Random is barely random at all?

Asked 14 years, 11 months ago Modified 5 years, 6 months ago Viewed 58k times

81

I did this to test the randomness of randint:

```
>>> from random import randint
>>>
>>> uniques = []
>>> for i in range(4500):  # You can see I was optimistic.
...     x = randint(500, 5000)
...     if x in uniques:
...         raise Exception('We duped %d at iteration number %d' % (x, i))
...     uniques.append(x)
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
Exception: We duped 887 at iteration number 7
```

I tried about 10 times more and the best result I got was 121 iterations before a repeater. Is this the best sort of result you can get from the standard library?

python random birthday-paradox

Share

Improve this question

Follow

edited Jun 8, 2015 at 12:12

Andriy M

77.6k • 18 • 98 • 157

asked Jan 27, 2010 at 8:50

orokusaki

57k • 60 • 183 • 263

62 "The Pragmatic Programmer", rule 26. "select" Isn't Broken. It is rare to find a bug in the OS or the compiler, or even a third-party product or library. The bug is most likely in the application. Or in this case, the application of probability theory. – user97370 Jan 27, 2010 at 9:11

11 Just nitpicking: uniques = set() and uniques.add(x) would be more appropriate (efficient). – Eric O. Lebigot Jan 27, 2010 at 10:23

23 One of the key properties of the birthday paradox is that it's counter-intuitive. Unless you are aware of it or have some background in probability theory then you would not necessarily have any reason to do a keyword search for it. One of the USP's of Q&A sites is that you can ask a question in terms that would never actually match answers to the question if you did a pure keyword search without knowing what to search for. – ConcernedOfTunbridgeWells Jan 28, 2010 at 0:07

8 @okoku: (regarding your reply to ConcernedOfTunbridge): what you are talking about is a completely different problem. One is the probability of getting the same card twice in a row; the other is the probability of getting *ANY* of the previous N-1 cards after N picks. The *average* number of cards from a **perfect** RNG for the second problem should be about 67;

considering you got anywhere from 8 to 121, that sounds about right.
– BlueRaja - Danny Pflughoeft Jan 28, 2010 at 17:24

6 you are confusing Random with Evenly Distributed. It is perfectly valid for a random generator to return the exact same value over and over numerous times. If you want an Evenly Distributed Random number generator that is a completely different problem, it is a shuffling problem not a generator problem. – user177800 Jan 28, 2010 at 17:39

11 Answers

Sorted by: Highest score (default)

296

The Birthday Paradox, or why PRNGs produce duplicates more often than you might think.

There are a couple of issues at play in the OP's problem. One is the birthday paradox as mentioned above and the second is the nature of what you are generating, which does not inherently guarantee that a given number will not be repeated.

The Birthday Paradox applies where given value can occur more than once during the period of the generator - and therefore duplicates can happen within a sample of values. The effect of the Birthday Paradox is that the real likelihood of getting such duplicates is quite significant and the average period between them is smaller than one might otherwise have thought. This dissonance between the perceived and actual probabilities makes the Birthday Paradox a good example of a cognitive bias, where a naive intuitive estimate is likely to be wildly wrong.

A quick primer on Pseudo Random Number Generators (PRNGs)

The first part of your problem is that you are taking the exposed value of a random number generator and converting it to a much smaller number, so the space of possible values is reduced. Although some pseudo-random number generators do not repeat values during their period this transformation changes the domain to a much smaller one. The smaller domain invalidates the 'no repeats' condition so you can expect a significant likelihood of repeats.

Some algorithms, such as the linear congruential PRNG (`A'=AX|M`) *do* guarantee uniqueness for the entire period. In an LCG the generated value contains the entire state of the accumulator and no additional state is held. The generator is deterministic and cannot repeat a number within the period - any given accumulator value can imply only one possible successive value. Therefore, each value can only occur once within the period of the generator. However, the period of such a PRNG is relatively small - about 2^30 for typical implementations of the LCG algorithm - and cannot possibly be larger than the number of distinct values.

