MODULE 02 - 061: Python Exercise: Building a Weighted Lottery function

2025-02-16 15:25:45 +0100

MODULE 02-061:P ython Exercise:

Building a Weighted Lottery Function

Introduction

A **weighted lottery function** is a system that randomly selects an item from a set, but with different probabilities based on predefined weights.

This guide will break down how to create a weighted lottery function in Python, using:

- **Dictionaries** to store weights.
- **Iteration** to process weights efficiently.
- **NumPy's `random.choice`** to select items based on their probabilities.

Python Docs: random.choice

Problem Statement

Input:

A dictionary where:

- **Keys** are the possible outcomes.
- **Values** are their respective weights (higher weight = higher probability of selection).

Example:

```
weights = {
    'winning': 1,
    'losing': 1000
}
```

Here, `winning` should be returned **1 in 1001 times**, while `losing` should be returned **1000 in 1001 times**.

Another possible set:

```
other_weights = {
    'winning': 1,
    'break_even': 2,
    'losing': 3
}
```

In this case, the function should return:

- `winning` **1/6** times,
- `break_even` **2/6** times,
- `losing` **3/6** times.

Output:

A randomly selected **string** representing one of the dictionary keys, chosen according to the weight distribution.

Step-by-Step Solution

1 Creating a Weighted Container List (Inefficient Approach)

The simplest way to implement this is by creating a list where each item appears as many times as its weight.

```
import numpy as np

def weighted_lottery(weights):
    container_list = []

    for (name, weight) in weights.items():
        for _ in range(weight):
            container_list.append(name)

    return np.random.choice(container_list)

weights = {'winning': 1, 'losing': 1000}
print(weighted_lottery(weights))
```

This method **works** but is highly inefficient for large datasets. Time complexity: **O(n)**, where `n` is the total sum of weights. **Memory Usage:** Creating a massive list for large weights (e.g., 100,000 entries) is impractical.

---

2 Optimized Approach: Using NumPy Probability Distribution

Instead of expanding weights into a list, we directly use **probability distributions**.

Explanation:

1. Extract the dictionary keys (`keys` list) and corresponding weights (`values` array).
2. Convert `values` to a **normalized probability distribution** (each weight divided by total weight sum).
3. Use `numpy.random.choice()` with `p` (probability distribution) instead of repeating values in a list.

```
import numpy as np

def weighted_lottery(weights):
    keys = list(weights.keys())
    values = np.array(list(weights.values()), dtype=np.float64)

    return np.random.choice(keys, p=values / values.sum())

other_weights = {
    'winning': 1,
    'break_even': 2,
    'losing': 3
}

print(weighted_lottery(other_weights))
```

Advantages of this method: **Memory efficient** – No need for a giant list. **Faster execution** – `numpy.random.choice()` is optimized for large datasets. **Scalability** – Handles thousands of items easily.

Alternative Use Cases

This function can be used in various applications:

large

Figure 1: large

Weighted Random Color Picker

```
colors = {
    'red': 1,
    'green': 2,
    'blue': 3,
    'yellow': 4,
    'purple': 5
}
print(weighted_lottery(colors))
```

Higher-weighted colors (like `purple`) will appear more frequently.

Lottery Simulation with 100,000 Entries

```
lottery_entries = {f"Ticket-{i:05d}": max(1, i) for i in range(100000)}
print(weighted_lottery(lottery_entries))
```

Scales to **100,000** entries effortlessly, something impossible with the naive approach.

Summary & Best Practices

Use NumPy's `random.choice()` with probability distribution (`p` parameter) for optimal performance. **Avoid list expansion for large datasets** – it's inefficient in memory and speed. **Ensure your weights are positive** – negative or zero values will break the probability calculation. **Test with different scenarios** to confirm correct probability distributions.

Video lesson Speech

In this Python coding exercise, we are tasked with a pretty challenging project.

We are going to build out a weighted lottery function.

What I mean by a weighted lottery function is, imagine that you are working on some kind of gaming system, and you are tasked with making sure that the house wins.

Kind of a, not a predetermined amount because the system has to be random, but it wins a majority of the time.

If I have a dictionary like this: I have a dictionary called `weights`, and just as a little spoiler alert this is going to be what I want you to pass into the function.

So the input for the function you'll build out should take a dictionary as an argument.

We might want to have something like this, where we have winning as 1 of the keys, and the value there just for the sake of argument is `1`. Then we might want to have losing and have the value of that being a `1000`. What this means is that our function should return if it runs through a `1001` times.

From a probability perspective, it should return winning 1 of those times and then losing 1000 times. So this is all kind of theoretical in this sense. I don't think you'd ever be tasked with building out a system with this poor of odds, but this is just so that you have an idea for what gets built out.

