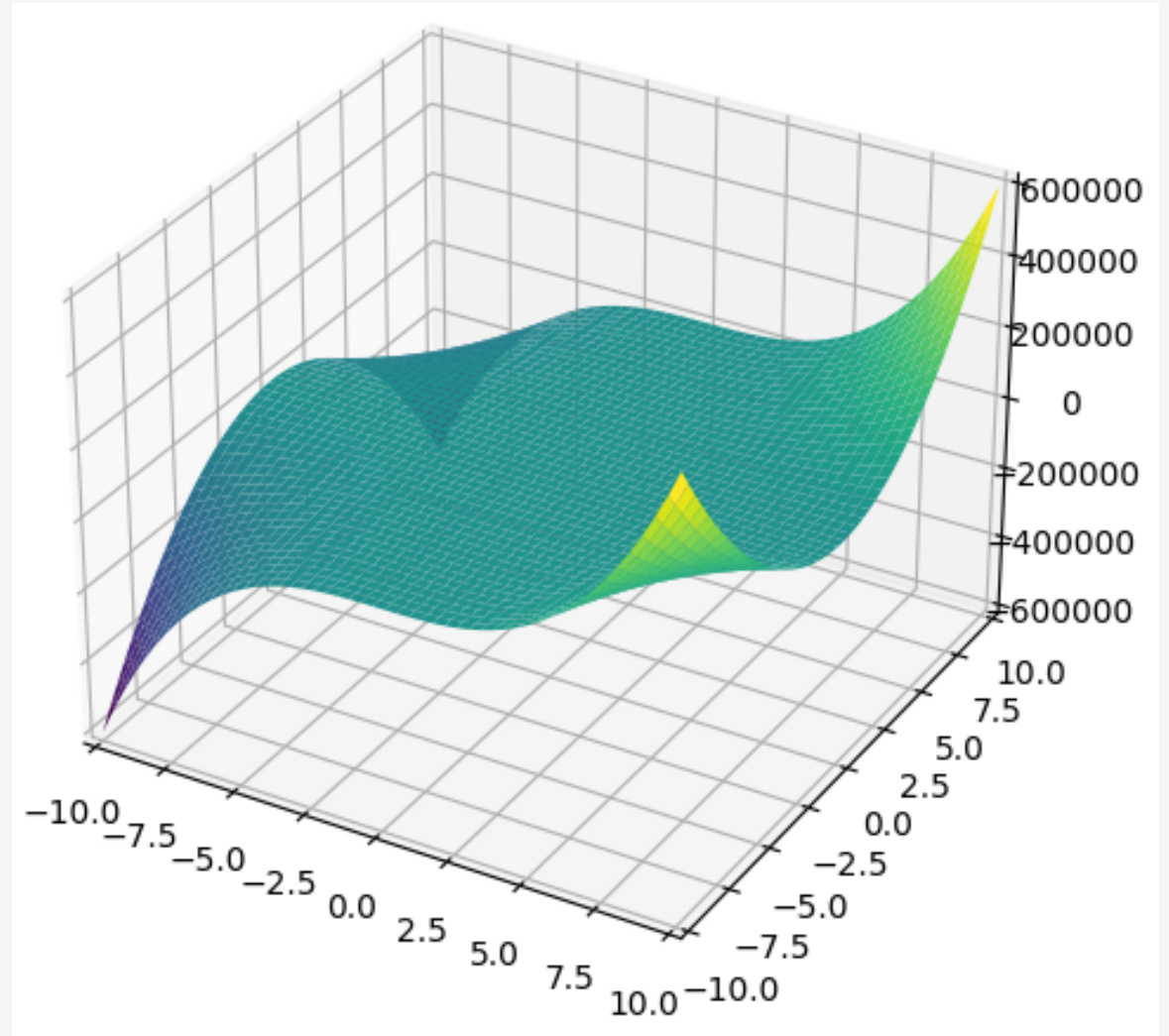# Calculus and Functions

Thomas Nield

O'Reilly Media

What to Expect

1. Number Theory
2. Expressions and Functions
3. Derivatives and Partial Derivatives
4. Gradient Descent
5. Integrals

What You'll Need

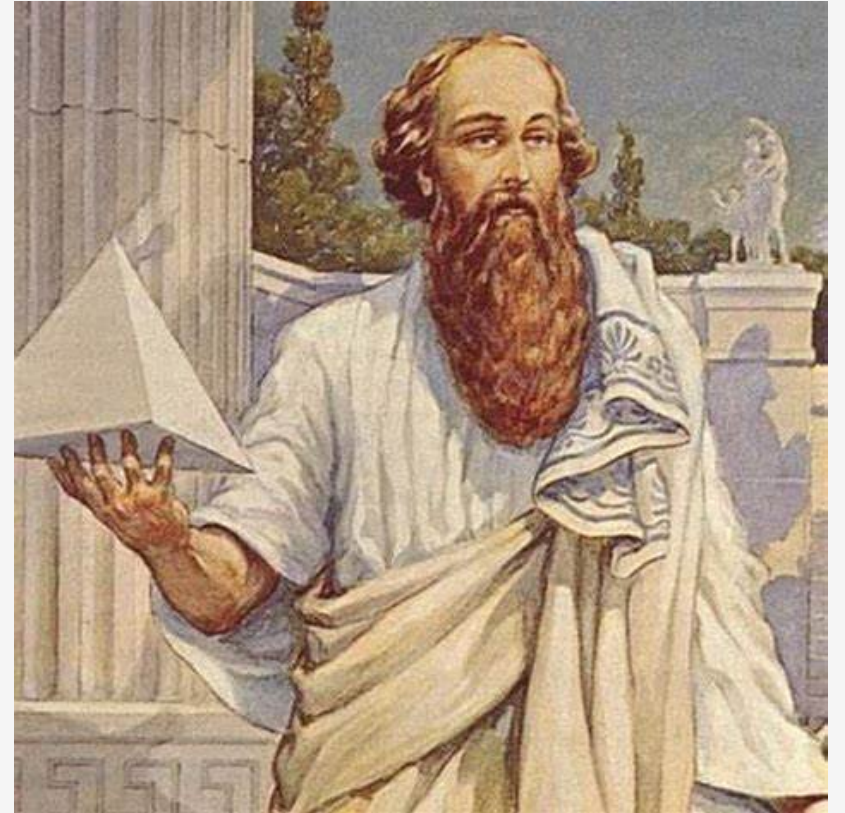**Python 3.x** - Latest Python 3 release recommended in environment of

choice.

SymPy – Symbolic math library for Python.

Matplotlib – Enables charting functionality for SymPy
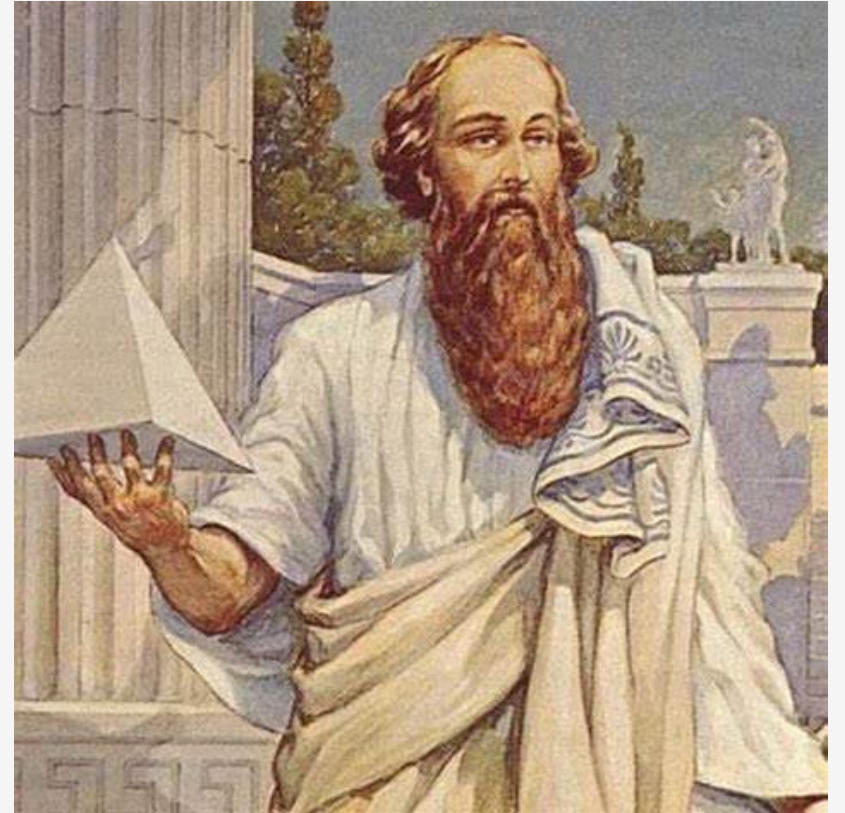
Section I

Mathematical Functions

What Are Numbers?

Pythagoras was an important philosopher in creating number systems, taking his beliefs as as far as a religious system.

What Are Numbers?

We will avoid being too philosophical with this class, but are numbers are not a construct we have defined?

Pythagoras was an important philosopher in creating number systems, taking his beliefs as as far as a religious system.

What Are Numbers?

We will avoid being too philosophical with this class, but are numbers are not a construct we have defined?

Why do we have the digits 0 through 9 and not have more digits?

Pythagoras was an important philosopher in creating number systems, taking his beliefs as as far as a religious system.

What Are Numbers?

We will avoid being too philosophical with this class, but are numbers are not a construct we have defined?
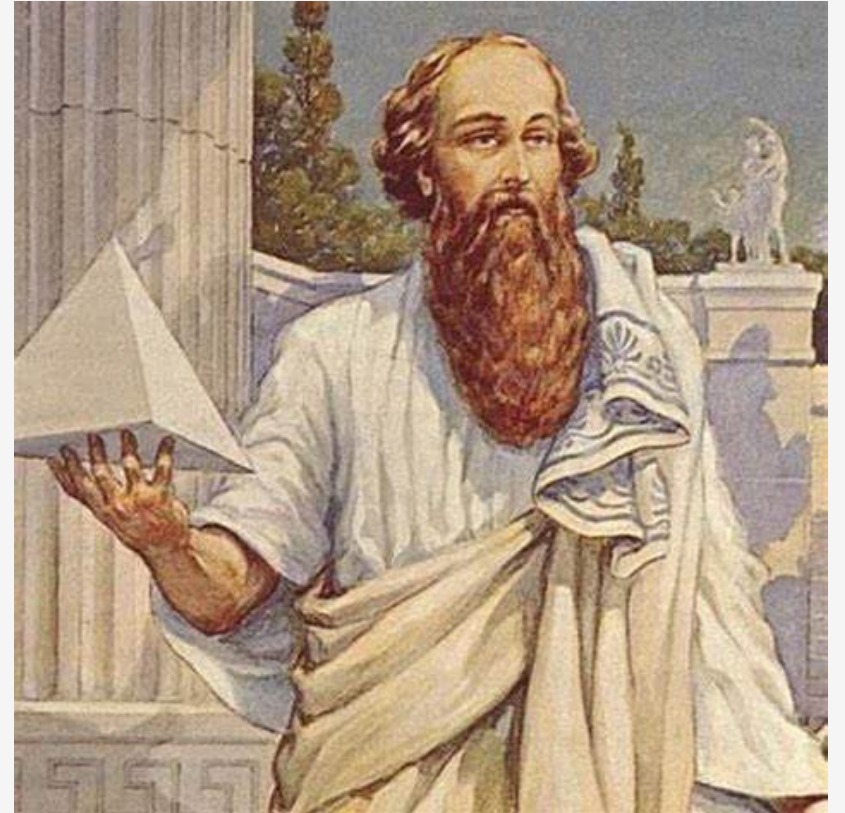
Why do we have the digits 0 through 9 and not have more digits?

Why do we have fractions and decimals and not just whole numbers?

Pythagoras was an important philosopher in creating number systems, taking his beliefs as as far as a religious system.

What Are Numbers?

We will avoid being too philosophical with this class, but are numbers are not a construct we have defined?