Not all PRNG algorithms share this characteristic; some can repeat a given value within the period. In the OP's problem, the Mersenne Twister algorithm (used in Python's random module) has a very long period - much greater than 2^32. Unlike a Linear Congruential PRNG, the result is not purely a function of the previous output value as the accumulator contains additional state. With 32-bit integer output and a period of ~2^19937, it cannot possibly provide a such a guarantee.

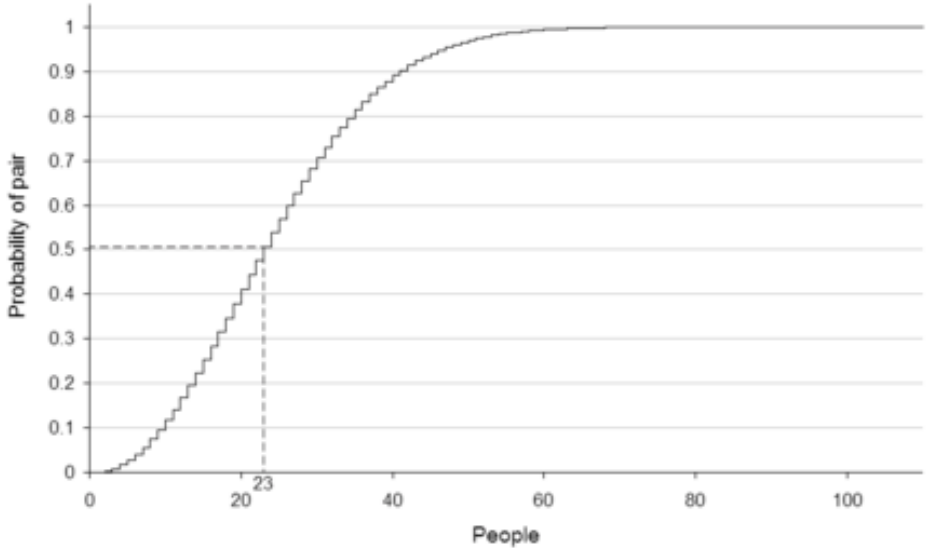
The Mersenne Twister is a popular algorithm for PRNGs because it has good statistical and geometric properties and a very long period - desirable characteristics for a PRNG used on simulation models.

- Good statistical properties mean that the numbers generated by the algorithm are evenly distributed with no numbers having a significantly higher probability of appearing than others. Poor statistical properties could produce unwanted skew in the results.
- Good geometric properies mean that sets of N numbers do not lie on a hyperplane in N-dimensional space. Poor geometric properties can generate spurious correlations in a simulation model and distort the results.
- A long period means that you can generate a lot of numbers before the sequence wraps around to the start. If a model needs a large number of iterations or has to be run from several seeds then the 2^30 or so discrete numbers available from a typical LCG implementation may not be sufficient. The MT19337 algorithm has a very long period - 2^19337-1, or about 10^5821. By comparison, the total number of atoms in the universe is estimated at about 10^80.

The 32-bit integer produced by an MT19337 PRNG cannot possibly represent enough discrete values to avoid repeating during such a large period. In this case, duplicate values are likely to occur and inevitable with a large enough sample.

The Birthday Paradox in a nutshell

This problem is originally defined as the probability of any two people in the room sharing the same birthday. The key point is that ***any two*** people in the room could share a birthday. People tend to naively misinterpret the problem as the probability of someone in the room sharing a birthday with a specific individual, which is the source of the cognitive bias that often causes people to underestimate the probability. This is the incorrect assumption - there is no requirement for the match to be to a specific individual and any two individuals could match.



The probability of a match occurring between any two individuals is much higher than the probability of a match to a specific individual as the match does not have to be to a specific date. Rather, you only have to find two individuals that share the same birthday. From this graph (which can be found on the Wikipedia page on the subject), we can see that we only need 23 people in the room for there to be a 50% chance of finding two that match in this way.

From the Wikipedia entry on the subject we can get a nice summary. In the OP's problem, we have 4,500 possible 'birthdays', rather than 365. For a given number of random values generated (equating to 'people') we want to know the probability of *any* two identical values appearing within the sequence.