If you want an idea on how to test this if you wanted to run your program, so if you have these `weights` here then we're going to call the function `weighted_lottery(weights)`, and then call `weights` right here. If we were to print this out and we're to run the program 1 time, we should almost be guaranteed that losing is the result that would get returned.

If we ran it through 1000 times then winning should come up 1 of those times theoretically. That's really all this system should do, but we need to be a little bit more scalable than this. If we're just tasked with building something like this out, it wouldn't be too difficult.

I want to add something in because rarely are you ever going to be working with a program that's hardcoded like this. Let's copy this, and let's look at another set of weights. We also should have the ability to have something like this where I say winning is 1 time, and then we'll say that they can break even and we'll just say this should be 2 times. Then losing could be 3.

```
weights = {
 'winning': 1,
 'losing': 1000
 }
weighted_lottery(weights)
other_weights = {
 'winning': 1,
 'break_even': 2,
 'losing': 3
 }
weighted_lottery(other_weights)
```

Now, this is going to give much higher odds of winning and break_even. You should either win or break_even half the time, and then you should lose the other half of the time. This is another set of `other_weights`. If you called it just like this the program should still work.

Do not limit your function to only being able to work with two sets of key-value pairs. It should work with 1, 2, or even 1000. Now obviously, if you work with 1, that's not going to be very helpful because the weights wouldn't really matter. It would be 100% of the time but just think of this as being a dynamic type of system.

Just to review what the input is: is a dictionary. What the output is: should be a string that should return in this case winning, break_even, or losing, and it should do it in the right proportion. I want you to pause the video right here, then go try to build this yourself, and then come back and watch my own solution.

I hope you had a good time building that out. Let's go through and see how I personally would build out a system like this. The first thing I would do is I'd probably import the `numpy` library. So I'd say `import numpy as np`, and we'll see why in a little bit.

Let's create the function, so I'm going to say `def weighted_library`. It's gonna take in the `weights` as the argument, and let's just give us a little bit of room. Now that we have this, what I first want to do is create a container list, so I need something to keep track of all of the weights. So I'm going to say: `container_list = []`.

```
import numpy as np
weighted_lottery(weights):
 container_list = []
```

Now what I need to do is I need to have some type of looping mechanism where I can loop through all of the weights, count them up, and then I can keep track of them. I can keep track of them in that `container_list`, but I'm going to need to create two loops.

I'm going to first have to loop through the key-value pairs. If you take this example, you have to loop through weights twice. For this example, I'm going to loop through it three times. If I have an example with 1000 weights, I need to loop through that 1000 times.

That's going to be my first loop so I'll say something like 4, and because this is a dictionary I need to get the key and the value. So I can say:

```
import numpy as np
weighted_lottery(weights):
 container_list = []
 for (name, weight)
```

Whenever you're looping through a dictionary, and you need access to the key and the value: you can wrap them in parens. You can wrap of these variables, so these iteration variables, in parens and say: `for (name, weight) in weights.items()`.

That's a function, so what that's going to do is it's going to give us the ability to loop through each 1 of those key-value pairs and give us access to the key which I'm assigning to name and then to the weight which I'm assigning to the weight variable.

Now that I have that what I want to do is start keeping track of how many items are there. I know that, for this example, let's get rid of a few of these empty lines, I know that I want to loop over winning 1 time. It only has 1 item, and then for losing I'm going to go through that 1000 times.

Now technically there are a few different ways to do this. I think this might be 1 of the most readable. Now, because we're creating two nested loops, if you're working with say millions of items then you'd have to come up with a different solution.

Because we know that our `N`, meaning the number of times that we're going to run through this is relatively small, then I can create a nested loop here. So I can say:

```
import numpy as np
weighted_lottery(weights):
 container_list = []
 for (name, weight) in weights.items():
 for _
```

Now, this isn't required. There's nothing special about `underscore`, but what it does is it'll tell myself or anyone else who's reading the code: whenever you use underscore as a variable name it usually is a sign that this is a variable that is not going to be used. We don't really need it. I'm going to show you why.

