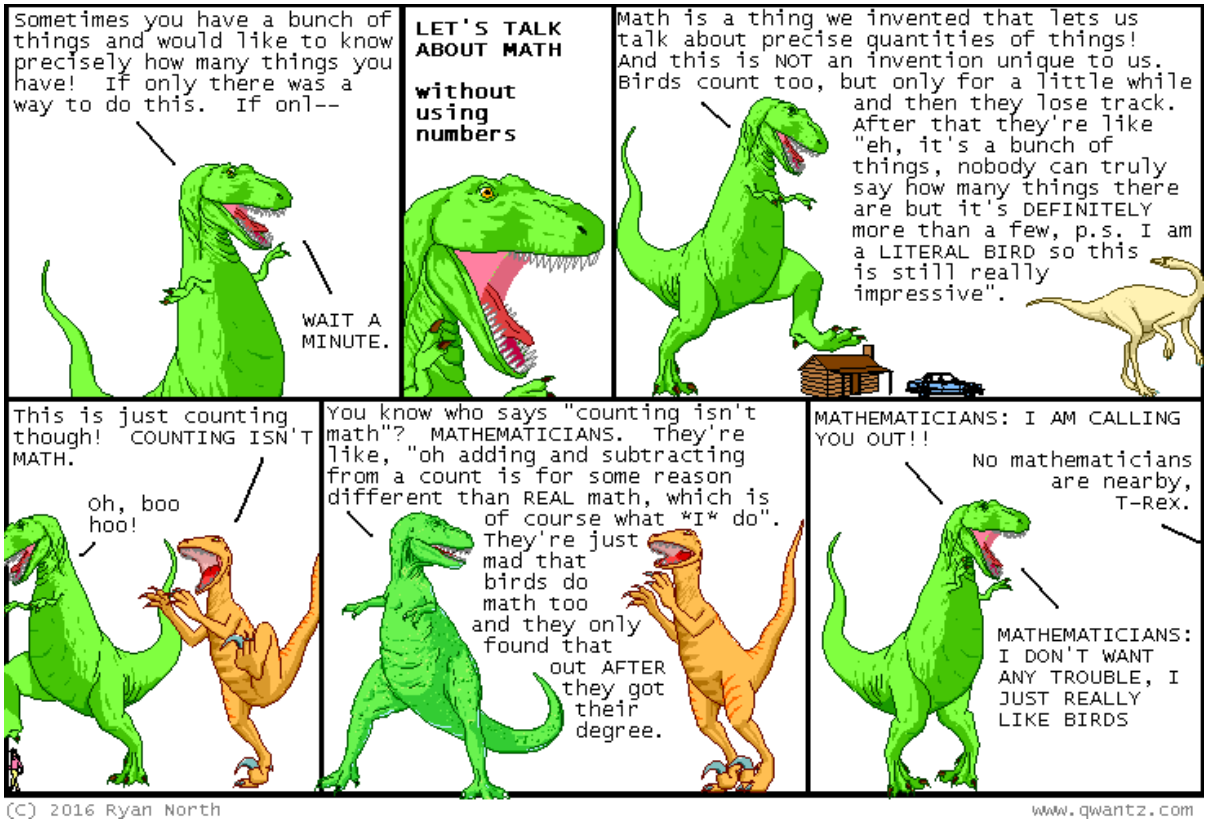
Lecture 04: Numbers, recursion, and continued fractions.

In the tutorial this week, we are experimenting with numbers and their different representations.

We explore how to compute iterations of rational numbers that converge to irrational numbers.

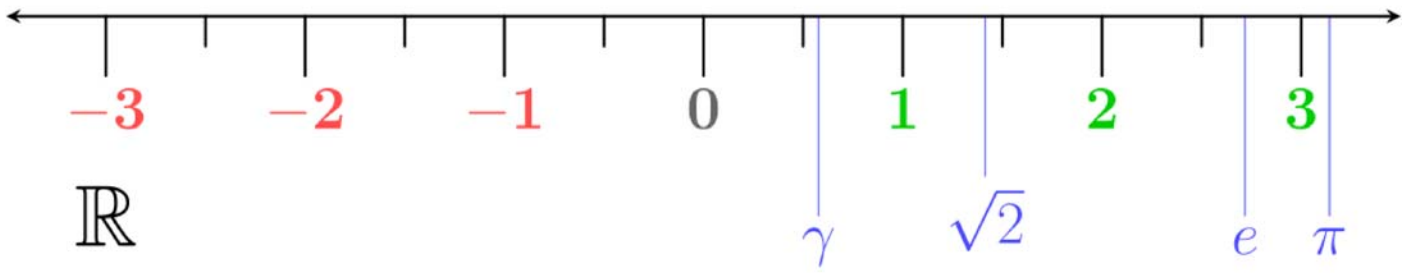
Dino, telling it like it is!