Computing the likely effect of the Birthday Paradox on the OP's problem

For a sequence of 100 numbers, we have $\binom{100}{2} = \frac{100 \cdot 99}{2} = 4950$ pairs (see Understanding the Problem) that could potentially match (i.e. the first could match with the second, third etc., the second could match the third, fourth etc. and so on), so the number of *combinations* that could potentially match is rather more than just 100.

From Calculating the Probability we get an expression of $\frac{4500!}{4500^{100}(4500-100)!}$. The following snippet of Python code below does a naive evaluation of the probability of a matching pair occurring.

```
# === birthday.py ============================================
#
from math import log10, factorial

PV=4500          # Number of possible values
SS=100           # Sample size

# These intermediate results are exceedingly large numbers;
# Python automatically starts using bignums behind the scenes.
#
numerator = factorial (PV)
```

```
denominator = (PV ** SS) * factorial (PV - SS)

# Now we need to get from bignums to floats without intermediate
# values too large to cast into a double.  Taking the logs and
# subtracting them is equivalent to division.
#
log_prob_no_pair = log10 (numerator) - log10 (denominator)

# We've just calculated the log of the probability that *NO*
# two matching pairs occur in the sample.  The probability
# of at least one collision is 1.0 - the probability that no
# matching pairs exist.
#
print 1.0 - (10 ** log_prob_no_pair)
```

This produces a sensible looking result of ***p=0.669*** for a match occurring within 100 numbers sampled from a population of 4500 possible values. (Maybe someone could verify this and post a comment if it's wrong). From this we can see that the lengths of runs between matching numbers observed by the OP seem to be quite reasonable.

Footnote: using shuffling to get a unique sequence of pseudo-random numbers

See this answer below from S. Mark for a means of getting a guaranteed unique set of random numbers. The technique the poster refers to takes an array of numbers (which you supply, so you can make them unique) and shuffles them into a random order. Drawing the numbers in sequence from the shuffled array will give you a sequence of pseudo-random numbers that are guaranteed not to repeat.

Footnote: Cryptographically Secure PRNGs

The MT algorithm is not cryptographically secure as it is relatively easy to infer the internal state of the generator by observing a sequence of numbers. Other algorithms such as Blum Blum Shub are used for cryptographic applications but may be unsuitable for simulation or general random number applications. Cryptographically secure PRNGs may be expensive (perhaps requiring bignum calculations) or may not have good geometric properties. In the case of this type of algorithm, the primary requirement is that it should be computationally infeasible to infer the internal state of the generator by observing a sequence of values.

edited Jun 9, 2019 at 7:06

community wiki
43 revs, 6 users 93%
ConcernedOfTunbridgeWells

> One correction: LCG-based PRNGs, used properly, do *not* guarantee unique output for the complete cycle. For example, the traditional Turbo Pascal LCG has (IIRC) 31 bits of internal state, but it only generates 15-bit numbers which can and do repeat within a single cycle.
> – Porculus Feb 24, 2010 at 10:16

47

Before blaming Python, you should really brush up some probability & statistics theory. Start by reading about the birthday paradox

By the way, the `random` module in Python uses the Mersenne twister PRNG, which is considered very good, has an enormous period and was extensively tested. So rest assured you're in good hands.