What I'm using this inner loop for, is simply a counter. So I can say `for _ in range(weight)`. If you want to translate what this looks like, let's add it as a comment up here. Let's come up, just so it's a little easier to read. If I say this I'm saying: for whatever it is in the list, and then I want to do this:

```
import numpy as np
weighted_lottery(weights):
 container_list = []
 for (name, weight) in weights.items():
 # loop 1 time for win/loop 1000 times for losing
 for _ in range(weight):
```

This is a way that you can do this in Python. When I say range, and pass in an integer, which could be 1, 1000, or in our other example, it could be 1, 2, and 3. It's going to loop through that many times. Now that we have that, we only have a few more lines of code that we need to implement. So here I can say:

```
import numpy as np
weighted_lottery(weights):
 container_list = []
 for (name, weight) in weights.items():
 for _ in range(weight):
 container_list.append(name)
```

What this is doing is it's building up a container list for us. What that container is going to look like, and we can run it through to see what it's actually going to look like. When the weights are like they are in this example, we're going to have the word `winning`, and then the word `losing.` We're going to have winning 1 time and then we're going to have a losing a 1000 times.

Let's just make sure that our system is working up to this point, so I'll say:

```
weighted_lottery(weights):
 container_list = []
 for (name, weight) in weights.items():
 for _ in range(weight):
 container_list.append(name)
 return container_list
```

Let's come down, and I'm going to comment this one out, just because for this specific case, I don't really want to print out a list of 1000 items. My little comment function isn't working here, so I will just delete it. We'll go with this example because it's a little bit easier to read, and it's easier to show on the screen as well.

Let's just print this value out. So `print(weighted_lottery(other weights))`. Now if I want to run this, I'm going to say `python weighted_lottery.py`, and there you go.

large

Figure 2: large

large

Figure 3: large

You can see that this returns a list it has `winning` 1 time, which is what we want because you can see we have winning right here. It's only showing up 1 time and so it should only be added to the list 1 time. Then it has `break_even` twice which is good because our weight there is two, and then it has `losing` three times. So we're really close.

Now, the example that I want you to think of because this is what I use when I was building out the system is: imagine that you're playing Scrabble or something and you have this bag. Inside of the bag is the set of words in this list.

If you have a little card that says `winning` you'd put 1 winning card in the bag. Then you'd have 2 `break_even` cards, then you'd have 3 `losing` cards, and these all go in the bag. What we need to do is to add a method that allows us to go in and grab a random sample from that bag.

We need the ability to randomly go into it and pick out 1 of these elements. That's how we have a randomized system. We're not hard coding in which one gets put in, but because we have more losing in this specific example, this means that losing is weighted much more than the other ones. We have a 50 percent chance of picking out `losing` out of this theoretical bag.

What we can do, and hopefully you went to when you were building this out and researched some of the various ways to do that, you can write it all in regular Python. Personally, whenever I build something like this out I usually turn to a library like `numpy` for it because it's very good. It's incredibly efficient from a performance perspective, it's also easier to call and to remember.

What I can do is say `np`, so I'm calling `numpy`, so `np.random.choice(container_list)`. That is a method in the random library for numpy. What that's going to do is `choice` reaches into a list and it pulls out a random sample.

Let's save this, and now let's run this again. I'm going to run this, and you can see it pulls out winning. Run it again, losing. Running again, break_even. Again, losing, break_even, and losing. If you were to run this 1000 times, losing should be there half the time or approximately around half the time. You should either win or break_even, with breaking_even being about twice as much as winning.

Great job if you went through that, you now know how to work with randomization inside of Python, and also to be able to manage that random process, by being able to work with iteration and with data structures. So a very good job.

Code

```
# 02-062: Python Exercise: Building a Weighted Lottery function
"""
import numpy as np

# This freezes the system:
# DEBUG: It creates almost infinite numbers.
def weighted_lottery(weights):
    container_list = []

    for (name, weight) in weights.items():
        for _ in range(weight):
            container_list.append(name)

    return np.random.choice(container_list)



other_weights = {
    # minimum 5digits
```

```
    f"{number:05d}": max(1, number) for number in range(100000)
}
print(weighted_lottery(other_weights))
"""



# Attempt 2

import numpy as np

def weighted_lottery(weights):

    keys = list(weights.keys())
    values = np.array( list(weights.values()), dtype=np.float64 )

    return np.random.choice( keys, p=values / values.sum() )

other_weights = {
    f"{number:05d}": max(1, number) for number in range(100000)
}

print('El número agraciado del sorteo de hoy es:  ' + str(weighted_lottery(other_weights)))


"""
# Whith colours
other_weights = {
    'red': 1,
    'green': 2,
    'blue': 3,
    'yellow': 4,
    'purple': 5,
    'orange': 6,
    'pink': 7,
    'brown': 8,
    'black': 9,
    'white': 10,
    'gray': 11,
    'cyan': 12,
    'magenta': 13,
    'lime': 14,
    'teal': 15,
    'indigo': 16,
    'violet': 17,
    'gold': 18,
    'silver': 19,
    'bronze': 20,
}
"""
```

Resources

- Source Code