Eric Rasmusen
November 30, 2022

Why 0.9999 . . . Equals 1.0000. . .

In the 7th-grade math class I teach, when we discussed closed and open intervals I stumbled a bit when it came to talking about 0.99999 . . . possibly being the edge of an open interval. I told the students that for technical reasons we can't say that, and 0.99999 . . . = 1, but to explain why would be too hard. Job Currell found out why from his brother Isaac, and we talked about it in class, which was good, but I thought I'd better write up a handout to explain this weirdness. It's something I'd never heard of myself before I was in my 50's, when Professor Christopher Connell told me about it. That handout became this Substack article.

Theorem 1: $0.999\ldots = 1$.

Proof. Define the number $zog1$ as $zog1 \equiv \frac{1}{3} = 0.3333\ldots$. (Remember, "≡" means "equals by definition", whereas "=" means "happens to equal".) Three times $zog1$ equals one, right?— because $3 \cdot \frac{1}{3} = 1$.

Thus, using the $0.3333\ldots$ way to write $zog1$, three times $zog1$ equals $3 \cdot 0.3333\ldots = 1$.

But

$$3 \cdot 0.3333\ldots = 0.999\ldots$$

Therefore, it must be that $0.999\ldots = 1$. Q.E.D.

I chose the name $zog1$ for the variable for $0.999\ldots$ in honor of Zog I, King of Albania from 1928 to 1939, when Italy conquered his country in World War II. He was born Ahmet Muhtar Zogolli, but changed his name to Zog. He was a favorite of my college debating club, and, in fact, when we were close to electing a chairman whose was a fine fellow but lacked humility, we instead elected King Zog and elected him as merely the Acting Chairman.

Mathematicians and math teachers have thought a lot about .999. . ., as you can see from the Wikipedia article at https://en.wikipedia.org/wiki/0.999.... That article seems to me to make it unnecessarily complicated. I don't like its Archimidean Principle proofs based on the number line and finding a least upper bound. I don't see why making the least upper bound 1 matters to $0.999\ldots$, which it seems to me could be *less* than the least upper bound when

the argument assumes that it *is* the least upper bound. I think that proof is assuming its answer, by defining $0.999\ldots$ in a certain way.

I don't see what's nonrigorous about the proof I gave of Theorem 1. It's the only way to make our ordinary rules of arithmetic work for $0.999\ldots$, a pretty convincing reason to accept that $0.999\ldots = 1$. Whenever we work with infinity in math, we have to be careful, because it's not a genuine number, it's shorthand for "and then keep going with whatever you're doing forever". We don't say that $4/0$ equals infinity, we just leave $4/0$ undefined, because if we say that $4/0$ = infinity, and $5/0$ = infinity too, we end up having to say that $4 = 5$ if we follow the ordinary rules of arithmetic. We can save arithmetic by saying that $0.999\ldots = 1$. It's weird, but that's because we think of $0.999\ldots$ as a genuine number and it really is not since it's built using the idea of infinity. We either have to say that $0.999\ldots$ is undefined, like $4/0$, or define it to equal 1, as Theorem 1 does. What Theorem 1 is doing is assuming implicitly– that is, assuming without coming out and saying it is making that assumption– that we want to define and can define $0.999\ldots$ in such a way that arithmetic works on it.

Another way to think about $0.999\ldots$ is not as its own number but as another way to write "1", just as *zog*1 was another way to write $0.333\ldots$ If you think about it as a separate number, you will get all mixed up because there's no way to make it its own number without wrecking arithmetic.

