Homework 4: Objects and Classes

Due March 2, 11:59 pm

Worth 15 points

Instructions on writing and submitting your homework can be found on the course webpage at `http://pages.stat.wisc.edu/~kdlevin/teaching/Spring2022/STAT679/hw_instructions.html`. *Failure to follow these instructions will result in lost points.* Please direct any questions the instructor.

Read this first. A few things to bring to your attention:

1. Start early! If you run into trouble or you have questions, it's best to find those problems well in advance, not in the hours before your assignment is due!
2. If you have clarifying questions or you run into issues, please do not email the instructor directly. Instead, post to the discussion board so that your classmates can benefit as well if they have the same question.
3. **Make sure you back up your work!** I recommend, at a minimum, doing your work in a Dropbox folder or, better yet, using `git`, which is well worth your time and effort to learn.

1 More Fun with Vectors (8 points)

In this exercise, we'll encounter our old friend the vector yet again, this time using objects.

1. Define a class `Vector`. Every vector should have a dimension (a positive integer) and a list or tuple of its entries. The initializer for your class should take the dimension as its first argument and a list or tuple of numbers (ints or floats), representing the vector's entries, as its second argument. If the user supplies only a dimension and no entries, the default behavior should be to create a vector of all zeroes of the given dimension. The initializer should raise a sensible error in the case where the dimension is invalid (i.e., wrong type or a negative number), and should also raise an error in the event that the dimension and the number of supplied entries disagree.
2. Implement a method `Vector.get_dim()` that returns the vector's dimension, and a method `Vector.get_entries()` that returns the vector's entries as a tuple.
3. Did you choose to make the vector's entries a tuple or a list (there is no strictly wrong answer here, although I would say one is better than the other in this context)? Defend your choice.
4. Are the dimension and entries class attributes or instance attributes? Why is this the right design choice?

5. Implement the necessary operator(s) to support comparison (equality, less than, less or equal to, greater than, etc) of `Vector` objects. We will say that two `Vector` objects are equivalent if they have the same coordinates (normally we would worry about float comparison here, but let's ignore that). Otherwise, comparison should be analogous to tuples in Python, so that comparison is done on the first coordinate first, then the second coordinate, then the third, and so on. So, for example, the two-dimensional vector $(2, 4)$ is ordered before (less than) $(2, 5)$. Attempting to compare two vectors of different dimensions should result in an error.

6. Implement the addition and subtraction operators for `Vector` objects, so that we can write `v1 + v2` and `v1 - v2` for `Vector` objects `v1` and `v2` of the same dimension. Adding two `Vector` objects of different dimensions should result in an error, as should adding a Vector and a non-Vector.

7. Implement a method `Vector.dot` for computing the inner product of the caller with another `Vector` object. That is, if `v1` and `v2` are `Vector` objects, `v1.dot(v2)` should compute their inner product. Your method should raise an appropriate error in the event that the argument is not of the correct type or in the event that the dimensions of the two vectors do not agree.

8. We would also like our `Vector` class to support scalar multiplication. Left- or right-multiplication by a scalar, e.g., `2*v` or `v*2`, where `v` is a `Vector` object, should result in a new `Vector` object with its entries all multiplied by the given scalar. This should support multiplication by both ints and floats. We will also follow conventions of `R` and `numpy` (which you will learn in a few weeks), and use `*` to denote entrywise vector-vector multiplication, so that for `Vector` objects `v` and `w`, `v*w` results in a new `Vector` object, with the i-th entry of `v*w` equal to the i-th entry of `v` multiplied by the i-th entry of `w`. Implement the appropriate operators to support this multiplication operation. Many languages have a convention for dealing with multiplication of vectors that differ in their dimension (look up *broadcasting* if you're curious, or just wait a few weeks until we discuss `numpy`), but we will punt on this matter. Your method should simply raise an appropriate error in the event that `v` and `w` disagree in their dimensions. Your method should also raise an error when trying to multiply a Vector object by anything that isn't a Vector, int or float.

9. For a real number $0 \leq p \leq \infty$, and a vector $v \in \mathbb{R}^d$, the p-norm of v, written $\|v\|_p$, is given by

$$\|v\|_p = \begin{cases} \sum_{i=1}^d 1_{v_i \neq 0} & \text{if } p = 0 \\ (\sum_{i=1}^d |v_i|^p)^{1/p} & \text{if } 0 < p < \infty, . \\ \max_{i=1,2,\ldots,d} |v_i| & \text{if } p = \infty \end{cases}$$

   Strictly speaking, this is only a norm for $p \geq 1$, but that's beside the point.[1] Implement a method `Vector.norm` that takes a single argument `p` (an int or float) as an argument and returns the `p`-norm of the calling `Vector` object. Your method should work whether `p` is an integer or float. Your method should raise a sensible error in the event that p is negative or is not of an appropriate type. **Hint:** see `https://docs.python.org/3/library/functions.html#float` for documentation on representing positive infinity in Python.

