Generate random numbers with a given (numerical) distribution

Ask Question

I have a file with some probabilities for different values e.g.:

```
1 0.1
2 0.05
3 0.05
4 0.2
5 0.4
6 0.2
```

I would like to generate random numbers using this distribution. Does an existing module that handles this exist? It's fairly simple to code on your own (build the cumulative density function, generate a random value [0,1] and pick the corresponding value) but it seems like this should be a common problem and probably someone has created a function/module for it.

I need this because I want to generate a list of birthdays (which do not follow any distribution in the standard `random` module).

python module random

asked Nov 24 '10 at 10:56

pafcu
2,827 7 31 49

2 Other than `random.choice()` ? You build the master list with the proper number of occurrences and choose

one. This is a duplicate question, of

2 @S.Lott: Your choice method would probably be fine for small numbers of occurrences but I'd rather avoid creating huge lists when it is not necessary. – pafcu Nov 24 '10 at 11:10

1 @Lucasmus: define "big". @pafcu: define "huge". This will work delightfully well until you fill up all of memory with the distribution table. Will you really have billions of choices? Really? For any practical simulation, a few thousand values in a choice table essentially nothing. – S.Lott Nov 24 '10 at 11:18

5 @S.Lott: OK, about 10000*365 = 3650000 = 3.6 million elements. I'm not sure about the memory usage in Python, but it's at least 3.6M*4B =14.4MB. Not a huge amount, but not something you should ignore either when there is an equally simple method that does not require the extra memory. – pafcu Nov 24 '10 at 11:25

12 Answers

`scipy.stats.rv_discrete` might be what you want. You can supply your probabilities via the `values` parameter. You can then use the `rvs()` method of the distribution object to generate random numbers.

As pointed out by Eugene Pakhomov in the comments, you can also pass a `p` keyword parameter to `numpy.random.choice()`, e.g.

```
numpy.random.choice(numpy.arange(1
```

If you are using Python 3.6 or above, you can use `random.choices()` from the standard library – see the answer by Mark Dickinson.

edited Apr 4 at 20:06

Mark Ransom
214k 28 261 482

answered Nov 24 '10 at 12:15

Sven Marnach
316k 70 716 674

4 On my machine

`numpy.random.choice()` is almost 20 times faster. – Eugene Pakhomov Jun 18 '16 at 6:26

7 it does exactly the same w.r.t. to the original question. E.g.: `numpy.random.choice(numpy.arange(1, 7), p=[0.1, 0.05, 0.05, 0.2, 0.4, 0.2])` – Eugene Pakhomov Jun 20 '16 at 12:17

1 @EugenePakhomov That's nice, I didn't know that. I can see there is an answer mentioning this further, but it doesn't contain any example code and hasn't a lot of upvotes. I'll add a comment to this answer for better visibility. – Sven Marnach Jun 20 '16 at 15:40

2 Surprisingly, rv_discrete.rvs() works in O(len(p) * size) time and memory! While choice() seems to run in optimal O(len(p) + log(len(p)) * size) time. – alyaxey Oct 9 '17 at 16:16

1 If you're using **Python 3.6** or newer there's **another answer** that doesn't require any addon packages. – Mark Ransom Apr 4 at 18:47

An advantage to generating the list using CDF is that you can use binary search. While you need O(n) time and space for preprocessing, you can get k numbers in O(k log n). Since normal Python lists are inefficient, you can use `array` module.

If you insist on constant space, you can do the following; O(n) time, O(1) space.

```
def random_distr(l):
    r = random.uniform(0, 1)
    s = 0
    for item, prob in l:
        s += prob
        if s >= r:
            return item
    return item  # Might occur bec
```

edited Dec 29 '15 at 12:17

answered Nov 24 '10 at 12:06

sdcvvc

21.1k 4 56 93

The order of the (item, prob) pairs in the list matters in your implementation, right? – stackoverflowuser2010 Jun 6 '13 at 22:37

1 @stackoverflowuser2010: It shouldn't matter (modulo errors in floating point) – sdcvvc Jun 7 '13 at 12:52

Nice. I found this to be 30% faster than scipy.stats.rv_discrete. – Aspen May 3 '15 at 3:07

1 Quite a few times this function will throw a KeyError because the last line. – Drunken Master Sep 9 '15 at 20:02

1 @Vaibhav: thanks, fixed – sdcvvc Dec 29 '15 at 12:17

Since Python 3.6, there's a solution for this in Python's standard library, namely `random.choices`.