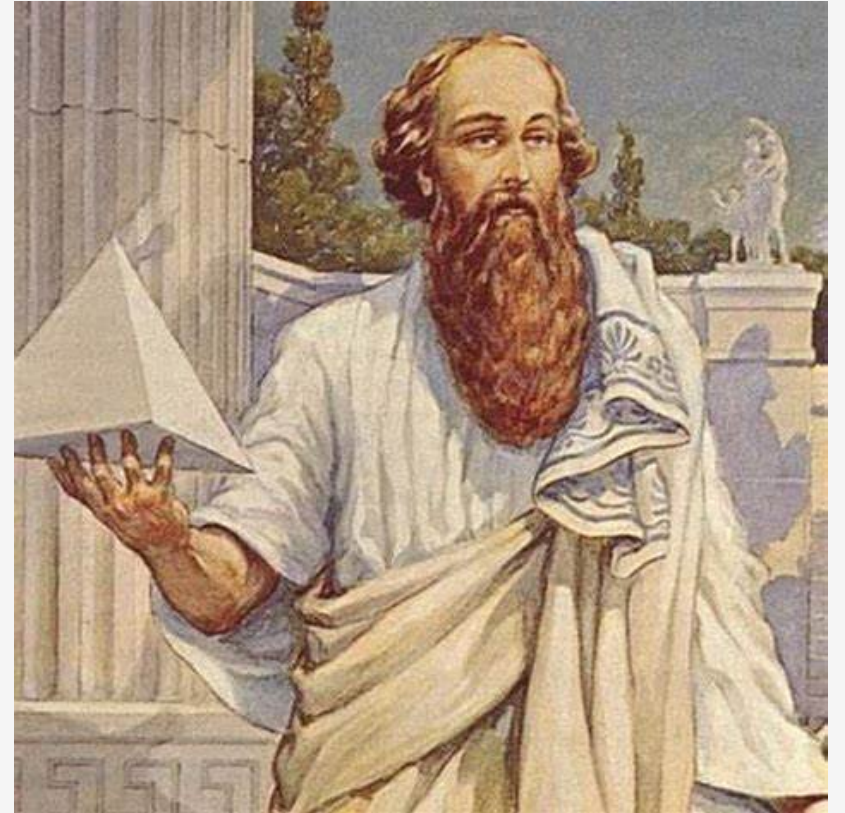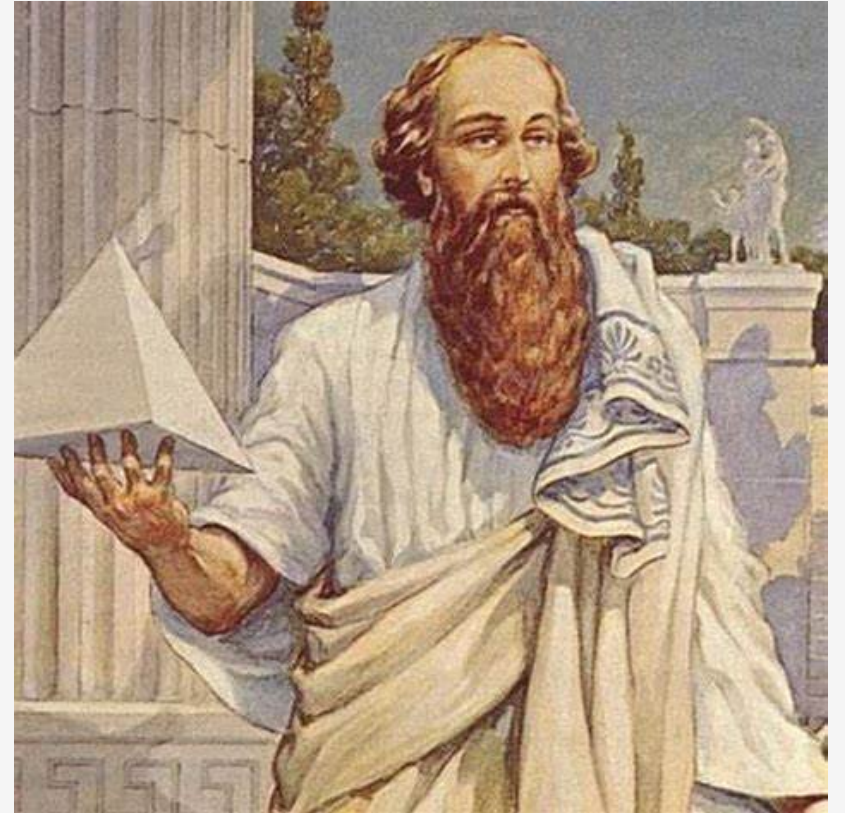Why do we have the digits 0 through 9 and not have more digits?

Why do we have fractions and decimals and not just whole numbers?

Why do negative numbers exist?

Pythagoras was an important philosopher in creating number systems, taking his beliefs as as far as a religious system.

What Are Numbers?

We will avoid being too philosophical with this class, but are numbers are not a construct we have defined?

Why do we have the digits 0 through 9 and not have more digits?

Why do we have fractions and decimals and not just whole numbers?

Why do negative numbers exist?

Number Theory goes all the way back to ancient times, where mathematicians study different number systems and why we have accepted them the way we do today.

Pythagoras was an important philosopher in creating number systems, taking his beliefs as as far as a religious system.
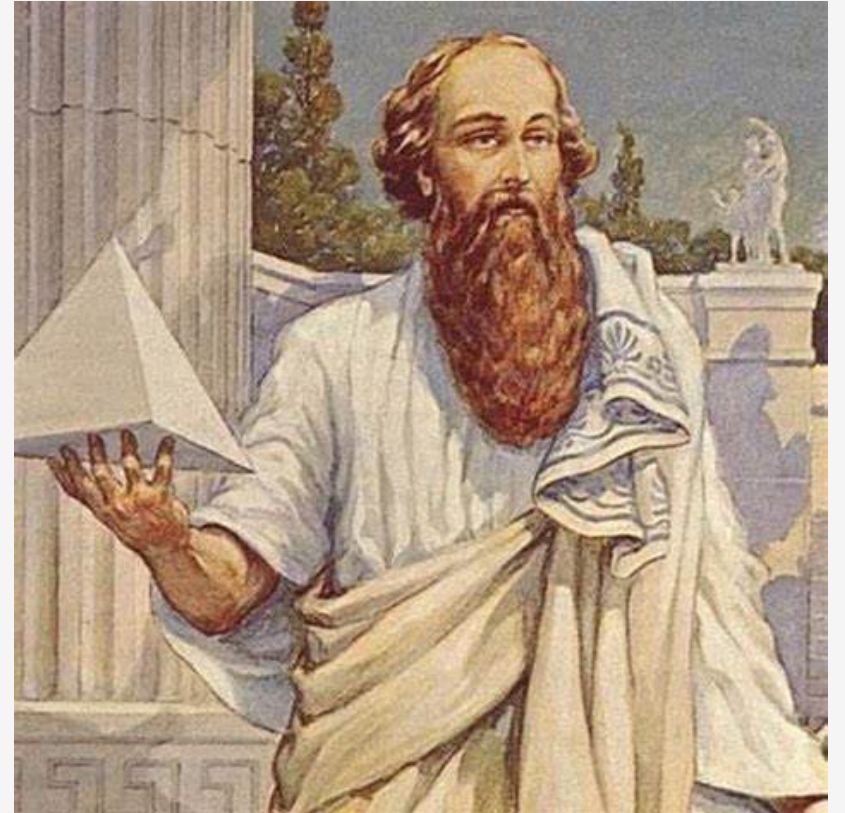
What Are Numbers?

We will avoid being too philosophical with this class, but are numbers are not a construct we have defined?

Why do we have the digits 0 through 9 and not have more digits?

Why do we have fractions and decimals and not just whole numbers?

Why do negative numbers exist?

Number Theory goes all the way back to ancient times, where mathematicians study different number systems and why we have accepted them the way we do today.

There are several number systems you may recognize, and they are key to understanding mathematical functions and applied mathematics in general.
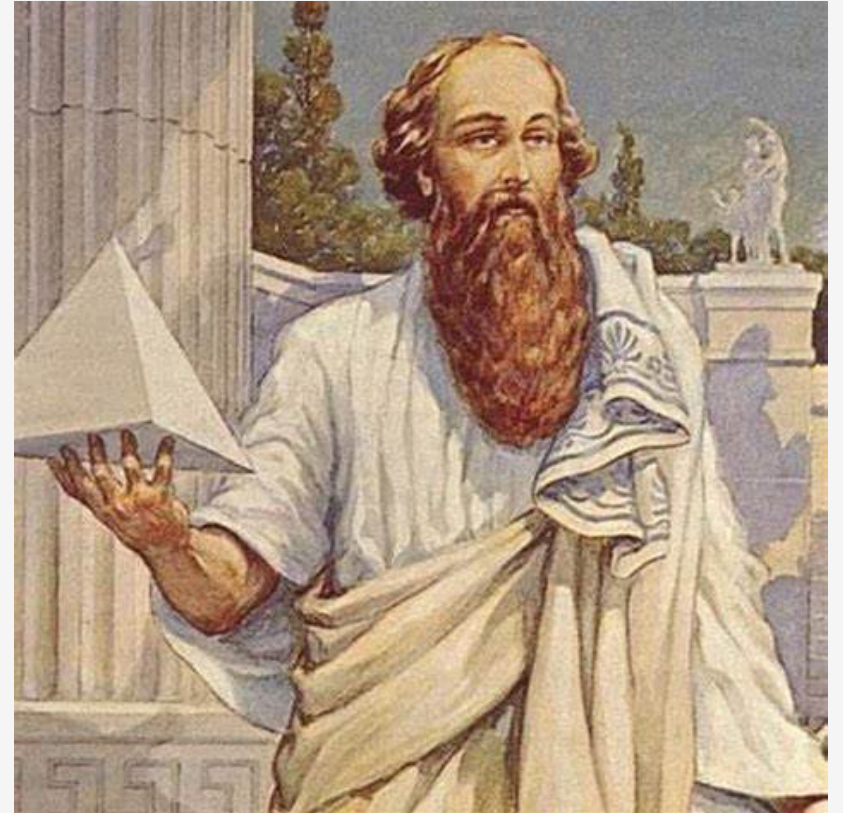
Pythagoras was an important philosopher in creating number systems, taking his beliefs as as far as a religious system.

Number Systems – Natural Numbers


```
# Print natural numbers 0 through 99
for i in range(0, 100):
    print(i)
```

The ancient Ishango bone is theorized to have tally marks scratched on it, keeping quantitative record of something.
https://en.wikipedia.org/wiki/Ishango_bone

Number Systems – Natural Numbers

Natural numbers are 0, 1, 2, 3, 4, 5, and so on...


```
# Print natural numbers 0 through 99
for i in range(0, 100):
    print(i)
```

The ancient Ishango bone is theorized to have tally marks scratched on it, keeping quantitative record of something.
https://en.wikipedia.org/wiki/Ishango_bone

Number Systems – Natural Numbers

Natural numbers are 0, 1, 2, 3, 4, 5, and so on…

$$\mathbb{N} = 0,\ 1,\ 2,\ 3,\ 4 \ldots$$

The ancient Ishango bone is theorized to have tally marks scratched on it, keeping quantitative record of something.
https://en.wikipedia.org/wiki/Ishango_bone

```
# Print natural numbers 0 through 99
for i in range(0, 100):
    print(i)
```

Number Systems – Natural Numbers

Natural numbers are 0, 1, 2, 3, 4, 5, and so on…

$$\mathbb{N} = 0,\ 1,\ 2,\ 3,\ 4\ \ldots$$

These are zero and positive whole numbers and are the oldest number system, going back to ancient cave people scratching tally marks on bones and walls to keep records.

The ancient Ishango bone is theorized to have tally marks scratched on it, keeping quantitative record of something.
https://en.wikipedia.org/wiki/Ishango_bone

```
# Print natural numbers 0 through 99
for i in range(0, 100):
    print(i)
```

Number Systems – Natural Numbers

Natural numbers are 0, 1, 2, 3, 4, 5, and so on…

$$\mathbb{N} = 0,\ 1,\ 2,\ 3,\ 4 \ldots$$

These are zero and positive whole numbers and are the oldest number system, going back to ancient cave people scratching tally marks on bones and walls to keep records.