Also, if you just think of it as another way to write "one", you can make a good joke:

Question: How many mathematicians does it take to screw in a lightbulb?
Answer: $0.999\ldots$

Here's another way to think about it. **The object "$0.999\ldots$" is an infinite series, not a number.** That's because what our dot-dot-dot notation really means is:

$$\begin{aligned} zog \equiv 0.999\ldots \quad &\equiv 0 + 9 \cdot \tfrac{1}{10} + 9 \cdot \tfrac{1}{100} + 9 \cdot \tfrac{1}{1{,}000} + \ldots \\ &= 9 \cdot \tfrac{1}{10^1} + 9 \cdot \tfrac{1}{10^2} + 9 \cdot \tfrac{1}{10^3} + \ldots \\ &= 9 \cdot (\tfrac{1}{10^1} + \tfrac{1}{10^2} + \tfrac{1}{10^3} + \ldots) \\ &= 9 \cdot \sum_{i=1}^{\infty} \frac{1}{10^i} \end{aligned}$$

This is easier to see as a computer program. Let's do it in Python. First, here is hard version of the code to print $0.999\ldots$ to 20 places.[1] Before, we defined $zog1 \equiv .333\ldots$. Now let's define plain zog as $zog \equiv .999\ldots$.

```
#Python 3 code to generate increasingly accurate printouts of 0.99999...
#Eric Rasmusen, erasmuse61@gmail.com, November 28, 2022.
#This is the full version. It woul be even better with a plot of the values, but I don't have time.

import decimal as d #For the number of decimal places, the precision level.

#Now we will initialize some values.
d.getcontext().prec = 64 #Sets decimal places you get displayed from d.Decimal()

end =20 #end must be 64 or smaller, or Python can't handle it.
estimate = d.Decimal(0) #Convert our starting value of 0 this so it will display lots of decimal places

for item in range(1,end+1):
    estimate =  estimate +   9*d.Decimal(10**(-item) )
     estimate = round( estimate, item)
    #The computer makes tiny errors because it computes in base 2, not 10.
    print("Decimal places:",item, "     estimate:", estimate)

print ("_____________________________________\n")
print("Our final value for approximating zog  is",  estimate)
print ("_____________________________________")
```

The output looks like this:

```
Decimal places: 1     estimate: 0.9
Decimal places: 2     estimate: 0.99
Decimal places: 3     estimate: 0.999
Decimal places: 4     estimate: 0.9999
Decimal places: 5     estimate: 0.99999
Decimal places: 6     estimate: 0.999999
Decimal places: 7     estimate: 0.9999999
Decimal places: 8     estimate: 0.99999999
Decimal places: 9     estimate: 0.999999999
Decimal places: 10     estimate: 0.9999999999
Decimal places: 11     estimate: 0.99999999999
Decimal places: 12     estimate: 0.999999999999
Decimal places: 13     estimate: 0.9999999999999
Decimal places: 14     estimate: 0.99999999999999
Decimal places: 15     estimate: 0.999999999999999
Decimal places: 16     estimate: 0.9999999999999999
Decimal places: 17     estimate: 0.99999999999999999
Decimal places: 18     estimate: 0.999999999999999999
Decimal places: 19     estimate: 0.9999999999999999999
Decimal places: 20     estimate: 0.99999999999999999999
---------------------------------

Our final value for approximating zog is 0.99999999999999999999
---------------------------------
```

[1] You can make it run by copying and pasting this code into https://www.cco/en/python-compiler.

That code has bells and whistles, so don't read it unless you like coding. The basics are in this stripped-down version:[2]

```
end =20

estimate = 0
for item in range(1,end+1):
    estimate = estimate +    9*10**(-item)
    print("Decimal places:",item, "    estimate:", estimate)

print("Our final value for approximating zog  is", estimate)
```

