Project 5 (Atomic Nature of Matter) Checklist

Prologue

Project goal: 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 number

Relevant lecture material

- Recursion
- Using Data Types
- Creating Data Types

Files

- `project5.pdf` (project description)
- `project5.zip` (starter files for the exercises/problems, `report.txt` file for the project report, and `run_tests.py` file to test your solutions)

Exercise 1. (*Sum of Integers*) Implement the functions `sum_iter()` and `sum_rec()` in `sum_of_ints.py` that take an integer n as argument and return the sum $S(n) = 1 + 2 + 3 + \cdots + n$, computed iteratively (using a loop) and recursively. The recurrence equation for the latter implementation is

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

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

Exercises

sum_of_ints.py

```python
import stdio
import sys


# Returns the sum S(n) = 1 + 2 + ... + n, computed iteratively.
def sum_iter(n):
    ...

# Returns the sum S(n) = 1 + 2 + ... + n, computed recursively.
def sum_rec(n):
    ...


# Test client [DO NOT EDIT]. Reads an integer n from command line and
# writes the sum S(n) = 1 + 2 + ... + n, computed both iteratively and
# recursively.
def _main():
    n = int(sys.argv[1])
    stdio.writeln(sum_iter(n))
    stdio.writeln(sum_rec(n))


if __name__ == '__main__':
    _main()
```

Exercise 2. (*Bit Counts*) Implement the functions `zeros()` and `ones()` in `bits.py` that takes a bit string (ie, a string of zeros and ones) s as argument and returns 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* of an empty string is 0 (base case). The *number of ones* in a bit string can be defined analogously.

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

Exercises

bits.py

```
import stdio
import sys


# Return the number of zeros in s, computed recursively.
def zeros(s):
    ...

# Return the number of ones in s, computed recursively.
def ones(s):
    ...


# Test client [DO NOT EDIT]. Reads a string s from command line and writes the
# the number of zeros and ones in s, both computed recursively.
def _main():
    s = sys.argv[1]
    stdio.writef('zeros = %d, ones = %d, total = %d\n',
                 zeros(s), ones(s), len(s))


if __name__ == '__main__':
    _main()
```

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, constructed 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).

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

Exercises

reverse.py

```python
import stdio
import sys


# Returns the reverse of the string s, computed recursively.
def reverse(s):
    ...


# Test client [DO NOT EDIT]. Read a string s from command line and writes its
# reverse, computed recursively.
def _main():
    s = sys.argv[1]
    stdio.writeln(reverse(s))


if __name__ == '__main__':
    _main()
```

Exercise 4. (*Palindrome*) Implement the function `is_palindrome()` 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 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).

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

Exercises

palindrome.py

```
import stdio
import sys


# A recursive function that returns True if s is a palindrome, and False
# otherwise.
def is_palindrome(s):
    ...


# Test client [DO NOT EDIT]. Read a string s from command line and writes
# whether or not s is a palindrome.
def _main():
    s = sys.argv[1]
    stdio.writeln(is_palindrome(s))


if __name__ == '__main__':
    _main()
```

Exercise 5. (*Rational Number*) Define a data type `rational` in `rational.py` that represents a rational number, ie, a number of the form a/b where a and $b \neq 0$ are integers. The data type must support the following API:

method	description
`Rational(x, y)`	a new rational r from the numerator x and denominator y
`r + s`	sum of r and s
`r - s`	difference of r and s
`r * s`	product of r and s
`abs(r)`	absolute value of r
`str(r)`	the string representation of r as `'x/y'`

Use the private function `_gcd()` to ensure that the numerator and denominator never have any common factors. For example, the rational number 2/4 must be represented as 1/2.

```
$ python3 rational.py 100
3.1315929035585515
```

Exercises

rational.py

```python
import stdio
import sys


# Returns the GCD of p and q, computed using Euclid's algorithm.
def _gcd(p, q):
    return p if q == 0 else _gcd(q, p % q)


class Rational:
    """
    Represents a rational number.
    """

    def __init__(self, x, y=1):
        """
        Constructs a new rational given its numerator and denominator.
        """

        d = _gcd(x, y)
        self._x = ...
        self._y = ...

    def __add__(self, other):
        """
        Returns the sum of self and other.
        """

        ...

    def __sub__(self, other):
        """
        Returns the difference of self and other.
        """

        ...
```

