π

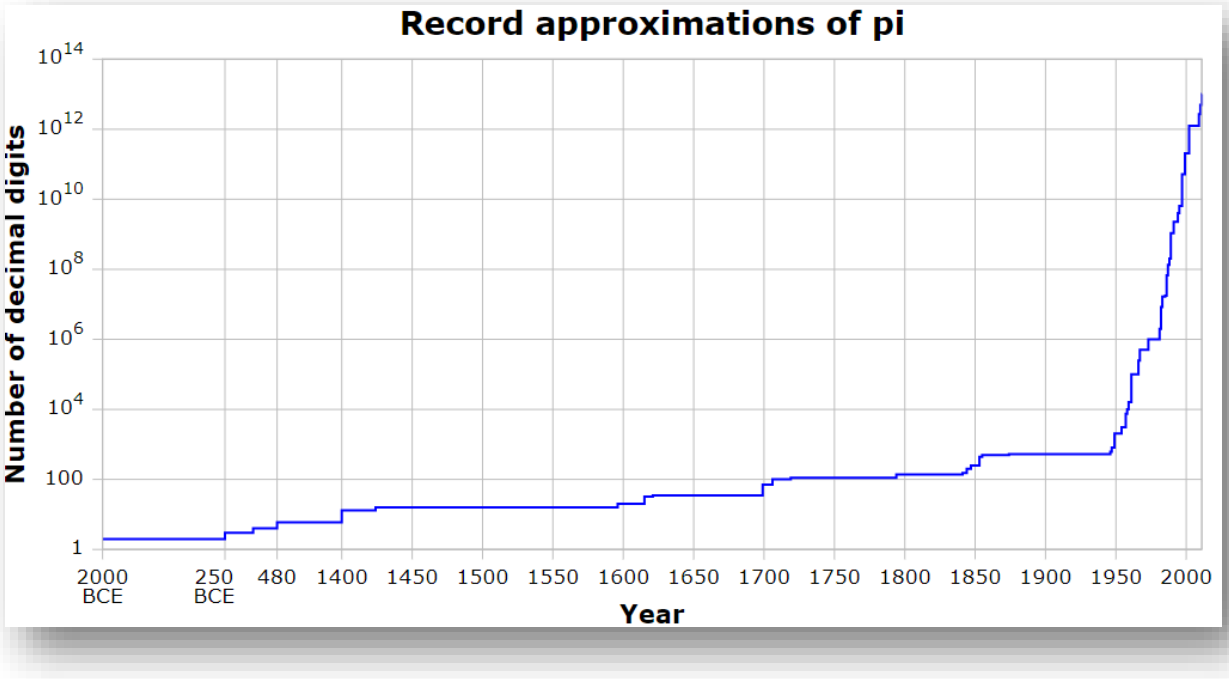
When Isaac Newton was sent home from college to self-isolate during the plague sweeping England in 1665, he amused himself by calculating π to 15 decimal places, by hand, using the new-found techniques of calculus and binomial expansions. He said: "I am ashamed to tell you to how many figures I carried these computations, having no other business at the time."

Why π?

The irrational mathematical constant which most of us associate most directly with circles crops up all over the place in maths. And since physicists, astronomers, physicists, etc like to use maths to do things with things, it can be really helpful to know precisely what it is. Ever since 1665, we've had sufficient precision to calculate the circumference of a circle the size of our entire solar system to within $1mm$. Good enough for NASA, apparently:

https://www.jpl.nasa.gov/edu/news/2016/3/16/how-many-decimals-of-pi-do-we-really-need/

So why has mankind devoted such a tremendous amount of effort finding the value of π to such a ludicrous level of precision? We passed a million digits in 1973, and we're still at it:



source:Wikipedia

One of the main reasons people like to compute π is that it provides a way to compare the efficiency of computing systems, or – still better – the algorithms that are used. What better way to flex your mathematical muscles than to churn out a few trillion digits of π in a fraction of the time that humanity has previously needed to find a mere few billion?

Similar reasons lie behind the human memorisation of the digits, I assume (although if you're going to devote your life to the accurate recitation of over $100{,}000$ digits, you may need to come up with a better rationale for yourself than "because I can").