The output looks like this:

```
Decimal places: 1     estimate: 0.9
Decimal places: 2     estimate: 0.99
Decimal places: 3     estimate: 0.999
Decimal places: 4     estimate: 0.9999
Decimal places: 5     estimate: 0.99999
Decimal places: 6     estimate: 0.9999990000000001
Decimal places: 7     estimate: 0.9999999
Decimal places: 8     estimate: 0.9999999900000001
Decimal places: 9     estimate: 0.999999999
Decimal places: 10     estimate: 0.9999999999
Decimal places: 11     estimate: 0.99999999999
Decimal places: 12     estimate: 0.999999999999
Decimal places: 13     estimate: 0.9999999999999
Decimal places: 14     estimate: 0.99999999999999
Decimal places: 15     estimate: 0.999999999999999
Decimal places: 16     estimate: 0.9999999999999999
Decimal places: 17     estimate: 1.0
Decimal places: 18     estimate: 1.0
Decimal places: 19     estimate: 1.0
Decimal places: 20     estimate: 1.0
Our final value for approximating zog  is 1.0
```

Let's go over the logic of what the computer is doing. For the moment, ignore the strange things that happen after 5 decimal places using the stripped-down code. You can get the logic from the first five decimal places. First, we set $end = 20$, to

[2]You can make it run by copying and pasting this code into https://www.cco/en/python-compiler.

do this for 20 decimal places (but we could change that to 5 or 40 easily). Then we set *estimate* = 0, just to get started. Then comes the line

```
for item in range(1,end+1):
```

What this says is for the computer to get ready to do something for all the numbers between 1 and *end*, inclusive., which means from 1 to 20. The colon means "And now comes what the computer is supposed to do, in the indented next lines of the program". So we need to look at those next lines. First comes

```
estimate = estimate +    9*10**(-item)
```

That is the "work line" of the program. It says to change the value of *estimate* to equal whatever the old level of *estimate* is– which is *estimate* = 0 at the start– and then add $9(10^{-item}$, using the current value of *item*. The first number assigned to *item* was 1, so this says the new value of item is going to be $0 + 9(10^{-1}$, which is $9 \cdot .1$, so now *estimate* = .9.

Python doesn't print out any results unless you tell it to (this saves computing time and space when you don't want to print every single number calculated). So the next line tells the computer to print something:

```
print("Decimal places:",item, "    estimate:", estimate)
```

The print line tells the computer to print the words "Decimal places:", followed by the value of *item*, which is 1, followed by the spaces and word " estimate", followed by the value of *estimate*, which is .9. So it prints that, and we see it as the first line of output,

```
Decimal places: 1     estimate: 0.9
```

Those two lines, the work line and the print line, are the only two that are indented, so the computer then goes back and does everything again in a loop, this time using *item* = 2 instead of *item* = 1. This time the old value of *estimate* is .9, and we calculate the new one as $estimate = .9 + 9 \cdot 10^{-2}$ which equals $.9 + 9 \cdot .01$, so we get *estimate* = .9 + .09 = .99. And the print line prints out using *item* = 2 and *estimate* = .99, giving us the second line of the output,

```
Decimal places: 2     estimate: 0.99
```

Then the computer goes back again and uses *item* = 3 and an old estimate of .99 to get $estimate = .99 + 9(10^{-3} = .99 + .009 = .999$ and prints out those values

to give the third line of output. And it keeps going all the way up to the last value of *item* that we specified, which was $item = end = 20$.

One of my points in going through all this is to show you how to use do-loops in Python coding. More important, though is that this shows that *zog* is a process, as much as a number. When we write .999..., we are talking about the process of adding 9's forever. It is like setting *end* to equal 10^{31}, which I think would cause the computer to go on till Resurrection Day, or, for you heathens, till the sun burns out (maybe I'm wrong, in which case try 10^{3100}; infinity means keep going forever).

Now I'll explain why the code produces peculiar results at the 6th decimal places. What happens there is that the computer prints out

```
Decimal places: 6     estimate: 0.9999990000000001
```

not what we expected, which was

```
Decimal places: 6     estimate: 0.999999
```

What's going on? The problem is with how computers do arithmetic. Computers are "digital", but they don't use base-10 numbers, they use base-2. When you tell a computer to calculate 10^{-3}, it has to convert 10 into base-2, which converts it to 1010 since $10 = 2^3 + 0 + 2^1 + 0$. Then it does the calculations. If it's a big calculation, there's going to be some rounding error. In base-10, 1/3 is .33333..., so it needs rounding. In base-2, 1/10 is .0001001100110011..., so it needs rounding too. That's why the computer made a mistake, and was off by 0.0000000000000001 from the right answer. With 7 decimal places, the rounding errors cancelled out. With 8 decimal places, they didn't. With 9 decimal places they did again, and everything was fine up through 16 decimal places. At 17 decimal places, though, we get 1.0 instead of 0.99999999999999999.