The concept of "0" was later accepted, and the Babylonians developed the useful idea for place-holding notation for empty "columns" on numbers 0", "1090", or "1090."

```
# Print natural numbers 0 through 99
for i in range(0, 100):
    print(i)
```


The ancient Ishango bone is theorized to have tally marks scratched on it, keeping quantitative record of something.
https://en.wikipedia.org/wiki/Ishango_bone

Number Systems – Natural Numbers

Natural numbers are 0, 1, 2, 3, 4, 5, and so on...

$$\mathbb{N} = 0, \ 1, \ 2, \ 3, \ 4 \ldots$$

These are zero and positive whole numbers and are the oldest number system, going back to ancient cave people scratching tally marks on bones and walls to keep records.

The concept of "0" was later accepted, and the Babylonians developed the useful idea for place-holding notation for empty "columns" on numbers ~~, such as "10", "1000", or "1090."~~

```
# Print natural numbers 0 through 99
for i in range(0, 100):
    print(i)
```
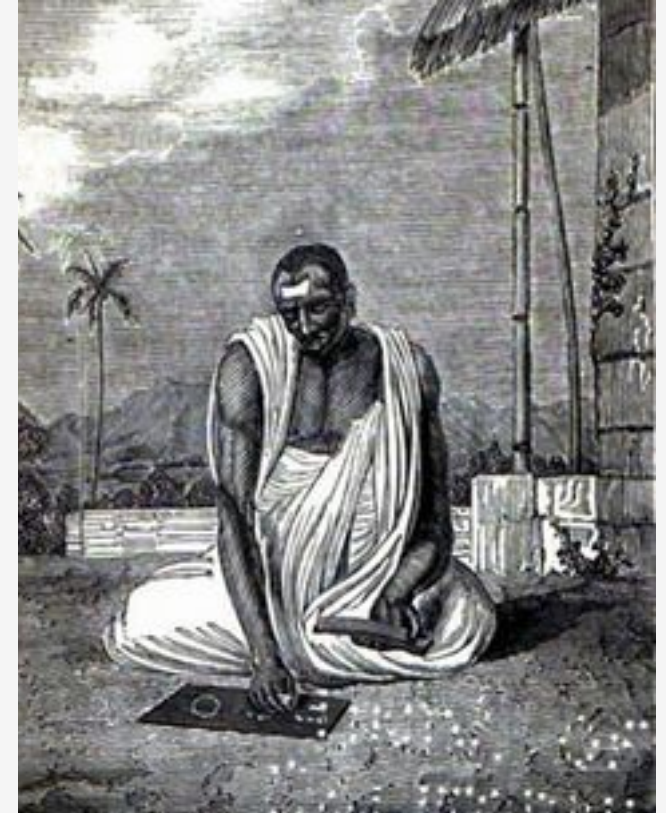
~~upying that column.~~

The ancient Ishango bone is theorized to have tally marks scratched on it, keeping quantitative record of something.
https://en.wikipedia.org/wiki/Ishango_bone

Number Systems – Integers

Brahmagupta argued the case for negative numbers in 628 AD.

```
# Print integers -100 through 100
for i in range(-100, 101): a
    print(i)
```

Number Systems – Integers

Integers include positive and negative whole numbers as well as 0.

Brahmagupta argued the case for negative numbers in 628 AD.

```
# Print integers -100 through 100
for i in range(-100, 101): a
    print(i)
```

Number Systems – Integers

Integers include positive and negative whole numbers as well as 0.

$$\mathbb{Z} = \ldots -3, \, -2, \, -1, 0, 1, 2, 3 \ldots$$

Brahmagupta argued the case for negative numbers in 628 AD.

```
# Print integers -100 through 100
for i in range(-100, 101): a
    print(i)
```
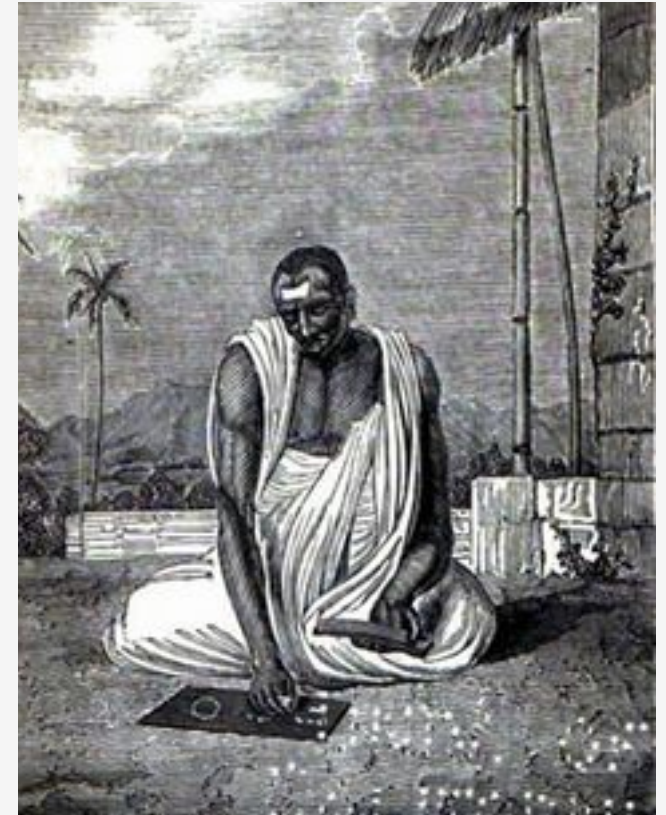
Number Systems – Integers

Integers include positive and negative whole numbers as well as 0.

$$\mathbb{Z} = \ldots -3, -2, -1, 0, 1, 2, 3 \ldots$$

We may take them for granted, but ancient mathematicians were deeply distrusting of the idea of negative numbers.

Brahmagupta argued the case for negative numbers in 628 AD.

```
# Print integers -100 through 100
for i in range(-100, 101): a
    print(i)
```
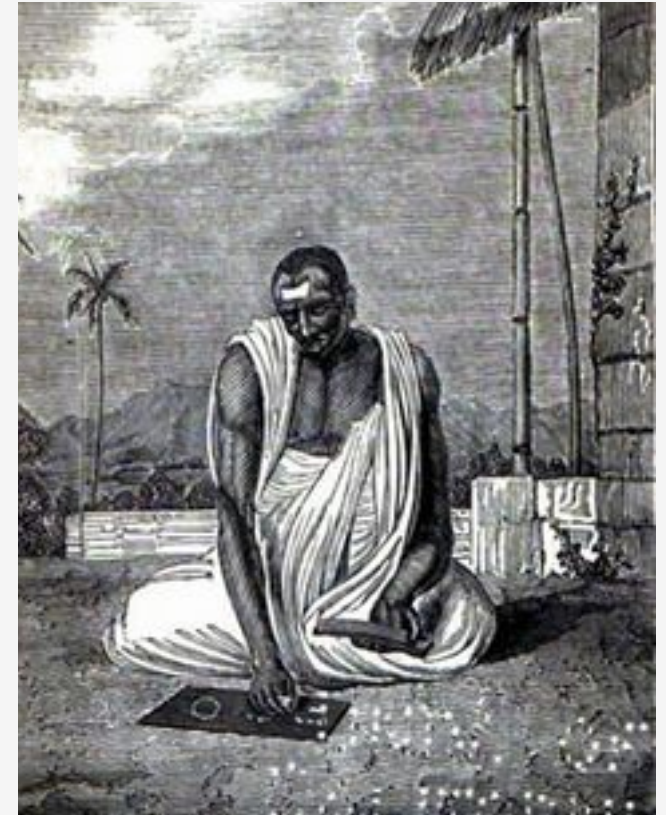
Number Systems – Integers

Integers include positive and negative whole numbers as well as 0.

$$\mathbb{Z} = \ldots -3, -2, -1, 0, 1, 2, 3 \ldots$$

We may take them for granted, but ancient mathematicians were deeply distrusting of the idea of negative numbers.

But when you subtract 5 from 3, you get -2 and this is useful especially when it comes to finances where we measure profits and losses.

Brahmagupta argued the case for negative numbers in 628 AD.

```
# Print integers -100 through 100
for i in range(-100, 101): a
    print(i)
```

Number Systems – Integers

Integers include positive and negative whole numbers as well as 0.

$\mathbb{Z} = \ ... -3, \ -2, \ -1, 0, 1, 2, 3 \ ...$

We may take them for granted, but ancient mathematicians were deeply distrusting of the idea of negative numbers.

But when you subtract 5 from 3, you get -2 and this is useful especially when it comes to finances where we measure profits and losses.

In 628 AD, an Indian mathematician named Brahmagupta showed why negative numbers were necessary for

came

```
# Print integers -100 through 100
for i in range(-100, 101): a
    print(i)
```
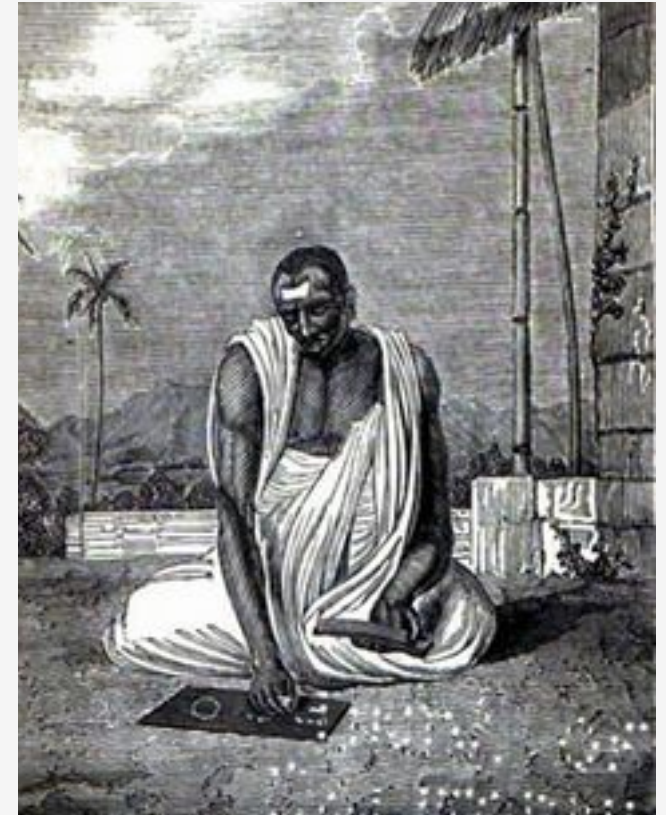
Brahmagupta argued the case for negative numbers in 628 AD.

Number Systems – Rational Numbers

Any number that you can express as a fraction, such as $\frac{3}{2}$, is a

rational number.

This includes all finite decimals, repeating decimals, and integers since they can be expressed as fractions too.

$$\frac{687}{100} = 6.87 \qquad \frac{2}{1} = 2 \qquad \frac{2}{3} = 0.666\ldots$$

They are called *rational* because they are *ratios*.

Rational numbers were quickly deemed necessary because time, resources, and other quantities could not always be measured in discrete units (e.g. part of a gallon, part of a mile).

Computers are better at computing with decimals rather than rational numbers, but you can use symbolic libraries like SymPy to maintain fractions during computation.
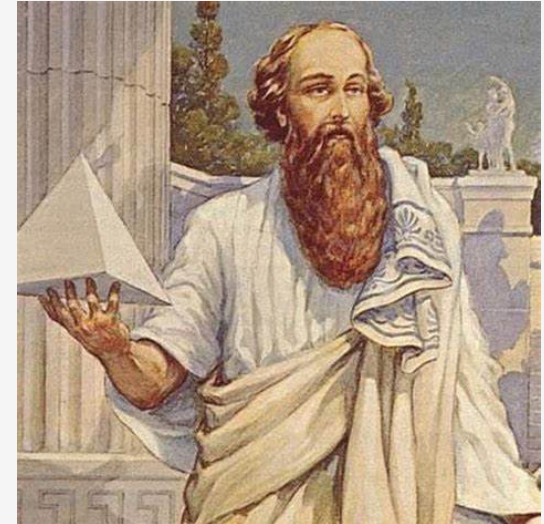
Pythagoras fervently believed all numbers were rational

```
from sympy import *

# Calculate 3/2 + 5/4
x = Rational(3/2)
y = Rational(5/4)

print(x + y) # 11/4
```

Number Systems – Irrational Numbers

Irrational numbers cannot be expressed as a fraction and have an infinite number of decimal places with no clear pattern.

