Weighted Random Distribution

Ferry Boender

Revision History

Revision 1.0 Dec 23 2009 Revised by: FB

1. Preface

Randomly selecting elements from a set of items is easy. Just generate a random number between 0 and the length of the set minus one, and use that as an index in the set (if it is an array) to get a random entry. The chance that an entry is picked is the same for each entry in the set. This is called *even distribution* or *uniform distribution*.

But what if we do not want each entry to appear as much as every other? Suppose we're creating a question-answer game, and we want the questions the user got wrong previously to appear more often than the question he or she got right? This is called a *Weighted Random Distribution*, or sometimes *Weighted Random Choice*, and there are multiple methods of implementing such as random picker.

This article explains these various methods of implementing Weighted Random Distribution along with their pros and cons. We use Python as our language of choice, because it has an easy to read syntax, and provides many useful tools which would take many more lines of code in most other languages. Along the way all Python "tricks" will be explained.

2. Methods

In each of the methods below, we assume a default set of choices and weights which we'll need to make random selections from. When I say "set", I don't mean a Python `set`, which is basically an immutable array which does not allow duplicate values, but a collection of choices with their associated weight.

We assume we have the following dictionary (Pythonese for an associative array, also known as a hashmap) of choices with their weights:

```
weights = {
        'A': 2,
        'B': 4,
        'C': 3,
        'D': 1
}
```

This means we want `B` to appear twice as much as `A`, and we want `A` to appear twice as much as `D`. In other words, if we generate ten random picks, we want two of those to be `A`, four of those to be `B`, etc. (This wont happen with only ten random picks of course, but you get the point).

2.1. Expanding

One of the easiest solutions is to simply expand our set so that each entry in it appears as many times as its weight. We start with our basic set of choices with their associated weights:

```
weights = {
        'A': 2,
        'B': 4,
        'C': 3,
        'D': 1
}
```

We create a new list (Pythonese for an array), in which we put each choice as many times as its associated weight.

```
dist = []
for x in weights.keys():
        dist += weights[x] * x
```

The list `dist` now looks like this:

```
['A', 'A', 'B', 'B', 'B', 'B', 'C', 'C', 'C', 'D']
```

We can now make a random selection from the list by generating a random number between 0 and the length of the list, and use that as an index in the list to get our weighted random choice:

```
results = {}
for i in range(100000):
        wRndChoice = random.choice(dist)
        results[wRndChoice] = results.get(wRndChoice, 0) + 1

print results
```

Python provides a module `random` which has a function `choice` which will select a random choice from the list for us. We keep a score of how many times each choice has appeared in `results`. The output of this is a dictionary which contains how many times each of the choices from the list has been randomly selected:

```
{'A': 20140, 'C': 29880, 'B': 39986, 'D': 9994}
```

As we can see, our random distribution is weighted properly. Of course there are some pros and cons to using this method.

Pros

- We can make very fast random selections from the set. In fact, selection time is `O(1)`, which is the fastest we can attain.
- It allows for reasonably easy and fast updating of the weights. If we want to lower the weight of a choice, we simply scan the list and remove as many occurrences of the choice as we need. Increasing weight or adding new choices is even simpler, because we can just add as many as we need to the end of the list. The order in which they appear does not matter.
- It's very simple. It's easy to see what's going on here.

Cons

- For large sets, or large values for weights, this will obviously take up too much memory. One possible way to optimize this would be to find the greatest common divisor, but this will take more processing time up front and will make it much slower to update our weights.
- If you only need to make a few random selections, it will probably not be worth preparing the expanded set.

I've looked around the Internet for various Weighted Random Distribution algorithms, and nobody seems to be suggesting this method in particular. I can understand why; it looks amateurish. But from my tests I have found that this is the easiest and fastest way to make Weighted Random Distributions for small sets and small values.