[1] `https://en.wikipedia.org/wiki/Norm_(mathematics)`

2 Random Walkers (7 points)

In this problem, you'll get some practice working with inheritance by building a collection of objects for generating and representing random walks.[2] Recall that a random walk is a discrete-time stochastic process, in which a "walker" starts out at some position S_0 at time 0, and for all $t = 1, 2, \ldots$, the walker's position at time $t + 1$ is given by $S_{t+1} = S_t + X_t$, where $X_1, X_2, \ldots$ are drawn i.i.d. from some distribution. The variables $X_1, X_2, \ldots$ are called the "steps" or "increments" of the random walk.

The simplest random walk is the symmetric random walk on the integers. The walker starts out at 0 at time $t = 0$, and the steps $X_1, X_2, \ldots$ are independent Rademacher random variables.[3] That is, for all $i = 1, 2, \ldots$, $\Pr[X_i = 1] = \Pr[X_i = -1] = 1/2$. At each time $t = 0, 1, 2, \ldots$, we say that the current position of the random walker is $S_t = \sum_{i=1}^{t} X_i$, so that $S_0 = 0$, $S_1 = X_1$, $S_2 = X_1 + X_2$, and so on. At any given time t, we can speak of the "history" or the "sample path" of the random walker, $(S_0, S_1, \ldots, S_{t-1})$. That is, the sequence of integers that the random walker has visited prior to time t.

1. Implement a class representing symmetric random walk on the integers, called `SymIntRW`, which has the following instance attributes:

   - `current_position`: an integer, the current position of the random walker.
   - `history`: a list of integers that specify the previous positions of the walker, in order (so that the last entry of the list is the most recent previous position of the walker).

   The class should support the following methods:

   - `get_time()`: return a non-negative integer encoding the number of steps that the walker has taken so far. That is, return the length of the `history` attribute.
   - `get_history()`: return the `history` attribute, i.e., the list of previous positions.
   - `get_position()`: return the `current_position` attribute, i.e., the integer that the walker is currently at.
   - `step()`: take a single step. Generate a random step according to a Rademacher distribution. Update the current position and history accordingly. **Hint:** use the Python `random.randint` function.[4]

   You should also implement an initialization method that takes no arguments (aside from the `self` argument) and initializes `self.current_position` to 0 and `self.history` to be the empty list. This reflects the fact that initially, the random walker starts at $S_0 = 0$, at which point it has no "history".

2. One natural extension of the symmetric random walk on the integers is to allow the steps to be drawn from a different distribution. Implement a class `IntegerRW` that inherits from `SymIntRW`, but has the additional attribute `p`, where `p` is a probability. Override the following methods:

[2] `https://en.wikipedia.org/wiki/Random_walk`
[3] `https://en.wikipedia.org/wiki/Rademacher_distribution`
[4] `https://docs.python.org/3/library/random.html#random.randint`

- Override the initialization method to take a single optional argument `p`, which should default to 0.5, which initializes the `self.p` attribute to be equal to the argument `p`, and raises an appropriate error in the event that `p` is not a probability (i.e., is not an int or float in the range $[0, 1]$).
- Override the `step` method so that the steps are generated according to

$$\Pr[X_i = 1] = 1 - \Pr[X_i = -1] = \texttt{p}.$$

  Hint: `random.random()` will generate a random float drawn uniformly from the interval $[0, 1)$.

3. Another extension of the symmetric random walk on the integers is to define the random walk on 2-, 3- or higher-dimensional space. The d-dimensional simple symmetric random walk (SSRW) is a random walk on the integer lattice

$$\mathbb{Z}^d = \{(z_1, z_2, \ldots, z_d) : z_1, z_2, \ldots, z_d \in \mathbb{Z}\},$$

in which the steps $X_1, X_2, \cdots \in \mathbb{Z}^d$ are drawn i.i.d. from the uniform distribution on

$$\left\{z \in \mathbb{Z}^d : \|z\| = 1\right\}.$$

Said another way, to generate a step X_t, we generate $X_t = \epsilon_t Z_t$, where $\epsilon_t \in \{-1, 1\}$ is a Rademacher random variable,

$$\Pr[\epsilon_t = 1] = \Pr[\epsilon_t = -1] = \frac{1}{2},$$

and Z_t is generated by choosing an index j uniformly from the set of integers $\{1, 2, \ldots, d\}$, and setting

$$(Z_t)_i = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

This process is called the symmetric random walk because the steps are chosen uniformly from the points adjacent to the origin in $\mathbb{Z}^d$ (i.e., so that the step distribution is symmetric about the origin).

Implement a class `SSRW` that inherits from `SymIntRW`, but represents a random walk on the d-dimensional lattice. Thus, the attribute `self.current_position` is now a point in $\mathbb{Z}^d$, which you should represent by a list of d integers (note that we could use our `Vector` class from the previous problem, but I want to avoid making one problem depend on the other). The class should have an attribute `self.dimension`, which is a positive integer representing the dimension of the random walk. You will need to override the following methods:

- `step()`: Update this method to generate a new step according to the process outlined above. **Hint:** use `random.choice` to select from $\{-1, 1\}$.
- The initialization method should take a single argument, a positive integer `d`, which specifies the `dimension` attribute. `d` should default to 1. The initialization method should raise an appropriate error in the event that this argument is not a positive integer. In keeping with the one-dimensional case, the current position should be initialized to be the origin.