Examples include the famous pi π, Euler's number e, and the square root of certain numbers like $\sqrt{2}$.

$$\pi = 3.14159\ldots \qquad e = 2.17828\ldots \qquad \sqrt{2} = 1.41421\ldots$$

The Greek mathematician Pythagoras created a religion around the belief all numbers were rational.

According to a legend, when one of his followers Hippasus demonstrated irrational numbers do exist by calculating $\sqrt{2}$, Pythagoras drowned him at sea.

Computers are limited to only so many decimal places and must approximate irrational numbers, but you can use symbolic libraries like SymPy to steer away from approximations.
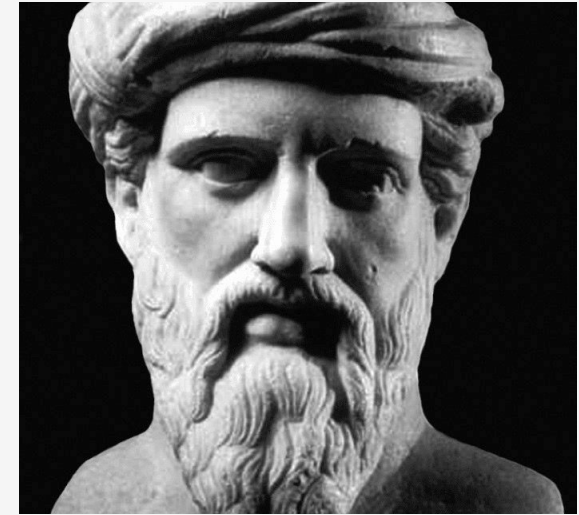
According to legend, Pythagoras drowned Hippasus for proving irrational numbers existed.

```python
from sympy import *

# Calculate pi^2 / pi^3
x = pi ** 2 / pi ** 3

print(x) # 1/pi
```
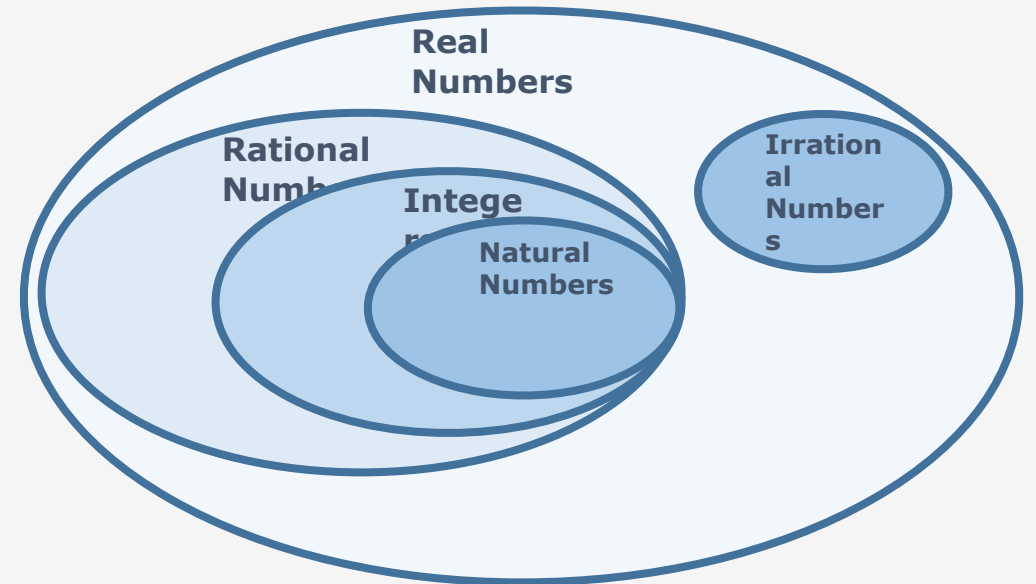
Number Systems - Real Numbers

Real numbers include rational as well as irrational numbers.

$$\mathbb{R}$$

Every number type we have learned up to this point is considered a real number.

In practicality, when you are doing any data science work you can treat any decimals you work with as real numbers.

In plain Python you effectively treat any float type as a real number.


```python
real_numbers = [4.5, 7, -650.3, 25.0 / 2.0]

for x in real_numbers:
    print(x)
```

Number Systems – Complex/Imaginary Numbers

When you take the square root of a negative number, you end up with a complex/imaginary number.

Imaginary numbers do in fact exist, but Descartes named them derogatorily deeming them "useless" until Euler and Gauss pointed out they do have legitimate use cases.

Imaginary Numbers are often denoted by i.

$$\sqrt{-4} = 2i$$

We will steer clear of imaginary and complex numbers but know they may occasionally pop up sometimes especially if you do matrix decomposition and other advanced tasks.

Plain Python will throw an error if you take a square root of a negative number, but SymPy will present it as a complex number.

Learn More: https://www.youtube.com/watch?v=T647CGsuOVU
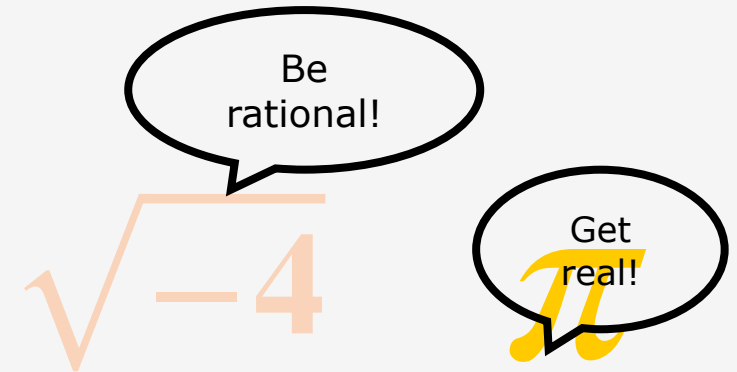

```
from sympy import *

print(sqrt(-4)) # 2*I
```

Why Number System Awareness Matters

Why do we care about number systems?

Doing any data science, machine learning, or any other mathematical modeling work you need to be aware what number domains you are working with.

Mathematical functions will do some combination of inputting integers or real numbers, and outputting integers or real numbers.

Sometimes they will even output complex numbers!

Having a grasp of number systems makes learning about mathematical functions and calculus more intuitive.

Computers like to approximate rational and irrational numbers as decimal values .

Most of the time this is fine, but floating point approximations can be confusing so consider using SymPy in these cases!
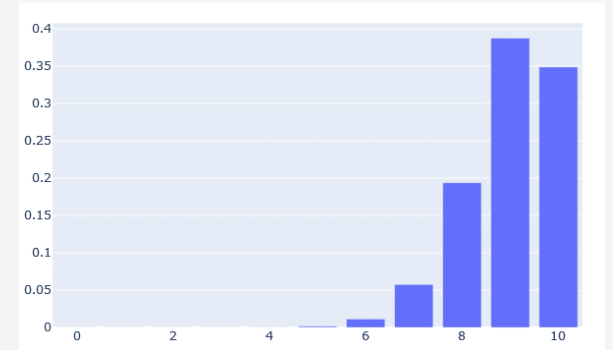
The binomial distribution inputs integers and outputs real numbers.
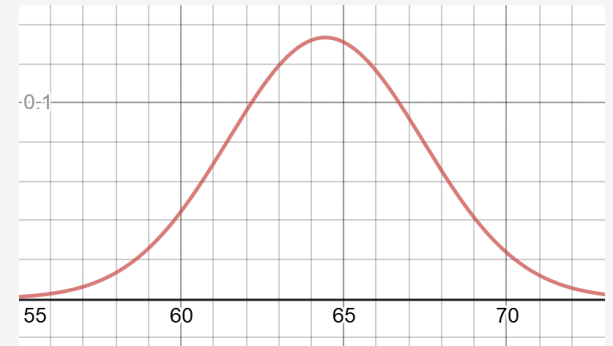
The normal distribution inputs real numbers and outputs real numbers.

Variable

x

In mathematics, a variable is a named placeholder for an unspecified or unknown number.

If you have done some scripting with Python or another programming language, you have an idea what a variable is.

You may have a variable **x** representing any real number, and you can multiply that variable without declaring what it is.

```python
x = int(input("Please input a number\n"))

product = 3 * x

print(product)
```

m a user a

riable in Sy

```python
from sympy import *

x = symbols('x')
product = 3 * x

print(product) # 3*x
```

Functions

Functions are expressions that define relationships between two or more variables.

More specifically a function takes **independent variables** (also called *domain variables* or *input variables*), plugs them into an expression, and then results in a **dependent variable** (also called an *output variable*).

Take this simple linear function: $y = 2x + 1$ or $f(x) = 2x + 1$

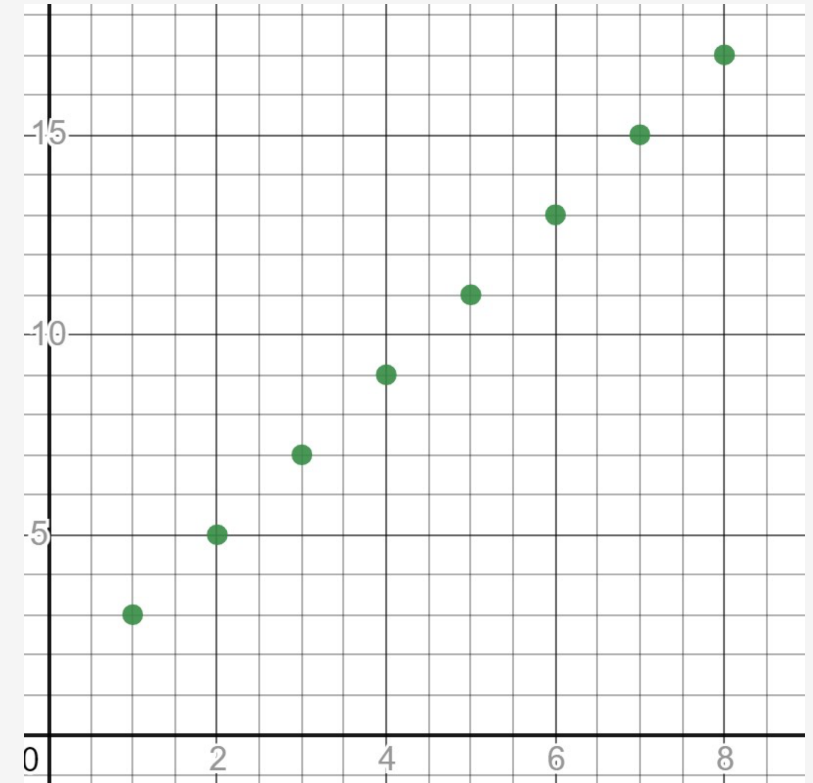**For any given x value, we solve the expression with that *x* to find *y*.**

> When x = 1, then y = 3.
>
> When x = 2, y = 5.
>
> When x = 3, y = 7
>
> and so on.