http://www.qwantz.com/index.php?comic=2976 (http://www.qwantz.com/index.php?comic=2976)



Here's a review of introductory material in that tutorial:

It seems obvious that real numbers $\mathbb{R}$ are a key element of computation. But there are some subtle aspects to numbers that it's worth thinking about. We think of numbers alnog a line like this:



You are told that "almost all" of the numbers on this line are irrational. That means if you throw a dart at the line you should never "hit" a rational number. The irrationals fill the entire line.

But there is a paradox:

No one has ever met a true irrational number in person. We hear a lot of name droping. People say things like, "I know π and e." They are big celebrities in some circles. But no one's ever really seen them in their total infinity. Only fleeting glimpses as they run down the street and jump into a limo.

My person view: Irrational numbers are a convinent fiction. They are "defined" as the completion of the rationals under limits of Cauchy sequences. What?

Say you have a sequence of rationals (ie "fractions" or ratios of integers), $r_0, r_1, r_2, \ldots, r_n, r_{n+1}, \ldots$. And say you have a way of comparing the distance between any two numbers,

$$|r_n - r_m|.$$

Now say that for any tiny number you pick, say 1/1,000,000,000, or 1/1,000,000,000,000, or 10^{-16}. You can always find an n and and m where

$$|r_n - r_m| < 10^{-16}.$$

And it *never* get bigger than that for any numbers higher than n, m.

And if someone had said $10^{-\text{googolplex}}$, we would have been able to find an n and m for that too.

We call this kind of sequence of rational numbers a ***Cauchy sequence***. It looks like it's going somewhere. But it's just a bunch of rational numbers at every step of the way.

The thing about these kinds of sequences is that there may not be a rational number at the end of it. The definition is to just make a bigger set of numbers that includes these limits that you can never get to. After that we just go on our way as if nothing was ever awkward.

Here's an example of one such sequence.

Take $r_0 = 1$. At every step of the sequence, define

$$r_{n+1} = \frac{r_n + 2}{r_n + 1}$$

Nothing but a bunch of rationals all the way... It's also possible to prove that the terms get closer and closer together.

Continued fractions

What you are seeing is an example of a ***continued fraction*** sequence.

$$1 + \cfrac{1}{2 + \cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2\cdots}}}}}}}$$

Continued fractions are a "natural" way of both rational and irrational numbers as sequences of integers. To know what I mean by "natural", it helps to understand how the decimal representations we use every day can be weird sometimes.

The decimal system

$$x = \lim_{D\to\infty} \sum_{i=-D}^{D} d_i 10^i,$$

with

$$d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

Notice that I took the limit as the range of terms $[-D, +D]$ goes to ∞. This is becuase we can't really ever get everyting in one place. The Decimal system is just one more way of looking at sequences of rational numbers. And we know there are some drawbacks to this also.

Numbers like $1/3$ have decimal representations that go on repeting forever. If you want to represent $1/3$ exactly you need to use a base-3 system.

But we still need to calculate with numbers like π and $1/3$. But we cannot fit them in our finite computers. This leads to one of the two main sources of error in computations.

Decimal and binary system (and all other integer-base systems) have the nice property that they are (almost) "unique". That is if any two numbers have the same binary or decimal expansion, then they are the same number.

There is one important exception to this. In decimal, the number

$$u = 0.9999999999999999999 \text{ (repeting)}$$

is not it's own number. This number is equal to $u = 1$.

