Introduction to Programming in Python

Assignment 5 (Atomic Nature of Matter) Discussion

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/atomic_nature_of_matter

```
$ python3 sum_of_ints.py 100
5050
```

Part I (Warmup Problems) · Problem 1 (Sum of Integers)

Base case: if $n = 1$, return 1

Recursive step: return n plus _sumOfInts($n - 1$)

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

```
>_ ~/workspace/atomic_nature_of_matter

$ python3 bits.py 1010010010011110001011111
zeros = 11, ones = 14, total = 25
```

`_zeros(s)`

- Base case: if s is empty, return 0
- Recursive step: if the first character of s is a 0, return 1 plus _zeros(s[1 :]); otherwise return _zeros(s[1 :])

`_ones(s)`

- Base case: if s is empty, return 0
- Recursive step: if the first character of s is a 1, return 1 plus _ones(s[1 :]); otherwise return _ones(s[1 :])

Implement the function `_reverse()` in `reverse.py` that takes a string s as argument and returns the reverse of the string, computed recursively

```
>_ ~/workspace/atomic_nature_of_matter

$ python3 reverse.py bolton
notlob
$ python3 reverse.py madam
madam
```

Set n to the length of s

Base case: if $n = 0$, return the empty string

Recursive step: Return a concatenation of $s[n-1]$ and _reverse($s[:n-1]$)

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

```
>_ ~/workspace/atomic_nature_of_matter

$ python3 palindrome.py bolton
False
$ python3 palindrome.py madam
True
```

Set n to the length of s

Base case: if $n = 0$, return `True`

Recursive step: return `True` if the first character in s is the same as the last character *and* _isPalindrome($s[1:n-1]$) is `True`; otherwise, return `False`

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/atomic_nature_of_matter

```
$ python3 password_checker.py Abcde1fg
False
$ python3 password_checker.py Abcde1@g
True
```

Set check_1, check_2, check_3, check_4, and check_5 to `False`

If length of *pwd* is at least 8, set check_1 to `True`

For each character *c* in *pwd*

- If *c* is a digit, set check_2 to `True`
- Else if *c* is upper case, set check_3 to `True`
- Else if *c* is lower case, set check_4 to `True`
- Else if *c* is *not* alphanumeric, set check_5 to `True`

Return the logical *and* of check_1, check_2, check_3, check_4, and check_5

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/atomic_nature_of_matter`

```
$ python3 point.py
p1         = (0, 1)
p2         = (1, 0)
d(p1, p2) = 1.4142135623730951
```

Instance variables

- *x*-coordinate, `_x` (float)
- *y*-coordinate, `_y` (float)

`__init__(self, x, y)`

- Initialize the instance variables to the values of the corresponding parameters

`distanceTo(self, other`

- Return the Euclidean distance between the points *self* and *other*

`__str__(self)`

- Return a string representation of the point *self*

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/atomic_nature_of_matter

```
$ python3 interval.py
a                    = [-2, 1]
b                    = [0, 3]
a.contains(b.lower()) = True
a.contains(b.upper()) = False
a.intersects(b)       = True
```

Instance variables

- Lower bound of the interval, `_lbound` (float)
- Upper bound of the interval, `_rbound` (float)

`__init__(self, lbound, ubound)`

- Initialize the instance variables to the values of the corresponding parameters

`lower(self)`

- Return the value of the instance variable `_lbound`

`upper(self)`

- Return the value of the instance variable `_ubound`

`contains(self, x)`

- Return `True` if the interval *self* contains x and `False` otherwise

`intersects(self, other)`

- Return `True` if the interval *self* intersects the interval *other* and `False` otherwise

`__str__(self)`

- Return a string representation of the interval *self*

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/atomic_nature_of_matter

```
$ python3 rectangle.py
a                 = [-1, 1] x [-1, 1]
b                 = [0, 2] x [0, 3]
area(a)           = 4
perimeter(b)      = 10
a.contains(1, 5)  = False
a.intersects(b)   = True
```

Instance variables

- x-interval of the rectangle, `_xint` (Interval)
- y-interval of the rectangle, `_yint` (Interval)

`__init__(self, xint, yint)`

- Initialize the instance variables to the values of the corresponding parameters

`area(self)`

- Return the area of this rectangle

`perimeter(self)`

- Return the perimeter of this rectangle

`contains(self, x, y)`

- Return `True` if the rectangle *self* contains the point (x, y) and `False` otherwise

`intersects(self, other)`

- Return `True` if the rectangle *self* intersects the rectangle *other* and `False` otherwise

`__str__(self)`

- Return a string representation of the rectangle *self*

Goal: track the motion of particles undergoing Brownian motion, fit this data to Einstein's model, and estimate Avogadro's constant

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/atomic_nature_of_matter

```
$ 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
```

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`

---

[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 $\frac{\bar{x}_{n-1} \cdot (n-1) + x_n}{n}$.

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/atomic_nature_of_matter
```

```
$ 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)
```

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 `luminance()` method from `Color` 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`.

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*)

```
>_ ~/workspace/atomic_nature_of_matter

$ 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
```

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`

Implement a program called `avogadro.py` that accepts the displacements (output of `bead_tracker.py`) from standard input; computes an estimate of Avogadro's constant using the formulae described above; and writes the value to standard output

```
>_ ~/workspace/atomic_nature_of_matter

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

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 Avogadro constant in scientific notation (using the format string `"%e"`)