Functions are useful because they help predict the relationship between variables, such as how many *y* firescan we expect at *x* temperature.
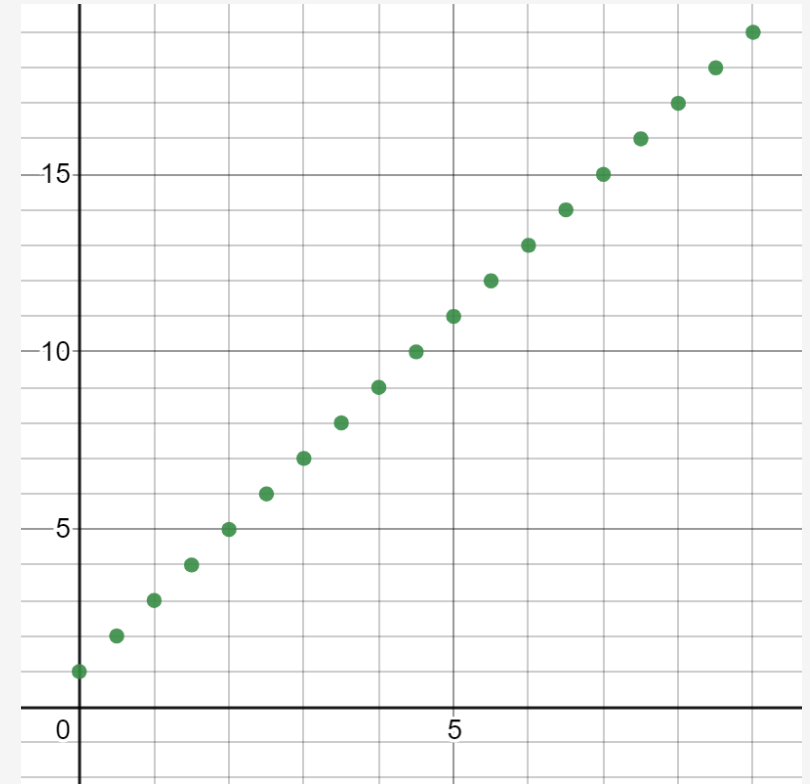


```python
def f(x):
    return 2*x + 1


for x in range(1, 10):
    print("{},{}".format(x, f(x)))
```

Functions

When dealing with real numbers, a subtle but important feature of functions is they often have an infinite number of x values and resulting y values.

Ask yourself this: how many y values can we put through the function $y = 2x + 1$?

Rather than input just 0, 1, 2, 3... why not 0, .5, 1, 1.5, 2, 2.5, 3 as shown to the right?
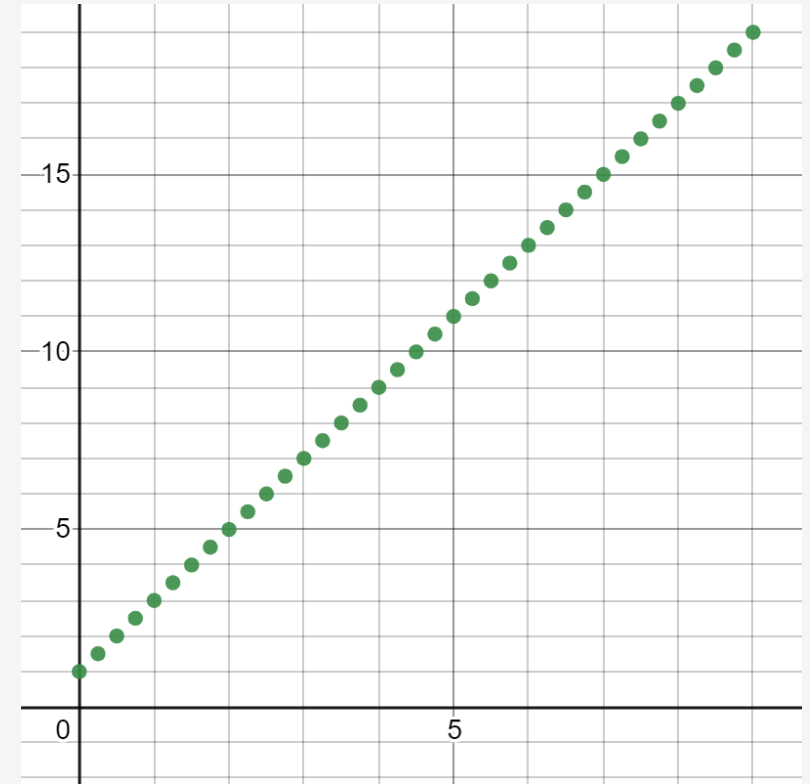
Functions

When dealing with real numbers, a subtle but important feature of functions is they often have an infinite number of x values and resulting y values.

Ask yourself this: how many y values can we put through the function $y = 2x + 1$?

Rather than input just 0, 1, 2, 3… why not 0, .5, 1, 1.5, 2, 2.5, 3 as shown to the right?

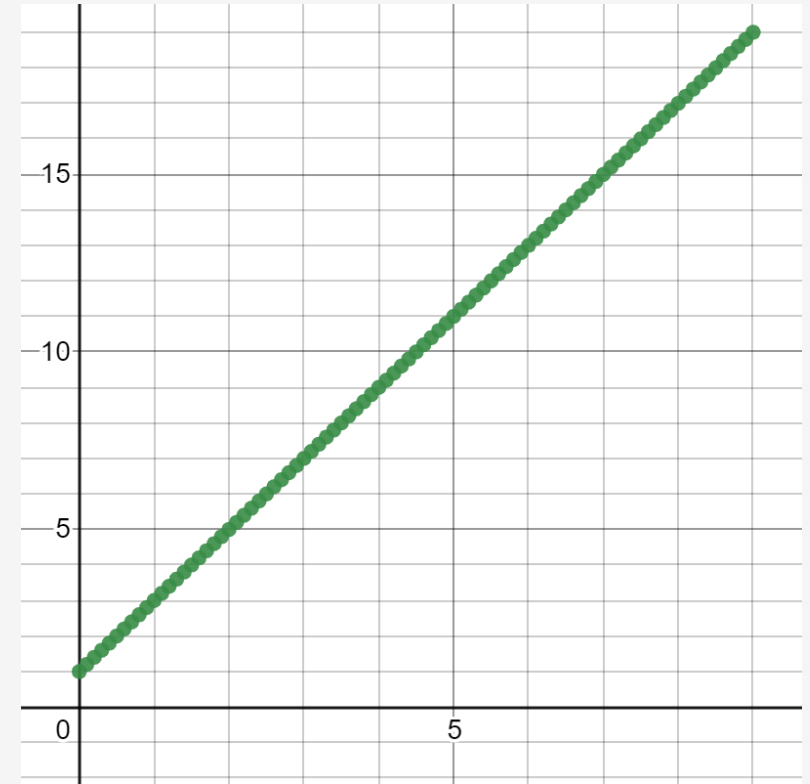Why not increment by .25 for each input?

Functions

When dealing with real numbers, a subtle but important feature of functions is they often have an infinite number of x values and resulting y values.

Ask yourself this: how many y values can we put through the function $y = 2x + 1$?

Rather than input just 0, 1, 2, 3... why not 0, .5, 1, 1.5, 2, 2.5, 3 as shown to the right?

Why not increment by .25 for each input?

Why not increment by .10 for each input?

Functions

When dealing with real numbers, a subtle but important feature of functions is they often have an infinite number of x values and resulting y values.

Ask yourself this: how many y values can we put through the function $y = 2x + 1$?

Rather than input just 0, 1, 2, 3... why not 0, .5, 1, 1.5, 2, 2.5, 3 as shown to the right?

Why not increment by .25 for each input?

Why not increment by .10 for each input?

We can make these steps infinitely small effectively showing $y = 2x + 1$ is a continuous function, where for every possible value of x there is a value for y.

We should visualize our function as a continuous line of real numbers as shown to the right.
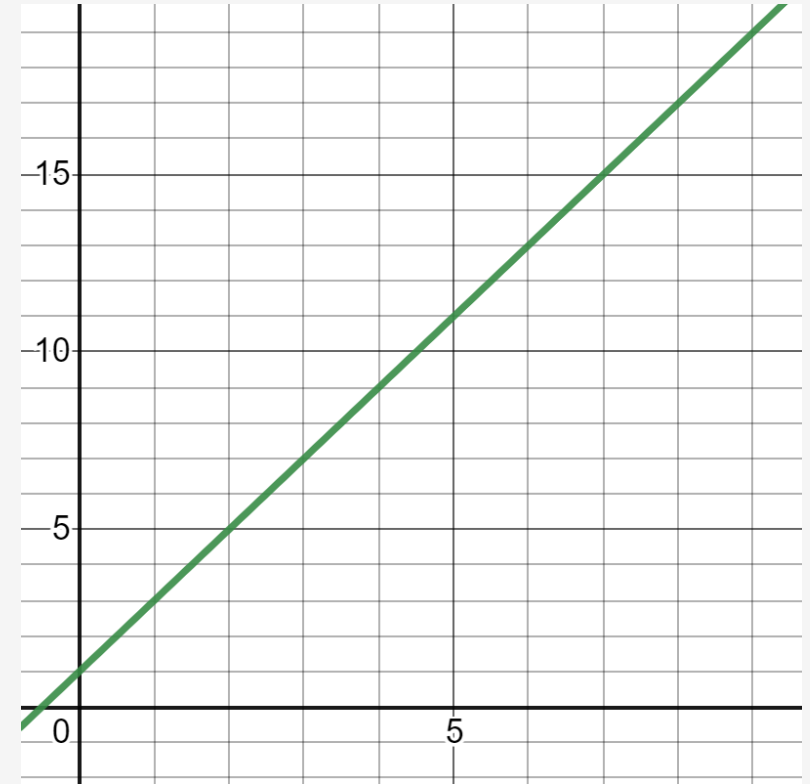
Functions

When we plot on a two-dimensional plane with two number lines (one for each variable) it is known as a **Cartesian plane, x-y plane**, or **coordinate plane**.

We trace a given x value and then look up the corresponding y value, and plot the intersections as a line.
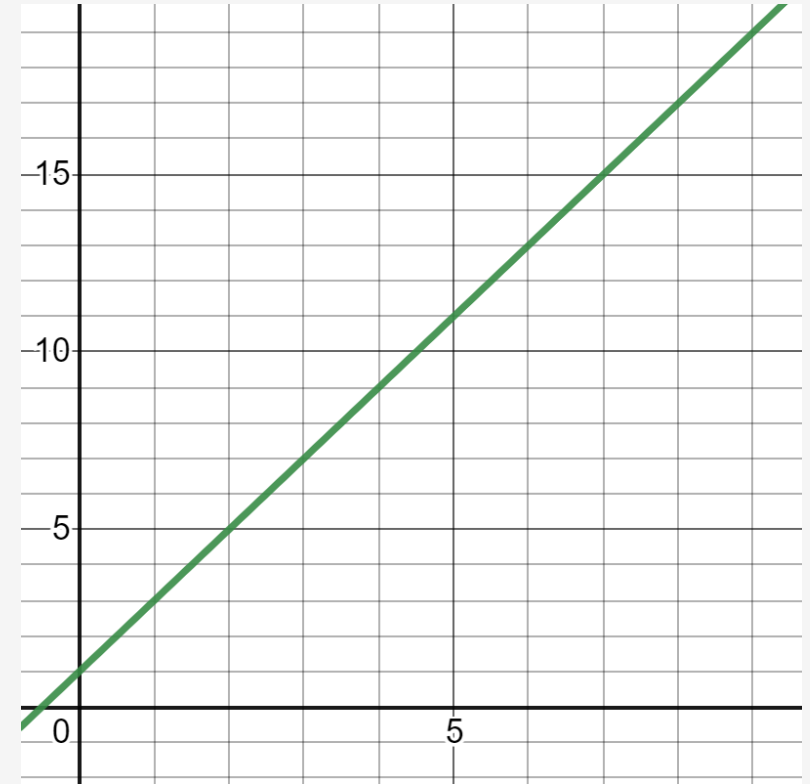
$$f(x) = 2x + 1$$

Notice that due to the nature of real numbers there are an infinite number of x values.

This is why when we plot the function $f(x)$ we get a continuous line with no breaks in it.

There are an infinite number of points on that line, or any part of that line.