There are a lot of clever ways to prove this. For example:

$$\frac{u}{10} = 0.0999999999999999999 \text{ (repeting)}$$

$$u - \frac{u}{10} = 0.9 = \frac{9}{10}$$

$$u = 1$$

Weird things like this happen when dealing with infinity. This is related to the "Hilbert's Hotel" paradox:

https://en.wikipedia.org/wiki/Hilbert's_paradox_of_the_Grand_Hotel (https://en.wikipedia.org/wiki/Hilbert's_paradox_of_the_Grand_Hotel)

My opinion is that it's best to just consider finite things if you ever really need to make sure you are doing things right.

In fact, whenever possible, keep everything *finite* as long as possible. Take limits after everything is done. People think something deep is going on when playing with infinities. It's just parlour tricks. Taking $x = -1$ in the infinite series produces the *Eilenberg--Mazur swindle*:

$$1 \;=\; 1 + (-1 + 1) + (-1 + 1) + \ldots \;=\; 1 - 1 + 1 - 1 + \ldots \;=\; (1 - 1) + (1 - 1) + \ldots \;=\; 0$$

Changing the location of the $(\ )$ "proves" that $1 = 0$; you can use this to prove any other absurd thing you want. The resolution to this comes from considering the finite version.

$$\sum_{k=0}^{n-1} (-1)^k \;=\; \begin{cases} 1 & \text{if } n \text{ is even} \\ 0 & \text{if } n \text{ is odd} \end{cases}$$

This doesn't have a well-defined limit. Although it is possible to make sense of the sum in some specific cases. In those cases, it's usally clear what needs to be done.

The *GREAT* thing about a computer is you can *NEVER* make anything *ACTUALLY INFINITY*.

All you can ever hope for

Continued fractions are a clean way of defining numbers without reference to a (non-unique) base system. It also avoids practical and fundemantal issues with repeting sequences.

We use the following notation to represent continued fraction sequences

$$[a_0; a_1, a_2, a_3, \ldots] \equiv a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cfrac{1}{a_5 + \cfrac{1}{a_6 + \cfrac{1}{a_7 \cdots}}}}}}}$$

For $i \geq 1$, all of the a_is are positive integers; a_0 can be a positive or negative integer.

https://en.wikipedia.org/wiki/Continued_fraction (https://en.wikipedia.org/wiki/Continued_fraction)

```
* The continued fraction representation for a rational number is finite and only ra
tional numbers have finite  representations. In contrast, the decimal representatio
n of a rational number may be finite.
```

For example: $137/1600 = 0.085625$, or infinite with a repeating cycle, for example $4/27 = 0.148148148148\ldots$.

```
* Every rational number has an essentially unique continued fraction representatio
n. Each rational can be represented in exactly two ways.
```

Since $[a_0; a_1, \ldots a_{n-1}, a_n] = [a_0; a_1, \ldots a_{n-1}, (a_n - 1), 1]$. Usually the first, shorter one is chosen as the *canonical representation*.

```
* The continued fraction representation of an irrational number is unique.

* The real numbers whose continued fraction eventually repeats are precisely the qu
adratic irrationals.
```

For example: the repeating continued fraction

$$\phi = \frac{1 + \sqrt{5}}{2} = [1; 1, 1, 1, \ldots]$$

is the golden ratio.

The repeating continued fraction

$$\sqrt{2} = [1; 2, 2, 2, \ldots]$$

is the square root of 2.

In contrast, the decimal representations of quadratic irrationals are apparently random. The square roots of all (positive) integers, that are not perfect squares, are quadratic irrationals, hence are unique periodic continued fractions.

For example:

$$\pi = [3; 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, \ldots]$$

$$e = [2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1, \ldots]$$

* The successive approximations generated in finding the continued fraction representation of a number, i.e. by truncating the continued fraction representation, are in a certain sense (described below) the "best possible".

For example:

$$\pi \approx 3 \approx \frac{22}{7} \approx \frac{333}{106} \approx \frac{355}{113} \approx \frac{103993}{33102}$$

The errors are:

$$\frac{|\pi - \pi_n|}{\pi} = 4.5 \times 10^{-2},\ 4.0 \times 10^{-4},\ 2.6 \times 10^{-5},\ 8.5 \times 10^{-8},\ 1.8 \times 10^{-10}$$

By definiton the ith decimal trunctaion is only accurate to $\approx 10^{-i}$.