The extra complexity in the longer version of the Python program adds extra precision in the calculations and the printout. We can go up to 64 decimal places, so it's no use setting *end* any bigger.

So much for Python. Let's now go back to proving that .999... = 1. This time I'll prove it another way. We proved it as Theorem 1 before. I'll restate it as Theorem 2, even though it's really the same as Theorem 1, just with a different proof.

Theorem 2. $0.999\ldots = 1$.

Proof. Let's define zog as:

$$zog \equiv 0.999\ldots \tag{1}$$

Notice that

$$zog = 9 \cdot (.1 + .01 + .001 + .0001 + \ldots) \tag{2}$$

$$zog = 9 \cdot \left(\frac{1}{10} + \frac{1}{100} + \frac{1}{1,000} + \frac{1}{10,000} + \ldots\right)$$

$$zog = 9 \cdot \left(\frac{1}{10^1} + \frac{1}{10^2} + \frac{1}{10^3} + \frac{1}{10^4} + \ldots\right) \tag{3}$$

$$zog = 9 \cdot \sum_{i=1}^{\infty} \frac{1}{10^i} \tag{4}$$

Now let's split up zog into two parts from equation (3):

$$zog = 9\left(\frac{1}{10^1}\right) + 9\left(\frac{1}{10^2} + \frac{1}{10^3} + \frac{1}{10^4} + \ldots\right)$$

$$zog = .9 + 9\left(\frac{1}{10}\right)\left(\frac{1}{10^1} + \frac{1}{10^2} + \frac{1}{10^3} + \ldots\right)$$

$$zog = .9 + \left(\frac{1}{10}\right)\left(9 \cdot \sum_{i=1}^{\infty} \frac{1}{10^i}\right) \tag{5}$$

But remember that equation (4) told us that $zog = 9 \cdot \sum_{i=1}^{\infty} \frac{1}{10^i}$. Inserting this into equation (5) we get:

$$zog = .9 + \frac{1}{10} \cdot zog$$

Thus,

$$zog - \frac{1}{10} \cdot zog = .9$$

$$\frac{9}{10} \cdot zog = \frac{9}{10}$$

$$zog = 1 \tag{6}$$

Since our definition in equation (1) says that $zog \equiv .999\ldots$, equation (6) can be rewritte as

$$0.999\ldots = 1$$

Q.E.D.

So we've shown that .999... is not a genuine number, just a strange way to write 1.0. My student Job Currell brought up that .3333... is not a genuine number either, though we think of it as equalling 1/3. It is, rather a process, or a label. $1/3 \approx .33$ and $1/3 \approx .333333$, but all we have is an approximation that gets better and better, closer and closer to 1/3. Using base-10 numbers, we can't have an exact representation of 1/3, because 10 doesn't divide into 3's evenly.

Infinity really means "keep going with what you're doing forever," where that "what you're doing" might be counting to bigger numbers or might be approximating more exactly or might be continuing to do a division problem to more and more digits or might be drawing a straight line longer and longer. So infinity has more of the flavor of a verb than a noun. Jonathan Swift wrote *Gulliver's Travels.* From Jonathan Swift's poem, "On Poetry: A Rapsody" (1733):

```
The Vermin only teaze and pinch
Their Foes superior by an Inch.
So, Nat'ralists observe, a Flea
Hath smaller Fleas that on him prey,
And these have smaller yet to bite 'em,
And so proceed ad infinitum:
Thus ev'ry Poet, in his Kind
Is bit by him that comes behind.
```