SymPy provides a nice, easy way to plot functions as shown to the right (just have matplotlib installed!)

```
from sympy import *

x = symbols('x')
f = 2*x + 1
plot(f)
```
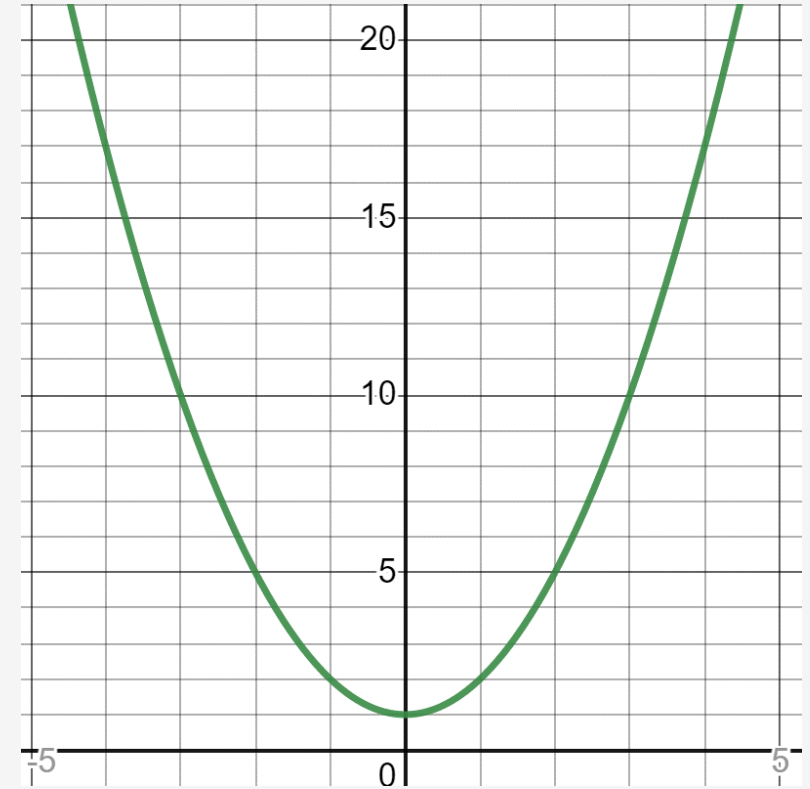
Functions

Here is another example, plotting an exponential function.

$$f(x) = x^2 + 1$$

Notice here we do not get a straight line but rather a smooth, symmetrical curve known as a parabola.

It is continuous but is not linear, as it does not produce values in a straight line.

Curvy functions like this are mathematically harder to work with, but we will learn some tricks to make it not so bad.

```
from sympy import *

x = symbols('x')
f = x**2 + 1
plot(f)
```

3D Functions and Beyond

We can also have functions that accept multiple independent (input) variables.

$$f(x, y) = 2x + 3y$$

Since we have 2 independent variables (x and y) and 1 dependent variable (the output of $f(x, y)$) we need to plot this graph on 3 dimensions to produce a plane of values rather than a line.

No matter how many independent variables you have, your function will typically still only output one dependent variable.

When you solve for multiple dependent variables, you will likely be using separate functions for each one.
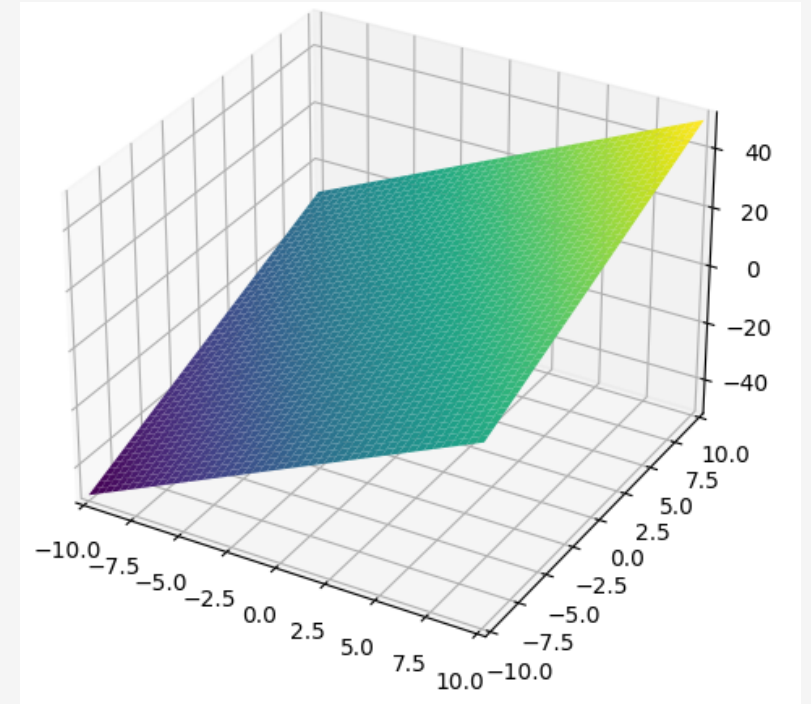

```python
from sympy import *
from sympy.plotting import plot3d

x, y = symbols('x y')
f = 2*x + 3*y
plot3d(f)
```

Exponential Functions

```python
from sympy import *

x = symbols('x')
f = x**2 * x**3

print(f) # x**5
```

```python
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

An **exponent** multiplies a **base** number (like 2) by itself.

'x'
3

'x'
5
 # x**(-3)

```python
from sympy import *

x = symbols('x')
f = x**2 * x**3


print(f) # x**5
```

```python
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

An **exponent** multiplies a **base** number (like 2) by itself.

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

'x'
**3

*x*5

```
from sympy import *

x = symbols('x')
f = x**2 * x**3

print(f) # x**5
```

```
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

An **exponent** multiplies a **base** number (like 2) by itself.

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

When you combine multiple exponents with the same base, you can consolidate them by adding the exponents together.

```python
from sympy import *

x = symbols('x')
f = x**2 * x**3

print(f) # x**5
```

```python
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

An **exponent** multiplies a **base** number (like 2) by itself.

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

When you combine multiple exponents with the same base, you can consolidate them by adding the exponents together.

$$x^2 x^3 = x \cdot x \cdot x \cdot x \cdot x = x^{2+3} = x^5$$

```
from sympy import *

x = symbols('x')
f = x**2 * x**3

print(f) # x**5
```

```
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

An **exponent** multiplies a **base** number (like 2) by itself.

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

When you combine multiple exponents with the same base, you can consolidate them by adding the exponents together.

$$x^2 x^3 = x \cdot x \cdot x \cdot x \cdot x = x^{2+3} = x^5$$

When you divide exponents with the same base, notice that exponents in the denominator can be expressed negatively.

$$\frac{'x'}{5}$$

```python
from sympy import *

x = symbols('x')
f = x**2 * x**3

print(f) # x**5
```

```python
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

An **exponent** multiplies a **base** number (like 2) by itself.

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

When you combine multiple exponents with the same base, you can consolidate them by adding the exponents together.

$$x^2 x^3 = x \cdot x \cdot x \cdot x \cdot x = x^{2+3} = x^5$$

When you divide exponents with the same base, notice that exponents in the denominator can be expressed negatively.

```python
from sympy import *

x = symbols('x')
f = x**2 * x**3

print(f) # x**5
```

```python
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

An **exponent** multiplies a **base** number (like 2) by itself.

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

When you combine multiple exponents with the same base, you can consolidate them by adding the exponents together.

$$x^2 x^3 = x \cdot x \cdot x \cdot x \cdot x = x^{2+3} = x^5$$

When you divide exponents with the same base, notice that exponents in the denominator can be expressed negatively.

$$\frac{x^2}{x^5} = \frac{x \cdot x}{x \cdot x \cdot x \cdot x \cdot x} = \frac{1}{x^3} = x^{-3}$$

```python
from sympy import *

x = symbols('x')
f = x**2 * x**3

print(f) # x**5
```

```python
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

An **exponent** multiplies a **base** number (like 2) by itself.

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

When you combine multiple exponents with the same base, you can consolidate them by adding the exponents together.

$$x^2 x^3 = x \cdot x \cdot x \cdot x \cdot x = x^{2+3} = x^5$$

When you divide exponents with the same base, notice that exponents in the denominator can be expressed negatively.

$$\frac{x^2}{x^5} = \frac{x \cdot x}{x \cdot x \cdot x \cdot x \cdot x} = \frac{1}{x^3} = x^{-3}$$

Alternatively, we could express this as a subtraction operation.

```python
from sympy import *

x = symbols('x')
f = x**2 * x**3

print(f) # x**5
```

```python
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

An **exponent** multiplies a **base** number (like 2) by itself.

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

When you combine multiple exponents with the same base, you can consolidate them by adding the exponents together.

$$x^2 x^3 = x \cdot x \cdot x \cdot x \cdot x = x^{2+3} = x^5$$

When you divide exponents with the same base, notice that exponents in the denominator can be expressed negatively.

$$\frac{x^2}{x^5} = \frac{x \cdot x}{x \cdot x \cdot x \cdot x \cdot x} = \frac{1}{x^3} = x^{-3}$$

Alternatively, we could express this as a subtraction operation.

```python
from sympy import *

x = symbols('x')
f = x**2 * x**3

print(f) # x**5
```

```python
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

An **exponent** multiplies a **base** number (like 2) by itself.

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

When you combine multiple exponents with the same base, you can consolidate them by adding the exponents together.

$$x^2 x^3 = x \cdot x \cdot x \cdot x \cdot x = x^{2+3} = x^5$$

When you divide exponents with the same base, notice that exponents in the denominator can be expressed negatively.

$$\frac{x^2}{x^5} = \frac{x \cdot x}{x \cdot x \cdot x \cdot x \cdot x} = \frac{1}{x^3} = x^{-3}$$

Alternatively, we could express this as a subtraction operation.

$$\frac{x^2}{x^5} = x^2 x^{-5} = x^{2-5} = x^{-3}$$

```python
from sympy import *

x = symbols('x')
f = x**2 * x**3

print(f) # x**5
```

```python
from sympy import *

x = symbols('x')
f = x**2 / x**5
print(f) # x**(-3)
```

Using SymPy to simplify exponential expressions.

Exponential Functions

```python
from sympy import *

expr1 = RealNumber(8)**(1/3)
print(expr1) # 2.0

expr2 = RealNumber(8)**(2/3)
print(expr2) # 4.0
```

Exponential Functions

Fractional exponents are less intuitive but are interpreted as root operations.

```python
from sympy import *

expr1 = RealNumber(8)**(1/3)
print(expr1) # 2.0

expr2 = RealNumber(8)**(2/3)
print(expr2) # 4.0
```

Exponential Functions

Fractional exponents are less intuitive but are interpreted as root operations.

$$\sqrt{9} = 9^{1/2} = 3 \qquad \sqrt[3]{8} = 8^{1/3} = 2$$

```python
from sympy import *

expr1 = RealNumber(8)**(1/3)
print(expr1) # 2.0

expr2 = RealNumber(8)**(2/3)
print(expr2) # 4.0
```

Exponential Functions

Fractional exponents are less intuitive but are interpreted as root operations.

$$\sqrt{9} = 9^{1/2} = 3 \qquad \sqrt[3]{8} = 8^{1/3} = 2$$

This interpretation makes sense when you use the additive property.

```python
from sympy import *

expr1 = RealNumber(8)**(1/3)
print(expr1) # 2.0

expr2 = RealNumber(8)**(2/3)
print(expr2) # 4.0
```

Exponential Functions

Fractional exponents are less intuitive but are interpreted as root operations.

$$\sqrt{9} = 9^{1/2} = 3 \qquad \sqrt[3]{8} = 8^{1/3} = 2$$

This interpretation makes sense when you use the additive property.

$$\sqrt{9} \cdot \sqrt{9} = 9^{1/2} \cdot 9^{1/2} = 9^{\left(\frac{1}{2}+\frac{1}{2}\right)} = 9^1 = 9$$

```python
from sympy import *

expr1 = RealNumber(8)**(1/3)
print(expr1) # 2.0

expr2 = RealNumber(8)**(2/3)
print(expr2) # 4.0
```

Exponential Functions

Fractional exponents are less intuitive but are interpreted as root operations.

$$\sqrt{9} = 9^{1/2} = 3 \qquad \sqrt[3]{8} = 8^{1/3} = 2$$

This interpretation makes sense when you use the additive property.