Exercises

rational.py

```
    def __mul__(self, other):
        """
        Returns the product of self and other.
        """

        ...

    def __abs__(self):
        """
        Return the absolute value of self.
        """

        ...

    def __str__(self):
        """
        Returns a string representation of self.
        """

        a, b = self._x, self._y
        if a == 0 or b == 1:
            return str(a)
        if b < 0:
            a *= -1
            b *= -1
        return str(a) + '/' + str(b)


# Test client [DO NOT EDIT]. Reads an integer n as command-line argument and
# writes the value of PI computed using Leibniz series:
#   PI/4 = 1 - 1/3 + 1/5 - 1/7 + ... + (-1)^n/(2n-1).
def _main():
    n = int(sys.argv[1])
    total = Rational(0)
    sign = Rational(1)
    for i in range(1, n + 1):
        total += sign * Rational(1, 2 * i - 1)
        sign *= Rational(-1)
```

Exercises

rational.py

```
    stdio.writeln(4.0 * total._x / total._y)


if __name__ == '__main__':
    _main()
```

Problems

Student

The guidelines for the project problems that follow will be of help only if you have read the description of the project and have a general understanding of the problems involved. It is assumed that you have done the reading.

Instructor

Please summarize the project description for the students before you walk them through the rest of this checklist document.

Problems

Problem 1. (*Particle Identification*) Define a data type `Blob` that has the following API:

method	description
`Blob()`	an empty blob b
`b.add(i, j)`	add a pixel (i, j) to the b
`b.mass()`	the number of pixels in b, ie, its mass
`b.distanceTo(c)`	the distance between the centers of b and c
`str(b)`	string representation of b's mass and center of mass

Next, define a data type `BlobFinder` that has the following API:

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

Problems

Hints

- ⇝ `Blob`
  - ⇝ Instance variables
    - ⇝ Number of pixels, `_P` (`int`)
    - ⇝ x-coordinate of center of mass, `_x` (`float`)
    - ⇝ y-coordinate of center of mass, `_y` (`float`)
  - ⇝ `Blob()`
    - ⇝ Initialize the instance variables appropriately
  - ⇝ `b.add(i, j)`
    - ⇝ Use the idea of *running average*[1] to update the x- and y-coordinates of the center of mass of blob `b` to include the new point (`i, j`)
    - ⇝ 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 $\frac{\bar{x}_{n-1} \cdot (n-1) + x_n}{n}$

Problems

- `BlobFinder`
  - 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)`: 1. Create a `Blob` object called `blob`; 2. Call `_findBlob()` with the appropriate arguments; and 3. 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(P)`
    - Return a list of blobs from `_blobs` that have a mass of at least `P`

Problems

Problem 2. (*Particle Tracking*) Implement a client program `bead_tracker.py` that takes an integer P, a float tau, a float $delta$, and a sequence of JPEG filenames as command-line arguments, identifies the beads in each JPEG image using `BlobFinder`, and prints out (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 $delta$).

Hints

- ⇝ Read command-line arguments `P`, `tau`, and `delta`
- ⇝ Construct a `BlobFinder` object for the frame `sys.argv[4]` and from it get a list of beads `prevBeads` that have at least `P` 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 `P` 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`

Problems

Problem 3. (*Data Analysis*) Implement a client program `avogadro.py` that reads in the displacements from standard input and computes an estimate of Boltzmann's constant and Avogadro's number using the formulae described above.

Hints

- 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`
- Initialize `eta`, `rho`, `T`, and `R` to appropriate values
- Estimate Boltzman constant `k` as `6 * math.pi * var * eta * rho / T`
- Estimate Avogadro's number `N_A` as `R / k`
- Write `k` and `N_A` using format string `'%e'` (for scientific notation)

Problems

Be sure to test your programs thoroughly using ten datasets (they are under the `data` directory), 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 subdirectory `run_1` through `run_10`, and the directory also contains some reference solutions.

Epilogue

Use the template file `report.txt` to write your report for the project

Your report must include

- Time (in hours) spent on the project
- Difficulty level (1: very easy; 5: very difficult) of the project
- A short description of how you approached each problem, issues you encountered, and how you resolved those issues
- Acknowledgement of any help you received
- Other comments (what you learned from the project, whether or not you enjoyed working on it, etc.)

Epilogue

Before you submit your files

- Make sure your programs meet the style requirements by running the following command on the terminal

```
$ pycodestyle <program>
```

  where `<program>` is the `.py` file whose style you want to check

- Make sure your programs meet the input and output specifications by running the following command on the terminal

```
$ python3 run_tests.py -v [<items>]
```

  where the optional argument `<items>` lists the exercises/problems (`Exercise1`, `Problem2`, etc.) you want to test, separated by spaces; all the exercises/problems are tested if no argument is given

- Make sure your code is adequately commented, is not sloppy, and meets any project-specific requirements, such as corner cases and running time
- Make sure your report uses the given template, isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling mistakes

Files to submit

1. `sum_of_ints.py`
2. `bits.py`
3. `reverse.py`
4. `palindrome.py`
5. `rational.py`
6. `blob.py`
7. `blob_finder.py`
8. `bead_tracker.py`
9. `avogadro.py`
10. `report.txt`