More succinctly, from August De Morgan's poem, "Siphonaptera" (1872):

```
Great fleas have little fleas upon their backs to bite 'em,
And little fleas have lesser fleas, and so ad infinitum.
```

De Morgan was a mathematician, and in logic De Morgan's Laws are named after him. On both poems, see the Wikipedia article. Note also that even bacteria get infected by viruses, and there are 10^{31} of these "bacteriophages" on earth, even bigger than Avogadro's number ($6x10^{23}$) and more than of all other organisms combined (including bacteria)– by number, though maybe not by weight.

This is why you never get closer to infinity when you're counting— the process can continue no matter where you have gotten to. From the hymn Amazing Grace:

```
When we've been there ten thousand years,
Bright shining as the sun,
We've no less days to sing God's praise
Than when we'd first begun.
```

Zeno's Paradox is worth thinking about in this connection, as USA Today sabremetrician and Central-Time-Zone-for-Indiana advocate Jeff Sagarin pointed out to me at lunch. Zeno of Elea (490–430 BC) told the story of Achilles and the Tortoise, who are in a race to a finish line 100 yards away. The Tortoise gets a 10-yard head start. In the first second, Achilles goes 10 yards, catching up. The Tortoise, however, has gone 1 yard by then, to the 11-yard mark, so it is still ahead. Achilles then runs up to the 11-yard mark. By then, though, the Tortoise has made it to the 11.1-yard mark and is still ahead. So Achilles keeps going to 11.1. But by then the Tortoise has made it to 11.11 and is still ahead. So how can Achilles ever catch up to the Tortoise?[3]

To solve the paradox, think back to *zog* as a sum. Equation (2) wrote *zog* as

$$zog = 9 \cdot (.1 + .01 + .001 + .0001 + \ldots) = 9 \cdot \sum_{i=1}^{\infty} \frac{1}{10^i}$$

I glossed over it before, but this means *zog* is the sum of an infinite number of numbers. If you add an infinite number of numbers, don't get you an infinite result? After all, $1 + 1 + 1 + \ldots = \infty$. If you keep adding ones, it really adds up. But we showed that $zog = 1$, so we have a paradox— logical reasoning that reaches two contradictory answers.

The summation paradox is solved by noticing that the *zog* sum isn't like adding one plus one forever. The big difference is that in the *zog* sum we're adding smaller and smaller numbers as we go along. First we add .1, then .001, then .0001, and so forth. The rate at which the numbers are getting smaller is a faster rate than the rate at which we're adding them. Thus, the sum is growing more and more slowly, and though it never grinds to a halt, the sum never becomes infinite. In fact, it flattens out at 1, which is why $zog = 1$.

Now we can go back to Zeno's Paradox. The problem Achilles faces is how to overcome an infinite series of head starts that the Tortoise gets. We can measure the amount the Tortoise is ahead either in yards or in seconds. Either way, it's not an infinite amount, because the sum is made up of smaller and smaller additions, so it converges. Achilles does have an infinite number of distances he has to catch up to— but that infinite number of distances is a finite distance! Thus, he has plenty of time to catch up past that infinite sum. He will win after all.

[3] My student Elijah Magnus pointed out that Zeno's Paradox is retold in the context of a mad scientist in a wheelchair in *The Mysterious Benedict Society: Mr. Benedict's Book of Perplexing Puzzles, Elusive Enigmas, and Curious Conundrums* by Trenton Lee Stuart, http://antinode.info/complaints/mbs.html (2011). In class, I had Job Achilles and Elijah Tortoise act the paradox out. In an earlier draft I said it was Liam Tucker who brought up the book, but Liam has corrected me.

Well, this is a lot to absorb. You should probably read it over a few times. I hope I've made it interesting enough to do that. It's worth it. You can rest satisfied that if you understand this article, you are well-poised to understand a lot of pretty advanced mathematics— not an infinite amount, but a lot.