Share
Improve this answer
Follow

edited Jan 27, 2010 at 9:00

answered Jan 27, 2010 at 8:51

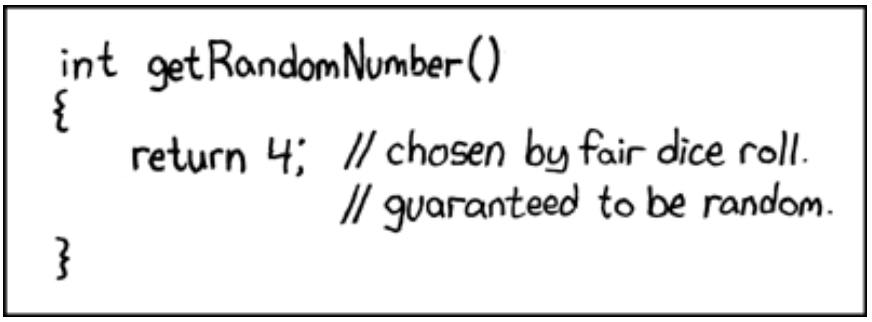
Eli Bendersky
273k ● 91 ● 362 ● 422

2 Eh, I don't think he needed to brush up on probability and statistics theory. Just understand that if a number is random in a range, it can still repeat. After all, the idea of a random value is it can be anything, even the same number. – dtc Apr 4, 2022 at 2:52

45

As an answer to the answer of Nimbuz:

http://xkcd.com/221/



Share
Improve this answer
Follow

edited Feb 8, 2017 at 14:07

Community Bot
1 ● 1

answered Jan 29, 2010 at 11:56

Curd
12.4k ● 3 ● 36 ● 50

2 @Zano: as an aside, do you know that RFC 1149 actually exists and has also been implemented at least once. That was a IP for data transmission through avian carriers. The implementation achieved 45% data transmission with just about 2 hour ping. – Arani Jun 28, 2022 at 10:39

43

If you don't want repetative one, generate sequential array and use random.shuffle

Share Improve this answer Follow

answered Jan 27, 2010 at 8:54

YOU
124k ● 34 ● 189 ● 222

3 God I love `random.shuffle`. It's one of the cores of my project :) – PizzAzzra Jun 9, 2010 at 3:11

16

True randomness definitely includes repetition of values before the whole set of possible values is exhausted. It would not be random otherwise, as you would be able to predict for how long a value would not be repeated.

If you ever rolled dice, you surely got 3 sixes in row quite often...

Share
Improve this answer
Follow

edited Feb 24, 2010 at 12:29

answered Jan 27, 2010 at 9:10

Ber
41.7k ● 16 ● 77 ● 89

6

You are generating `4500` random numbers from a range `500 <= x <= 5000`. You then check to see for each number whether it has been generated before. The birthday problem tells us what the probability is for two of those numbers to match given `n` tries out of a range `d`.

You can also invert the formula to calculate how many numbers you have to generate until the chance of generating a duplicate is more than `50%`. In this case you have a `>50%` chance of finding a duplicate number after `79` iterations.

Share
Improve this answer
Follow

edited Jan 28, 2010 at 4:06

YOU
124k ● 34 ● 189 ● 222

answered Jan 27, 2010 at 9:30

liwp
6,926 ● 2 ● 28 ● 40

5

Python's random implementation is actually quite state of the art:

- http://en.wikipedia.org/wiki/Mersenne_twister
- http://docs.python.org/library/random.html

Share Improve this answer Follow

answered Jan 27, 2010 at 8:57

bayer
6,904 ● 26 ● 35

That's not a repeater. A repeater is when you repeat the same *sequence*. Not just one number.

5

Share Improve this answer Follow

answered Jan 27, 2010 at 9:34

Lennart Regebro

172k ● 44 ● 228 ● 253

1

There is the birthday paradox. Taking this into account you realize that what you are saying is that finding "764, 3875, 4290, 4378, 764" or something like that is not very random because a number in that sequence repeats. The true way to do it is to compare sequences to each other. I wrote a python script to do this.

```
from random import randint
y = 21533456
uniques = []
for i in range(y):
    x1 = str(randint(500, 5000))
    x2 = str(randint(500, 5000))
    x3 = str(randint(500, 5000))
    x4 = str(randint(500, 5000))
    x = (x1 + ", " + x2 + ", " + x3 + ", " + x4)
if x in uniques:
    raise Exception('We duped the sequence %d at iteration number %d' % (x, i))
else:
    raise Exception('Couldn\'t find a repeating sequence in %d iterations' %
(y))
uniques.append(x)
```

Share

Improve this answer

Follow

edited Jun 11, 2015 at 21:26

Kara

6,206 ● 16 ● 53 ● 58

answered Jul 31, 2014 at 17:03

raspberry guy

11 ● 1

This answer was given years ago (see selected answer above). Its not called the birthday paradox, since it's not a paradox, rather just the birthday problem. – orokusaki Jul 31, 2014 at 17:06

1 @orokusaki It is the birthday problem and the associated birthday paradox because humans tend to underestimate the correct probability widely (a form of cognitive bias). See en.wikipedia.org/wiki/Birthday_problem – xuiqzy Jan 24, 2022 at 17:54

0

You have defined a random space of 4501 values (500-5000), and you are iterating 4500 times. You are basically guaranteed to get a collision in the test you wrote.