The golden ratio is the *"most irrational number"*. That is, it has the slowest converging continued fraction sequence. This is the reason it shows up in a lot of applications. Sometimes (like in plants) it's best to have things as far away from rational ratios as possible.

The iteration that generates the golden ratio continued fraction sequence is

$$r_{n+1} = \frac{r_n + 1}{r_n}$$

And $r_0 = 1$. It looks a lot like the sequence that generates $\sqrt{2}$. The limiting case is where the iterates stop changing. Ie, $r_{n+1} = r_n$. Solving

$$\phi = \frac{1 + \phi}{\phi}$$

indeed gives the expected answer.

In [1]:
```
import numpy as np

phi = (1+np.sqrt(5))/2
print(phi)
```

```
1.618033988749895
```

Here is the simplest way to express the recursion in code:

In [2]:
```
def f(n):
    if n == 0: return 1
    return (f(n-1)+1)/f(n-1)
```

Try different values and see how it behaves:

This should happen fast:

In [3]:
```
f(1)
```

Out[3]: 2.0

You might notice that this doesn't happen right away:

In [4]:
```
f(10)
```

Out[4]: 1.6179775280898876

This will take long enough to really notice

In [5]:
```
f(23)
```

Out[5]: 1.618033988957902

It wouldn't be wise to go much further. But the question is why? Are recursive function really not a good idea in general. Are they always doomed to slowness?

`print` statements are the go-to way to debugg code. This will help you figgure out what's going on, and maybe suggest a way to fix it:

In [6]:
```
def f(n):
    if n == 0: return 1
    print('n:',n)
    return (f(n-1)+1)/f(n-1)
```

In [7]:
```
f(1)
```

```
n: 1
```

Out[7]: 2.0

In [8]:
```
f(2)
```

```
n: 2
n: 1
n: 1
```

Out[8]: 1.5

In [9]:
```
f(3)
```

```
n: 3
n: 2
n: 1
n: 1
n: 2
n: 1
n: 1
```

Out[9]: 1.6666666666666667

We can see the issue. Each level n, is calling each lower level twice.

n=1 --> 1 calls

n=2 --> 3 calls

n=3 --> 7 calls

n=4 --> 15 calls

n=k --> (2**k - 1) calls

This is an **exponential** increase in computation with each level. BAD!

Using memory to save computation

But it doesn't need to be this way. A little care saves a lot of effort. The idea is to use a little bit of "memory" to store the value of $f(n-1)$ and use that value two times in the return statement.

In [10]:
```
def f(n):
    if n == 0: return 1
    print('n:',n)
    temp = f(n-1)
    return (temp+1)/temp
```

Now it should behave much better:

In [11]:
```
f(1)
```

```
n: 1
```

Out[11]: 2.0

In [12]:
```
f(2)
```

```
n: 2
n: 1
```

Out[12]: 1.5

In [13]:
```
f(10)
```

```
n: 10
n: 9
n: 8
n: 7
n: 6
n: 5
n: 4
n: 3
n: 2
n: 1
```

Out[13]: 1.6179775280898876

In [14]:
```
f(100)
```

```
n: 100
n: 99
n: 98
n: 97
n: 96
n: 95
n: 94
n: 93
n: 92
n: 91
n: 90
n: 89
n: 88
n: 87
n: 86
n: 85
n: 84
n: 83
n: 82
n: 81
n: 80
n: 79
n: 78
n: 77
n: 76
n: 75
n: 74
n: 73
n: 72
n: 71
n: 70
n: 69
n: 68
n: 67
n: 66
n: 65
n: 64
n: 63
n: 62
n: 61
n: 60
n: 59
n: 58
n: 57
n: 56
n: 55
n: 54
n: 53
n: 52
n: 51
n: 50
n: 49
n: 48
n: 47
n: 46
n: 45
n: 44
```

```
n: 43
n: 42
n: 41
n: 40
n: 39
n: 38
n: 37
n: 36
n: 35
n: 34
n: 33
n: 32
n: 31
n: 30
n: 29
n: 28
n: 27
n: 26
n: 25
n: 24
n: 23
n: 22
n: 21
n: 20
n: 19
n: 18
n: 17
n: 16
n: 15
n: 14
n: 13
n: 12
n: 11
n: 10
n: 9
n: 8
n: 7
n: 6
n: 5
n: 4
n: 3
n: 2
n: 1
```

```
Out[14]: 1.6180339887498947
```

There absolutely no way we would ever have been able to computer $f(100)$ using the other method. Memeory is a way to avoid redundant computations. And sometimes this is a huge deal.