$$\sqrt{9} \cdot \sqrt{9} = 9^{1/2} \cdot 9^{1/2} = 9^{\left(\frac{1}{2}+\frac{1}{2}\right)} = 9^1 = 9$$

When you exponent an exponent, you multiply the exponents together.

```python
from sympy import *

expr1 = RealNumber(8)**(1/3)
print(expr1) # 2.0

expr2 = RealNumber(8)**(2/3)
print(expr2) # 4.0
```

Exponential Functions

Fractional exponents are less intuitive but are interpreted as root operations.

$$\sqrt{9} = 9^{1/2} = 3 \qquad \sqrt[3]{8} = 8^{1/3} = 2$$

This interpretation makes sense when you use the additive property.

$$\sqrt{9} \cdot \sqrt{9} = 9^{1/2} \cdot 9^{1/2} = 9^{\left(\frac{1}{2} + \frac{1}{2}\right)} = 9^1 = 9$$

When you exponent an exponent, you multiply the exponents together.

$$\left(\sqrt{9}\right)^2 = \left(9^{1/2}\right)^2 = 9^{\left(\frac{1}{2} \cdot 2\right)} = 9^1 = 9$$

```python
from sympy import *

expr1 = RealNumber(8)**(1/3)
print(expr1) # 2.0

expr2 = RealNumber(8)**(2/3)
print(expr2) # 4.0
```
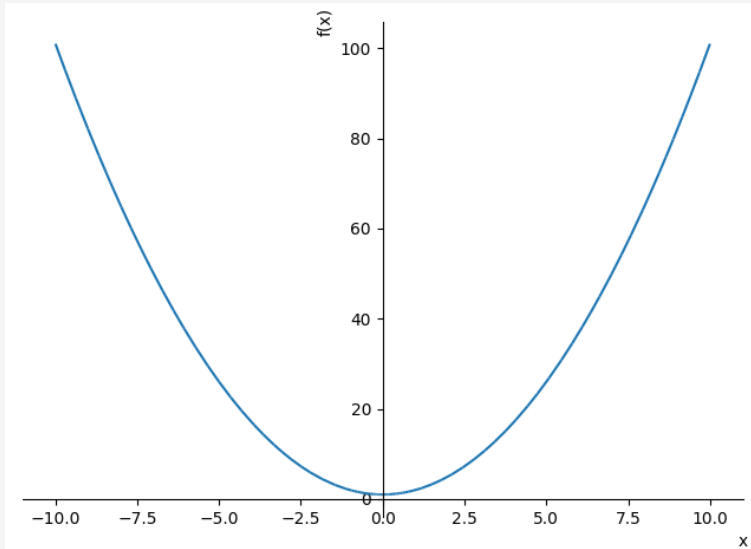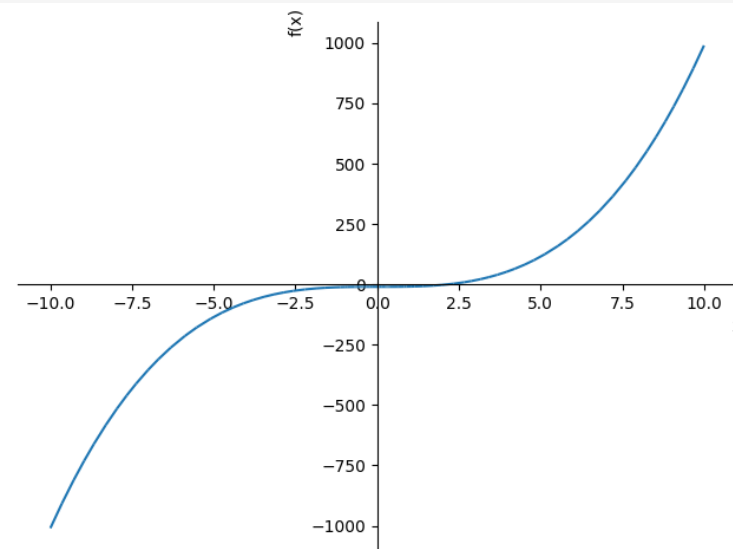
Exponential Functions

Fractional exponents are less intuitive but are interpreted as root operations.

$$\sqrt{9} = 9^{1/2} = 3 \qquad \sqrt[3]{8} = 8^{1/3} = 2$$

This interpretation makes sense when you use the additive property.

$$\sqrt{9} \cdot \sqrt{9} = 9^{1/2} \cdot 9^{1/2} = 9^{\left(\frac{1}{2}+\frac{1}{2}\right)} = 9^1 = 9$$

When you exponent an exponent, you multiply the exponents together.

$$\left(\sqrt{9}\right)^2 = \left(9^{1/2}\right)^2 = 9^{\left(\frac{1}{2}\cdot 2\right)} = 9^1 = 9$$

This helps interpret fractional exponents where numerator is not "1".

```python
from sympy import *

expr1 = RealNumber(8)**(1/3)
print(expr1) # 2.0

expr2 = RealNumber(8)**(2/3)
print(expr2) # 4.0
```

Exponential Functions

Fractional exponents are less intuitive but are interpreted as root operations.

$$\sqrt{9} = 9^{1/2} = 3 \qquad \sqrt[3]{8} = 8^{1/3} = 2$$

This interpretation makes sense when you use the additive property.

$$\sqrt{9} \cdot \sqrt{9} = 9^{1/2} \cdot 9^{1/2} = 9^{\left(\frac{1}{2}+\frac{1}{2}\right)} = 9^1 = 9$$

When you exponent an exponent, you multiply the exponents together.

$$\left(\sqrt{9}\right)^2 = \left(9^{1/2}\right)^2 = 9^{\left(\frac{1}{2}\cdot 2\right)} = 9^1 = 9$$

This helps interpret fractional exponents where numerator is not "1".

$$8^{2/3} = \left(8^{1/3}\right)^2 = 2^2 = 4$$

```python
from sympy import *

expr1 = RealNumber(8)**(1/3)
print(expr1) # 2.0

expr2 = RealNumber(8)**(2/3)
print(expr2) # 4.0
```

Exponential Functions – Plot Examples

$$f(x) = x^2 + 1$$

$$f(x) = x^3 - 10$$

$$f(x) = x^{1/2}$$


```
from sympy import *
x = symbols('x')
f = x**2 + 1
plot(f)
```

```
from sympy import *
x = symbols('x')
f = x**3 - 10
plot(f)
```
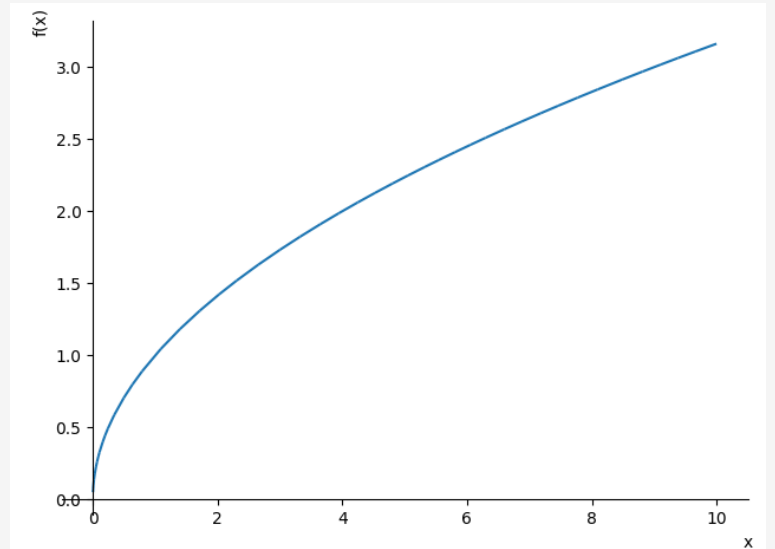
```
from sympy import *
x = symbols('x')
f = x**(1/2)
plot(f)
```

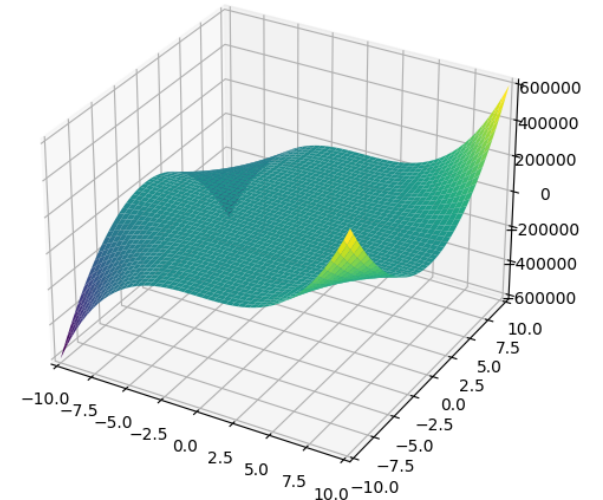# Exponential Functions – Plot Examples

$$f(x) = x^{1/3}$$

$$f(x) = x^{-2}$$
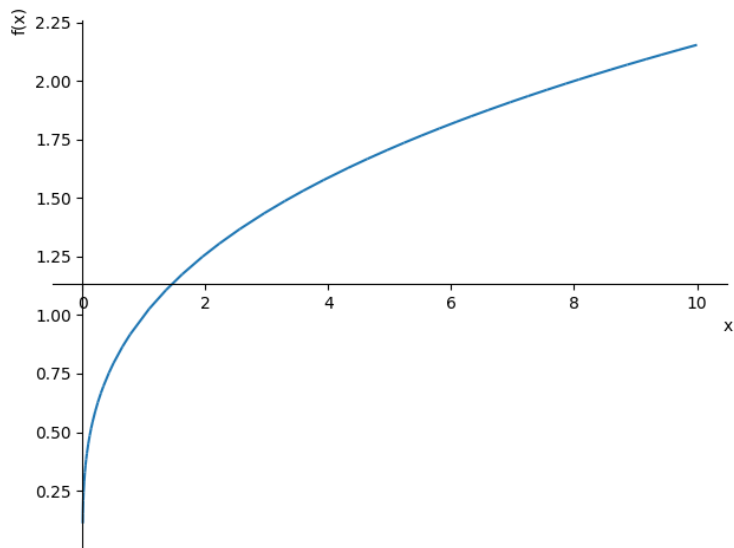
$$f(x) = 2x^2 * 3y^3$$


```python
from sympy import *
x = symbols('x')
f = x**(1/3)
plot(f)
```
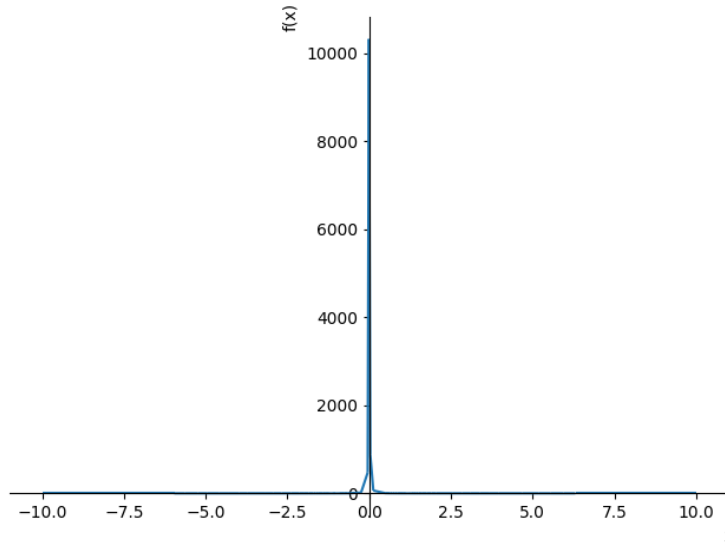
```python
from sympy import *
x = symbols('x')
f = x**(-2)
plot(f)
```

```python
from sympy import *
from sympy.plotting import plot3d
x,y = symbols('x y')
f = 2*x**2 * 3*y**3
plot3d(f)
```

Logarithmic Functions

Commonly we are interested in solving for an exponent as shown below:

$$2^x = 8$$

We intuitively can figure out that $x = 3$, but we need to be able to express this in a more streamlined way, and this is what the logarithm does.