To think about it another way:

- When the result array is empty P(dupe) = 0

- 1 value in Array P(dupe) = 1/4500
- 2 values in Array P(dupe) = 2/4500
- etc.

So by the time you get to 45/4500, that insert has a 1% chance of being a duplicate, and that probability keeps increasing with each subsequent insert.

To create a test that truly tests the abilities of the random function, increase the universe of possible random values (eg: 500-500000) You may, or may not get a dupe. But you'll get far more iterations on average.

Share Improve this answer Follow

answered Jan 27, 2010 at 9:24

sfrench
910 ● 5 ● 9

4 Your math is incorrect because of the birthday problem. See other answers. After 45 inserts, you have a 1% chance of having repeated the first insert, but you also have 44 other distinct inserts that you might have repeated. – jcdyer Jan 28, 2010 at 17:58

I read the answer of them as we have x *distinct* values already in there and when 45 distinct values are there, the next one inserted has a probability of 1% of being a duplicate. But it is not clear and it is correct that the probability of having at least one duplicate after 45 inserts is therefore far greater than 1%. – xuiqzy Jan 24, 2022 at 10:37

For anyone else with this problem, I used uuid.uuid4() and it works like a charm.

0

Share Improve this answer Follow

answered Jan 27, 2010 at 20:31

orokusaki
57k ● 60 ● 183 ● 263

3 The question then might have been better phrased as "I want to generate a series of non-repeating numbers, Python's randint() doesn't seem to do that - what does?" rather than "Python's random number generator is bad" :-) Assuming uuid4() is truly random, it may still repeat - just really unlikely. What are the actual properties you want from the numbers? Non-repeating? Random? (Pick one.) Not-repeating-often? (Use a bigger int range, effectively all uuid4 is, it seems.) What exactly do you want to use the numbers *for* is the real question. – agnoster May 20, 2010 at 11:53

@agnoster I really didn't intend on insulting Python, but Random: Lack of predictability, without any systematic pattern, and Repeating Pattern: A pattern of a group of items that repeats over and over. See, the random generator is not random if it repeats because it then has a pattern. – orokusaki May 22, 2010 at 0:36

10 Your definition of "random" is wrong. Seriously, go back and re-read the bits on the birthday paradox. A truly random number generator will still have repeats much more frequently than you expect by intuition. As @ConcernedOfTunbridgeW points out, the probability of getting a repeat in the range 500-5000 within the first 100 numbers is ~66%, not at all inconsistent with what you observed, I believe. Randomness does *not* mean "without repeats", it just means... well, random. In fact, if you guarantee a lack of repeats the generator must be *less* random in order to enforce that. – agnoster May 24, 2010 at 6:06

2 The question about what you want these numbers for still stands. If you specifically want non-repeating numbers, why? uuid4() is (if it's truly random) no different from randint() with a very very large range. If you want the sequence to be hard to guess, eliminating repeats actually hurts you, because once I see the number, say, 33, I know that whatever comes next *doesn't* have 33 in it. So enforcing non-repetition actually makes your sequence *more* predictable - do you see? – agnoster May 24, 2010 at 6:12

@agnoster An implementation of uuid4 might guarantee that it uses / might use a cryptographically secure source of randomness in contrast to `randint` (Only the `secrets` python module is cryptographically secure for random numbers). See also security.stackexchange.com/questions/165987/… and stackoverflow.com/questions/41505448/… – xuiqzy Jan 24, 2022 at 10:33