Mathematicians have never needed much motivation beyond "I wonder if I can…" to wrestle with a problem, and finding ever more digits of π, and ever more efficient ways to calculate them, has been a favourite pastime of mathematicians for centuries.

The Dartboard Method

It's not the most efficient method, but it is one of the easiest to make sense of: throw darts at random onto a dartboard, and you should expect roughly 314 out of every 400 that land in the square to also lie in the circle.

Throwing your darts

We can use the **random** module to generate a random position on the dart board, and Pythagoras to determine whether or not a given dart landed in the circle.

```
1   import random
2
3   on_target = 0
4   num_darts = 0
5   while True:
6       num_darts += 1
7       x = random.uniform(-1, 1)
8       y = random.uniform(-1, 1)
9       if x**2 + y**2 < 1:
10          on_target += 1
11      pi = 4 * on_target / num_darts
12      print(f"{num_darts} darts:  {pi}")
```

The **random.uniform** function generates a decimal chosen uniformly (evenly) from the range.
This loop continues indefinitely, throwing darts and tallying the total and the number on target.
On each run through the loop, a print statement is generated.
Note that, since a circle in a square takes up $\frac{\pi}{4}$ of the area, we multiply our proportion by 4 to approximate π.

Notice that sometimes the approximation will be very accurate early on just by fluke (eg if 11 of the first 14 darts are on target), but rest assured, even if it deviates somewhat as time goes by, it will eventually home in on the true value.

Improvement option 1 - Faster

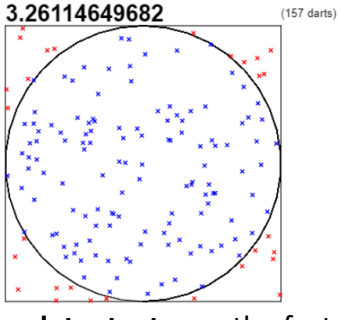
Printing the value every single time is a significant drain on computational resources. Adapt your program to show you the result after it has completed a million iterations, or – for the best of both worlds – have it calculate its approximation and update the user on the value every 100,000 iterations (hint: use the $\%$ operator to check when the number of darts is a multiple of $100{,}000$).

Improvement option 2 - Prettier

It's all well and good to see the approximation gradually improving over time, but it would be nice to see the dart board. Make a **turtle** version of the program that actually shows each dart as it lands. *Here's a few snippets of code to get you started:*

```
on_target, num_darts = 0, 0
while True:
    x = random.uniform(-r, r)
    y = random.uniform(-r, r)
    dart.goto(x, y)
    if x**2 + y**2 < r**2:
        cross("blue")
        on_target += 1
    else:
        cross("red")
    num_darts += 1
    pi = 4 * on_target / float(num_darts)
    # turtle does integer division by default!
    update_text(pi)
```

I created a turtle called **dart**, and a separate function for drawing a cross.
I also made a turtle called **scribe** whose job it is to rub out the last value written and update with the latest:



update_text uses the fact that turtles can write with the **.write** method as shown below:

```
scribe.write("(" + str(total) + " darts)", font=("Arial", 10, "normal"))
```

Iterative Processes

There are many, many sequences which converge to something involving π. I've listed a few here. They're nice partly because forming a suitable loop and ensuring the numbers match is a satisfying challenge. It's also interesting to pit one against another and see which ones converge to a decent approximation the fastest.

Infinite sums

Recall that **+=** lets you add a given value to your variable, allowing you to 'accumulate' numbers as you go along.

James Gregory (1671): $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$

Nilakantha (15th century): $\frac{\pi-3}{4} = \frac{1}{2\times3\times4} - \frac{1}{4\times5\times6} + \frac{1}{6\times7\times8} - \cdots$

Leonard Euler (1735): $\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots$

Infinite products

Just like the **+=** operator, you can use ***=** to multiply your variable by a given value.