Rewriting expressions to save computation

In the current problem you don't really need to use the temp variable. You could just reformulate the `return` statement to only call `f(n-1)` one time. This is not always easy.

In [15]:
```
def f(n):
    if n == 0: return 1
    print('n:',n)
    return 1 + (1/f(n-1))

f(30)
```

```
n: 30
n: 29
n: 28
n: 27
n: 26
n: 25
n: 24
n: 23
n: 22
n: 21
n: 20
n: 19
n: 18
n: 17
n: 16
n: 15
n: 14
n: 13
n: 12
n: 11
n: 10
n: 9
n: 8
n: 7
n: 6
n: 5
n: 4
n: 3
n: 2
n: 1
```

Out[15]: 1.6180339887496482

The big problem with continued fractions is they are really hard to compute arithmitic with. It's a lot like Roman Numerials. The best way to do it is to convert them into decimals, add, subtract, multiply, divide, and then convert them back. That is why we use floating-point numbers.

Floating Point Numbers

Throughout this class, we will need to work with computer representations of real numbers: we'll be adding, subtracting, dividing, and multiplying them in order to solve complex problems regarding the world. As an example, The observable physical universe covers length scales from the size of the proton, $l_p \sim 10^{-14}$ m,* to the size of the observable universe, $L_u \sim 10^{27}$ m. Taking the ratio of $L_u/l_p \simeq 10^{41}$, we see there is a factor of 10^{41} between the smallest and largest length scales.

In order to cover quantities over a large "dynamic range", we typically use scientific notation in our paper-and-pencil work, and computers do very much the same thing. The only tricky part is that the number, which we like to represent in decmial form, is stored in binary.

Let's analyse scientific notation. Take an interesing small number. For example, the mass of an electron $\mu = 9.10938356 \times 10^{-31}\text{kg}$.

We can write this schematically as

$$\mu = (-1)^s\, m\ \times 10^e,$$

where $s = 0$ is called the *sign*, $m = 9.10938356$ the ***mantissa*** (or significant), and $e = -31$ the ***exponent***.

Of course, the computer stores the number μ in binary, so it must be converted. We'll use the notation N_{10} to mean a number N in base 10, and N_2 to mean a number in base 2. For example $10_{10} = 1010_2$.

IEEE 754

There are many ways of storing floating point numbers on a computer; we will only describe one: IEEE 754. This is the most widely used standard for representing these numbers. An IEEE 754 number is represented in the following way:

$$x = (-1)^s\, 1.f\ \times 2^{e-B},$$

where s is the sign, f is the *fractional part* of the mantissa (note that there is 1 in front), e is the exponent of the number you want to represent, and B is called the *bias*. Note that the total exponent stored in memory e has no sign bit and is thus **always positive**. This explains the need for the bias: if we want to represent actual exponents $p = e - B < 0$, $B > 0$. A floating point number standard that can't hold the mass of an electron is not very useful to scientists. The bias $B = 127$ for single precision, and

IEEE 754 defines two different kinds of floating point numbers: single and double precision. Single precision floating point numbers are called `floats` and take up 32 bits of memory, and double precision is called `double` and take up 64 bits of memory.

Unfortunately, python doesn't respect this naming convention, chosing instead to use `float` to mean a 64-bit number, and `float32` to mean a 32-bit number. This is only mildly annoying in practice, since you'll probably never need a 32-bit float in this class. Computer 15 years ago all used 32-bit number as defaults. They pretty much all switched over at some point to using 64-bit defaults. Maybe on day, this will change to 128-bit. But for now that is also an option that will slow down computations.

We'll use 32-bit single precision as our example here, even though you will almost always work in double precision. A 32 bit `float` uses 1 bit to hold the sign s, 8 bits to hold the exponent e, and the remaining 23 bits to