4]` and from it get a list of beads `prevBeads` that have at least `pixels` pixels.
- For each frame starting at `sys.argv[5]`:
  - Construct a `BlobFinder` object and from it get a list of beads `currBeads` that have at least `pixels` pixels.
  - For each bead `currBead` in `currBeads`, find a bead `prevBead` from `prevBeads` that is no further than `delta` and is closest to `currBead`, and if such a bead is found, write its distance (using format string `'%.4f\n'`) to `currBead`.
  - Write a newline character.
  - Set `prevBeads` to `currBeads`.

Problem 4. (*Data Analysis*) Einstein's theory of Brownian motion connects microscopic properties (eg, radius, diffusivity) of the beads to macroscopic properties (eg, temperature, viscosity) of the fluid in which the beads are immersed. This amazing theory enables us to estimate Avogadro's constant with an ordinary microscope by observing the collective effect of millions of water molecules on the beads.

1. *Estimating the self-diffusion constant.* The self-diffusion constant D characterizes the stochastic movement of a molecule (bead) through a homogeneous medium (the water molecules) as a result of random thermal energy. The Einstein-Smoluchowski equation states that the random displacement of a bead in one dimension has a Gaussian distribution with mean zero and variance $\sigma^2 = 2D\Delta t$, where Δt is the time interval between position measurements. That is, a molecule's mean displacement is zero and its mean square displacement is proportional to the elapsed time between measurements, with the constant of proportionality $2D$. We estimate σ^2 by computing the variance of all observed bead displacements in the x and y directions. Let $(\Delta x_1, \Delta y_1), \ldots, (\Delta x_n, \Delta y_n)$ be the n bead displacements, and let $r_1, \ldots, r_n$ denote the radial displacements. Then
$$\begin{aligned}\sigma^2 &= \frac{(\Delta x_1^2 + \cdots + \Delta x_n^2) + (\Delta y_1^2 + \cdots + \Delta y_n^2)}{2n} \\ &= \frac{r_1^2 + \cdots + r_n^2}{2n}.\end{aligned}$$
For our data, $\Delta t = 0.5$ so our estimate for σ^2 is an estimate for D as well. Note that the radial displacements in the formula above are measured in meters. The radial displacements output by your `bead_tracker.py` program are measured in pixels. To convert from pixels to meters, multiply by 0.175×10^{-6} (meters per pixel). The value of n is the count of the total number of displacements read.

2. *Estimating the Boltzmann constant.* The Stokes-Einstein relation asserts that the self-diffusion constant D of a spherical particle immersed in a fluid is given by $D = \frac{kT}{6\pi\eta\rho}$, where, for our data T (absolute temperature) is 297 degrees Kelvin (room temperature), η (viscosity of water) is 9.135×10^{-4} Nsm^{-2} (at room temperature), ρ (radius of bead) is 0.5×10^{-6}, and k is the *Boltzmann constant*. All parameters are given in SI units. The Boltzmann constant is a fundamental physical constant that relates the average kinetic energy of a molecule to its temperature. Use $k = \frac{6\pi D\eta\rho}{T}$ as an estimate of Boltzmann's constant.

3. *Estimating Avogadro's constant.* Avogadro's constant N_A is defined to be the number of particles in a mole. By definition, $k = \frac{R}{N_A}$, where the universal gas constant R is approximately 8.31457 JK^{-1}mol^{-1}. Use $N_A = \frac{R}{k}$ as an estimate of Avogadro's constant.

Implement a program called `data_analysis.py` that accepts the displacements (output of `bead_tracker.py`) from standard input; computes an estimate of Boltzmann's constant and Avogadro's constant using the formulae described above; and writes the values to standard output, separated by a space.

```
>_ ~/workspace/project5

$ python3 bead_tracker.py 25 180.0 25.0 data/run_1/* | python3 data_analysis.py
1.253509e-23 6.633037e+23
```

Directions:

- Initialize `ETA`, `RHO`, `T`, and `R` to appropriate values.
- Calculate `var` as the sum of the squares of the `n` displacements (each converted from pixels to meters) read from standard input.
- Divide `var` by `2 * n`.
- Estimate Boltzmann's constant as `6 * math.pi * var * ETA * RHO / T`.
- Estimate Avogadro's constant as `R / k`.
- Write to standard output the two constants in scientific notation (using the format string `'%e'`) and separated by a space.

Data Be sure to test your programs thoroughly using ten datasets (they are under the `data` folder), obtained by William Ryu (Princeton University) using fluorescent imaging. Each run contains a sequence of two hundred 640-by-480 color JPEG images, `frame00000.jpg` through `frame00199.jpg` and is stored in a subfolder `run_1` through `run_10`, and the folder also contains some reference solutions.

Acknowledgements This project is an adaptation of the Atomic Nature of Matter assignment developed at Princeton University by David Botstein, Tamara Broderick, Ed Davisson, Daniel Marlow, William Ryu, and Kevin Wayne.

Files to Submit

1. `sum_of_ints.py`
2. `bits.py`
3. `reverse.py`
4. `palindrome.py`
5. `password_checker.py`
6. `point.py`
7. `interval.py`
8. `rectangle.py`
9. `blob.py`
10. `blob_finder.py`
11. `bead_tracker.py`
12. `data_analysis.py`
13. `report.txt`

Before you submit your files, make sure:

- You do not use concepts from sections beyond "Designing Data Types".
- Your programs meet the style requirements by running the following command in the terminal.

```
>_ ~/workspace/project5
$ pycodestyle <program>
```

- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You use the template file `report.txt` for your report.
- Your report meets the prescribed guidelines.