The full example:

```
#!/usr/bin/python

import random

weights = {
        'A': 2,
        'B': 4,
        'C': 3,
        'D': 1
}

dist = []
for x in weights.keys():
        dist += weights[x] * x

results = {}
for i in range(100000):
        wRndChoice = random.choice(dist)
        results[wRndChoice] = results.get(wRndChoice, 0) + 1
```

```
print results
```

2.2. In-place (Unsorted)

Instead of expanding the set as in the method used above, we can also keep the set in its current form and simply emulate the expansion of the set in a loop. In order to do this, we first have to know the total weight of the set.

```
wTotal = sum(weights.values())
```

We then select a random value between 0 and that total weight - 1. We emulate the expanded set we've seen in the previous method by looping over the elements of the set and keeping score of the total value we've seen so far. When that value is larger than the random value we picked, we've found our random choice.

```
wRndNr = random.randint(0, wTotal - 1)
s = 0
for w in weights.items():
        s += w[1]
        if s > wRndNr:
                break;
```

`w` is now our weighted random choice.

Pros

- It is very easy and fast to update our set of weights. Adding and removing items; lowering and heiring weights: all are equally fast. All we have to do is keep an eye on our total weight and either update or recalculate it when we add or remove values or change weights.
- This method uses as little memory as possible. No duplicates of our original set have to be made.

Cons

- Selecting random values is somewhat slower because of the added calculation in the loop. The larger our initial set, the slower this becomes. The complexity of selecting is O(n), where `n` is the number of elements in the set.

The full example:

```
#!/usr/bin/python
```

```
import random

weights = {
        'A': 2,
        'B': 4,
        'C': 3,
        'D': 1
}

wTotal = sum(weights.values())

results = {}
for i in range(100000):
        wRndNr = random.randint(0, wTotal - 1)
        s = 0
        for w in weights.items():
                s += w[1]
                if s > wRndNr:
                        break;
        results[w[0]] = results.get(w[0], 0) + 1

print results
```

2.3. In-place (sorted)

In theory, we can speed up our previous in-place algorithm by sorting the set before we start selecting from it. Since the set is sorted, we can start scanning through it in reversed order, starting at the end. Since the highest weights will appear at the end of the set, and those are the most likely to be chosen randomly, we can get a speed increase when selecting random numbers from our set. Whether we get a speed increase in practice depends on our initial set of weights.

First we prepare a new set which contains our sorted weigths.

```
weights = {
        'A': 2,
        'B': 4,
        'C': 3,
        'D': 1
}

sWeights = sorted(weights.items(), lambda x, y:cmp(x[1], y[1]))
wTotal = sum(weights.values())
```

You'll see a little bit of Python magic right here, so let me explain. The `sorted` function is a Python built-in which returns a sorted duplicate of the first argument you pass it. We pass it `weights.items()` which looks like `( ('A', 2), ('B', 4), ('C', 3), ('D', 1) )`. As the second parameter we pass an anonymous (nameless) function (a *lambda*) that takes two parameters: `x` and `y`. The anonymous

function compares the second value in both arguments and returns whether `x` is smaller than, equal to or larger than `y`. The `sorted` function uses that function to determine how to sort the first parameter.

`sWeights` now looks like this:

```
[('D', 1), ('A', 2), ('C', 3), ('B', 4)]
```

Then we walk through the set backwards. Instead of adding up the weights we've seen so far (like we did in the unsorted in-place example), we subtract the weights from the total weight:

```
results = {}
for i in range(100000):
        wRndNr = random.randint(0, wTotal - 1)
        s = wTotal
        for i in xrange(len(sWeights) - 1, -1, -1):
                wRndChoice = sWeights[i]
                s -= wRndChoice[1]
                if s <= wRndNr:
                        break
        results[wRndChoice[0]] = results.get(wRndChoice[0], 0) + 1
print results
```

Pros

- Increase in selection speed over unsorted, as long as our set has at least one weight which is significantly larger than the other weights.

Cons

- Speed decrease in pre-selection (due to sorting time).
- Speed decrease in updating the set of weights (due to resorting).
- Increase in complexity may not be worth the speed gain.