Example usage: let's set up a population and weights matching those in the OP's question:

```
>>> from random import choices
>>> population = [1, 2, 3, 4, 5, 6
>>> weights = [0.1, 0.05, 0.05, 0.
```

Now `choices(population, weights)` generates a single sample:

```
>>> choices(population, weights)
4
```

The optional keyword-only argument `k` allows one to request more than one sample at once. This is valuable because there's some preparatory work that `random.choices` has to do every time it's called, prior to generating any samples; by generating many samples at once, we only have to do that preparatory work once. Here we generate a million samples, and use `collections.Counter` to check that the distribution we get roughly matches the weights we gave.

```
>>> million_samples = choices(popu
>>> from collections import Counte
>>> Counter(million_samples)
Counter({5: 399616, 6: 200387, 4: 
```

answered Jan 25 '17 at 12:59

Mark Dickinson

16.7k 5 47 74

3 This needs more upvotes. – Ramazan Polat Mar 7 at 22:16

Maybe it is kind of late. But you can use `numpy.random.choice()` , passing the `p` parameter:

```
val = numpy.random.choice(numpy.ar
```

edited Nov 14 '17 at 20:49

Engineero

4,833 3 21 45

answered Dec 1 '13 at 0:59

Ramon Martinez

2,835 1 22 28

1 The OP doesn't want to use `random.choice()` - see the comments. – pobrelkey Dec 1 '13 at 1:17

4 `numpy.random.choice()` is completely different from `random.choice()` and supports probability distribution. – Eugene Pakhomov Jun 18 '16 at 6:25

(OK, I know you are asking for shrink-wrap, but maybe those home-grown solutions just weren't succinct enough for your liking. :-)

```
pdf = [(1, 0.1), (2, 0.05), (3, 0.
cdf = [(i, sum(p for j,p in pdf if
R = max(i for r in [random.random(
```

I pseudo-confirmed that this works by eyeballing the output of this expression:

```
sorted(max(i for r in [random.rand
       for _ in range(1000))
```

answered Nov 24 '10 at 11:32

Marcelo Cantos
138k 30 273 323

This looks impressive. Just to put things in context, here are the results from 3 consecutive executions of the above code: ['Count of 1 with prob: 0.1 is: 113', 'Count of 2 with prob: 0.05 is: 55', 'Count of 3 with prob: 0.05 is: 50', 'Count of 4 with prob: 0.2 is: 201', 'Count of 5 with prob: 0.4 is: 388', 'Count of 6 with prob: 0.2 is: 193'].....['Count of 1 with prob: 0.1 is: 77', 'Count of 2 with prob: 0.05 is: 60', 'Count of 3 with prob: 0.05 is: 51', 'Count of 4 with prob: 0.2 is: 193', 'Count of 5 with prob: 0.4 is: 438', 'Count of 6 with prob: 0.2 is: 181'] ..... and – Vaibhav Dec 29 '15 at 10:10

['Count of 1 with prob: 0.1 is: 84', 'Count of 2 with prob: 0.05 is: 52', 'Count of 3 with prob: 0.05 is: 53', 'Count of 4 with prob: 0.2 is: 210', 'Count of 5 with prob: 0.4 is: 405', 'Count of 6 with prob: 0.2 is: 196'] – Vaibhav Dec 29 '15 at 10:11

Essentially, this appears to be an involved convolution of 'sdcvvc's' answer.. – Vaibhav Dec 29 '15 at 10:17

A question, how do I return max(i... , if 'i' is an object? – Vaibhav Dec 29 '15 at 11:33

@Vaibhav `i` isn't an object. – Marcelo Cantos Dec 31 '15 at 1:34

you might want to have a look at NumPy Random sampling distributions

answered Nov 24 '10 at 11:15

Manuel Salvadores
12.6k 3 24 50

3 The numpy functions also seem to only support a limited number of distributions with no support for specifying your own. – pafcu Nov 24 '10 at 11:20

Make a list of items, based on their `weights`:

```
items = [1, 2, 3, 4, 5, 6]
probabilities= [0.1, 0.05, 0.05, 0
# if the list of probs is normaliz
prob = sum(probabilities) # find s
c = (1.0)/prob # a multiplier to m
probabilities = map(lambda x: c*x,
print probabilities

ml = max(probabilities, key=lambda
ml = len(str(ml)) - str(ml).find('
amounts = [ int(x*(10**ml)) for x
itemsList = list()
for i in range(0, len(items)): # i
  itemsList += items[i:i+1]*amount

# choose from itemsList randomly
print itemsList
```

An optimization may be to normalize amounts by the greatest common divisor, to make the target list smaller.

Also, this might be interesting.

edited May 23 '17 at 12:26

Community ♦
1 1

answered Nov 24 '10 at 11:34

khachik
19.7k 5 39 79

If the list of items is large this might use a lot of extra memory. – pafcu Nov 24 '10 at 11:39

@pafcu Agreed. Just a solution, the second which came to my mind (the first one was to search for something like "weight probability python" :)). – khachik Nov 24 '10 at 11:46

Another answer, probably faster :)

```
distribution = [(1, 0.2), (2, 0.3)
# init distribution
dlist = []
sumchance = 0
for value, chance in distribution:
    sumchance += chance
    dlist.append((value, sumchance
assert sumchance == 1.0 # not good

# get random value
r = random.random()
# for small distributions use line
if len(distribution) < 64: # don't
    for value, sumchance in dlist:
        if r < sumchance:
            return value
else:
    # else (not implemented) binar
```

edited Nov 24 '10 at 11:50

answered Nov 24 '10 at 11:38

Lucas Moeskops
4,479 2 19 35

```
from __future__ import division
import random
from collections import Counter


def num_gen(num_probs):
    # calculate minimum probabilit
    min_prob = min(prob for num, p
    lst = []
    for num, prob in num_probs:
        # keep appending num to ls
distribution
        for _ in range(int(prob/mi
            lst.append(num)
    # all elems in lst occur propo
    while True:
        # pick a random index from
        ind = random.randint(0, le
        yield lst[ind]
```