François Viète (1593): $\frac{2}{\pi} = \frac{\sqrt{2}}{2} \times \frac{\sqrt{2+\sqrt{2}}}{2} \times \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \times \ldots$

John Wallis (1655): $\frac{\pi}{2} = \left(\frac{2}{1}\times\frac{2}{3}\right)\left(\frac{4}{3}\times\frac{4}{5}\right)\left(\frac{6}{5}\times\frac{6}{7}\right)\left(\frac{8}{7}\times\frac{8}{9}\right)\ldots$

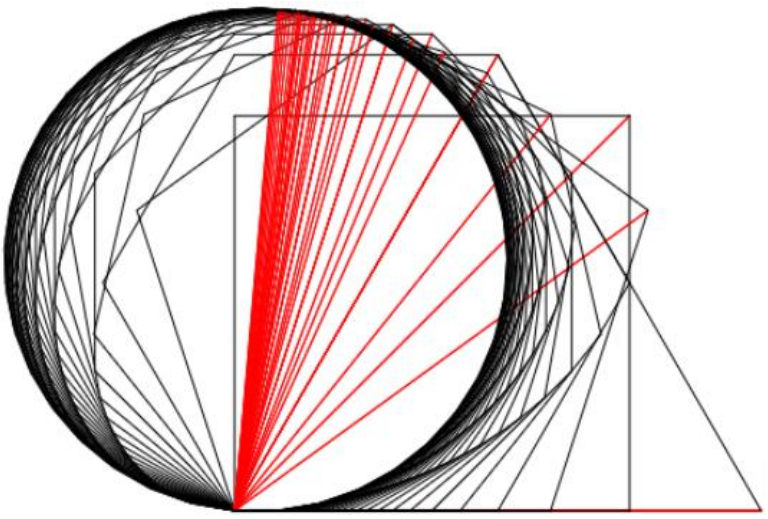
Spigot algorithms

Incredibly, there are algorithms for efficiently computing a single digit of the (hexadecimal) expansion of π, *without needing to compute the preceding digits*. These are used to verify the results of other algorithms which claim to compute π to a greater level of precision than ever before.

Ancient methods

Archimedes famously approximated π by constructing regular polygons with ever-more sides and comparing them to the circles they approach.

We can do something similar with the Python turtle, by having him draw us a polygon with a fixed perimeter, but stopping halfway to measure the distance to the start, and divide one by the other.



Unbelievably efficient algorithm

The final algorithm is absolutely mind-blowing. It is so efficient that you can essentially 'run the code' yourself with a simple calculator and a pen and paper:

```
1   a = 1
2   b = 1 / 2**0.5
3   c = 1 / 4
4   x = 1
5   for i in range(3):
6     y = a
7     a = (a + b) / 2
8     b = (b * y)**0.5
9     c -= x * (a - y)**2
10    x *= 2
11    print((a + b)**2 / (4 * c))
```

To 'trace' this algorithm, you'll need to keep track of the values of **a**, **b**, **c**, **x** and **y**. You can use the 'memory slots' provided!

Each time you go through the loop, make a note of the value of $\frac{(a+b)^2}{4c}$, as indicated by the **print** statement. That way, you'll see your approximation improving each time.

Memory Slots: Use these spaces to record and update, the values of **a**, **b**, **c**, **x**, and **y**:

a =	**b =**	**c =**
x =	**y =**	**print…**

*This algorithm gives a level of precision that improves so fast, it takes fewer than 20 iterations to achieve over a million decimal places of π! If you want to see more of them than the 15 or so that Python usually shows, you might find the **Decimal** module useful:*

```
1   from decimal import *
2   getcontext().prec = 100
3
4   a = Decimal(1)
5   b = Decimal(1) / Decimal(2).sqrt()
6   c = Decimal(1) / Decimal(4)
7   x = Decimal(1)
8   for i in range(5):
9     y = a
10    a = (a + b) / Decimal(2)
11    b = Decimal(b * y).sqrt()
12    c -= x * (a - y)**Decimal(2)
13    x *= Decimal(2)
14    pi = (a + b)**Decimal(2) / (Decimal(4) * c)
15    print(pi)
```

This code looks somewhat messier: our numbers are stored as **Decimal** objects rather than **Float** or **Int**, and any operations we use have to be done in that module.

But it's totally worth it. Line 2 sets the precision to 100 decimal places, which you may think is a bit over-kill, but run the loop just 5 times and see if you still think so…