The full example:

```
#!/usr/bin/python

import random

weights = {
        'A': 2,
        'B': 4,
        'C': 3,
        'D': 1
```

```
}

sWeights = sorted(weights.items(), lambda x, y:cmp(x[1], y[1]))
wTotal = sum(weights.values())

print sWeights

results = {}
for i in range(100000):
        wRndNr = random.randint(0, wTotal - 1)
        s = wTotal
        for i in xrange(len(sWeights) - 1, -1, -1):
                wRndChoice = sWeights[i]
                s -= wRndChoice[1]
                if s <= wRndNr:
                        break
        results[wRndChoice[0]] = results.get(wRndChoice[0], 0) + 1
print results
```

2.4. Pre-calculated

One other possibility remains for performing random weighted selections. We can prepare a new set with pre-calculated weights. So instead of calculating the weights in the loops, we already calculate them beforehand. This will give a performance increase in selecting random values because we don't have to subtract or add values in the loop. One other important optimization this allows us to make is that we can apply a divide-and-conquer algorithm (also known as bisecting) to quickly find the random choice. The drawbacks are that updating the weights becomes much harder.

First we create a new set in which the weights have already been added to each other. In all the other examples we created a set with `(choice, weight)` members. Now we create `(weight, choice)` members, so that we can use Python's `bisect` module.

The bisect module implements a divide-and-conquer algorithm to find the index in the set where a new value should be inserted. It does this by starting in the middle of the set and checks if the value there is higher or lower than the value we need. It then divides the left-hand or right-hand side again and does the same, until it has found the correct index. For this to work, the set must be sorted. In our case, this isn't a problem, since we sum up our weights the further we get into the set, so the weights will always be sorted.

```
wTotal = 0
sWeights = []
for w in weights.items():
        wTotal += w[1]
        sWeights.append( (wTotal, w[0]) )
```

The set now looks like this:

```
[(2, 'A'), (5, 'C'), (9, 'B'), (10, 'D')]
```

Next we get random weighted selections from the set by picking a random number between 0 and the total weight. We create a temporary fake member (`(wRndNr, None)`) so we can pass it to the bisect module. The bisect function finds our index for us, and we use that index to get the member of our set.

```
results = {}
for i in range(100000):
        wRndNr = random.randint(0, wTotal - 1)
        wRndChoice = sWeights[bisect.bisect(sWeights, (wRndNr, None))]

        results[wRndChoice[1]] = results.get(wRndChoice[1], 0) + 1
```

Pros

- Fast selection, even on large sets, due to the bisection.

Cons

- Slow updates of weights due to having to scan through the list for the value we need to update, then have to update all the items in the set after it.

The complete example:

```
#!/usr/bin/python

import random
import bisect

weights = {
        'A': 2,
        'B': 4,
        'C': 3,
        'D': 1
}

wTotal = 0
sWeights = []
for w in weights.items():
        wTotal += w[1]
        sWeights.append( (wTotal, w[0]) )

results = {}
for i in range(100000):
        wRndNr = random.randint(0, wTotal - 1)
```

```
        wRndChoice = sWeights[bisect.bisect(sWeights, (wRndNr, None))]

        results[wRndChoice[1]] = results.get(wRndChoice[1], 0) + 1

print results
```

3. Conclusion

As you can see there are many different ways of performing random weighted selections. Each has it's own pros and cons. Here's a small summary:

If you need **speedy selections**

- Use expanded if the number of items in your set is low, and your weights are not very high.
- Use pre-calculated if your set has lots of numbers and/or your weights are high.

If you need **speedy updates**

- Use in-place (unsorted) for the fastest updates.
- Use in-place (sorted) for somewhat slower updates and somewhat faster selections.

If you need **low memory usage**

- **Don't** use expanded.

If you need **simplicity**

- Use expanded or in-place (unsorted).

4. Copyright and license

Author:

```
Ferry Boender
<ferry (DOT) boender (AT) electricmonk (DOT) nl>
```