Verification:

```
gen = num_gen([(1, 0.1),
               (2, 0.05),
               (3, 0.05),
               (4, 0.2),
               (5, 0.4),
               (6, 0.2)])
lst = []
times = 10000
for _ in range(times):
    lst.append(next(gen))
# Verify the created distribution:
for item, count in Counter(lst).it
    print '%d has %f probability' %
```

```
1 has 0.099737 probability
2 has 0.050022 probability
3 has 0.049996 probability
4 has 0.200154 probability
5 has 0.399791 probability
6 has 0.200300 probability
```

edited May 2 '15 at 0:40

answered May 2 '15 at 0:10

Saksham Varma
1,612 6 13

---

based on other solutions, you generate accumulative distribution (as integer or float whatever you like), then you can use bisect to make it fast

this is a simple example (I used integers here)

```
l=[(20, 'foo'), (60, 'banana'), (1(
def get_cdf(l):
    ret=[]
    c=0
    for i in l: c+=i[0]; ret.appen(
    return ret

def get_random_item(cdf):
    return cdf[bisect.bisect_left(

cdf=get_cdf(l)
for i in range(100): print get_ran(
```

the `get_cdf` function would convert it from 20, 60, 10, 10 into 20, 20+60, 20+60+10, 20+60+10+10

now we pick a random number up to 20+60+10+10 using `random.randint` then we use bisect to get the actual value in a fast way

edited Apr 26 '16 at 9:48

answered Apr 26 '16 at 9:41

muayyad alsadi

958 7 17

None of these answers is particularly clear or simple.

Here is a clear, simple method that is guaranteed to work.

accumulate_normalize_probabilities takes a dictionary `p` that maps symbols to probabilities **OR** frequencies. It outputs usable list of tuples from which to do selection.

```
def accumulate_normalize_values(p)
        pi = p.items() if isinstan(
        accum_pi = []
        accum = 0
        for i in pi:
                accum_pi.append((i
                accum += i[1]
```

```
        if accum == 0:
                raise Exception( "
? Y/N " )
        normed_a = []
        for a in accum_pi:
                normed_a.append((a
        return normed_a
```

Yields:

```
>>> accumulate_normalize_values( {
[('a', 0.1), ('c', 0.5), ('b', 0.8
```

Why it works

The **accumulation** step turns each symbol into an interval between itself and the previous symbols probability or frequency (or 0 in the case of the first symbol). These intervals can be used to select from (and thus sample the provided distribution) by simply stepping through the list until the random number in interval 0.0 -> 1.0 (prepared earlier) is less or equal to the current symbol's interval end-point.

The **normalization** releases us from the need to make sure everything sums to some value. After normalization the "vector" of probabilities sums to 1.0.

The *rest of the code* for selection and generating a arbitrarily long sample from the distribution is below :

```
def select(symbol_intervals,random
        print symbol_intervals,ran
        i = 0
        while random > symbol_inte
                i += 1
                if i >= len(symbol_
                        raise Exce
        return symbol_intervals[i]


def gen_random(alphabet,length,pro
        from random import random
        from itertools import repe
        if probabilities is None:
                probabilities = di
        elif len(probabilities) > (
(int,long,float)):
                probabilities = di
        usable_probabilities = acc
        gen = []
        while len(gen) < length:
                gen.append(select(
        return gen
```

Usage :

```
>>> gen_random (['a','b','c','d'],
['d', 'b', 'b', 'a', 'c', 'c', 'b'
```

edited Feb 28 '13 at 9:26

answered Feb 28 '13 at 9:15

Cris Stringfellow

2,907 16 38

Here is a **more effective way** of doing this:

Just call the following function with your 'weights' array (assuming the indices as the corresponding items) and the no. of samples needed. This function can be easily modified to handle ordered pair.

Returns indexes (or items) sampled/picked (with replacement) using their respective probabilities:

```
def resample(weights, n):
    beta = 0

    # Caveat: Assign max weight to
    max_w = max(weights)*2

    # Pick an item uniformly at ra
    current_item = random.randint(
    result = []

    for i in range(n):
        beta += random.uniform(0,m

        while weights[current_item
            beta -= weights[curren
            current_item = (curren
        else:
            result.append(current_
    return result
```

A short note on the concept used in the while loop. We reduce the current item's weight from cumulative beta, which is a cumulative value constructed uniformly at random, and increment current index in order to find the item, the weight of which matches the value of beta.

edited Dec 29 '15 at 13:11

answered Dec 29 '15 at 12:57

Vaibhav

1,829 14 20