Moving things around, here is how we re-express the operation as a logarithm.

$$\log_2 8 = x$$

Above, we can read this operation as "2 raised to what exponent gives me 8?" and the resulting value will be the exponent 3.

More generically, we can re-express finding an exponent using the formula to the right.

```python
from math import log

# 2 raised to what exponent gives us 8?
exponent = log(8,2)

# The answer is 3.0
print(exponent)
```

$$\log_a b = x$$

$$a^x = b$$

Logarithmic Functions – Plot Examples

$$f(x) = \log_2(x)$$
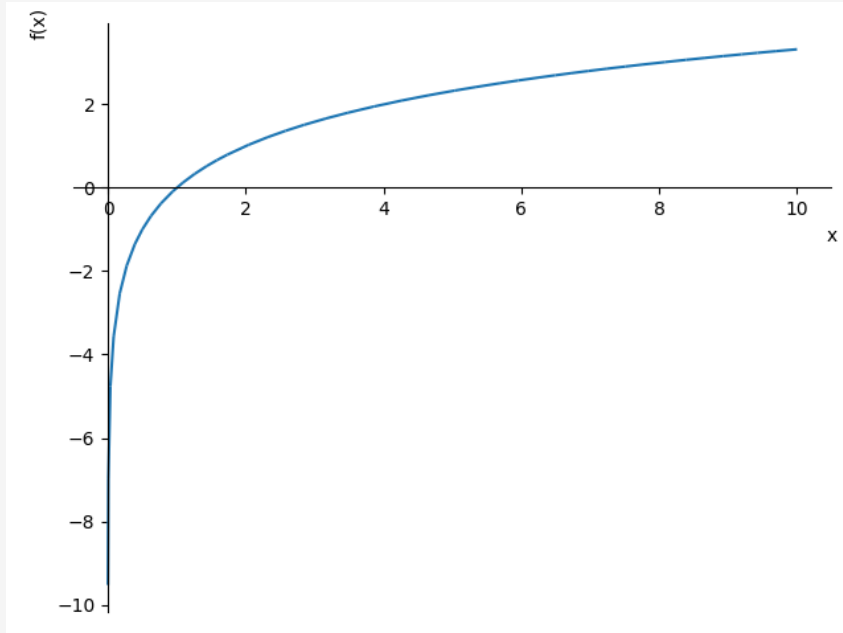
$$f(x) = \log_{10}(x)$$


```
from sympy import *
x = symbols('x')
f = log(x,2) # x raised to what power gives me 2?
plot(f)
```
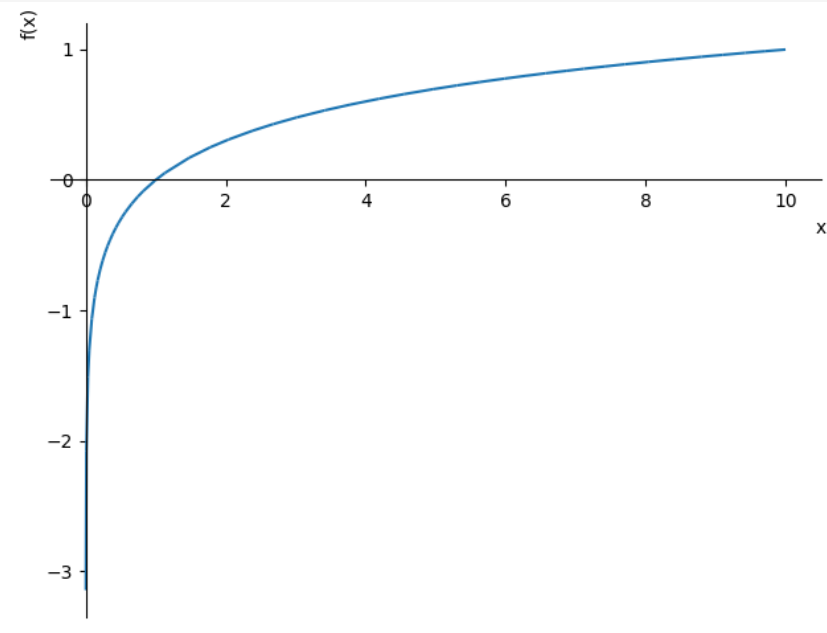
```
from sympy import *
x = symbols('x')
f = log(x,10) # x raised to what power gives me
10? plot(f)
```

Euler's Number

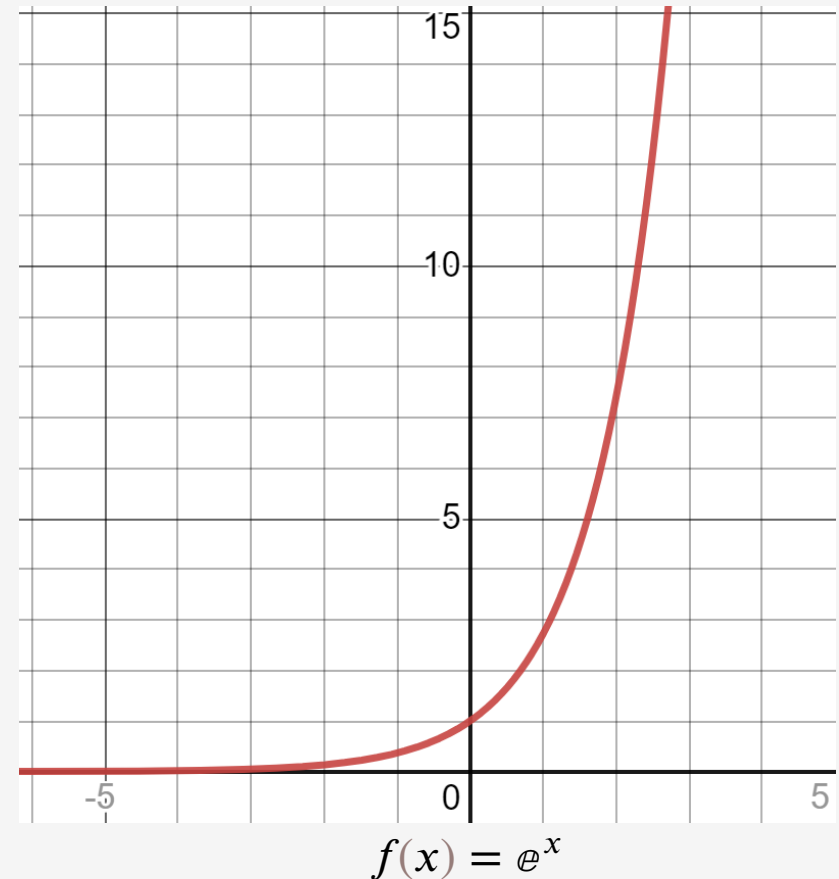**There is a special number that shows up quite a bit in math called Euler's number e.**

It is transcendental just like π and is approximately 2.71828.

In practicality, e is often used as an exponential base.

$f(x) = e^x$

You will also see it used as a **natural logarithm**, which is a logarithm with a base of e.

$\log_e y = \ln y$

$$f(x) = e^x$$

Euler's Number

	Exponential	**Logarithmic**
Plain Python	```python\nfrom math import exp\n\nx = 3\nresult = exp(x) # e^x\n\nprint(result) # 20.085536923187668\n```	```python\nfrom math import log\n\nx = 3\nresult = log(x) # defaults to e base\n\nprint(result) # 1.0986122886681098\n```
SymPy	```python\nfrom sympy import *\n\nx = symbols('x')\nf = exp(x) # e^x\n\nplot(f)\n```	```python\nfrom sympy import *\n\nx = symbols('x')\nf = log(x) # defaults to e base\n\nplot(f)\n```

Section II

Calculus

Limits

We will see a few key concepts that involve infinity, forever approaching a value but never touching that value.

Take this function which is plotted to the right:

$$f(x) = \frac{1}{x}$$

Notice that as x increases we approach $f(x)$ being 0 but we never actually reach 0.

The fate of this function is as x forever extends into infinity, it will keep getting closer to 0 but never reach 0.
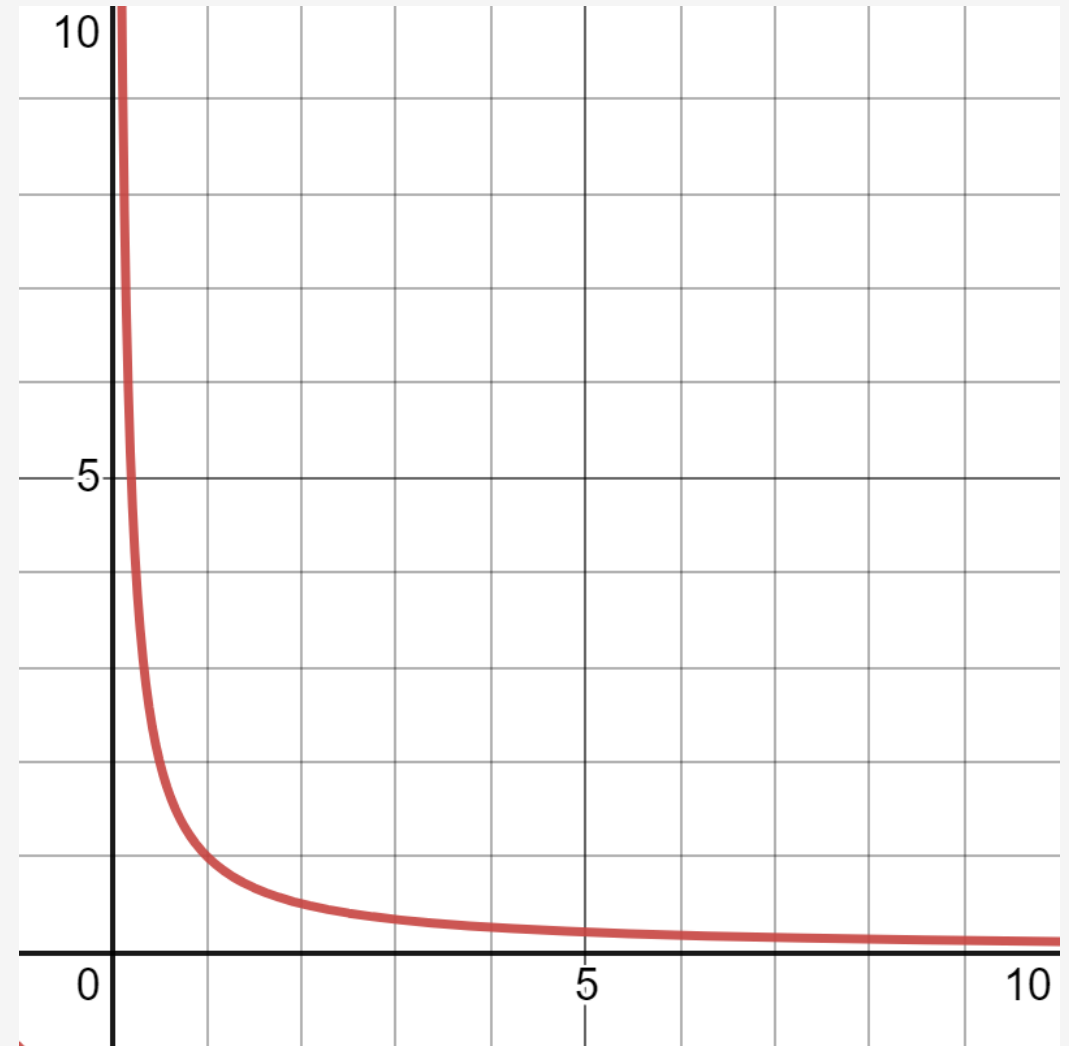
Limits

The way we express a value that is forever being approached, but never reached, is through a limit.

$$\lim_{x \to \infty} \frac{1}{x} = 0$$

The way we read this is "as x approaches infinity ∞ the function $\frac{1}{x}$ approaches 0."

Here is how we calculate a limit in SymPy:

```python
from sympy import *

x = symbols('x')
f = 1 / x
result = limit(f, x, oo)

print(result) # 0
```
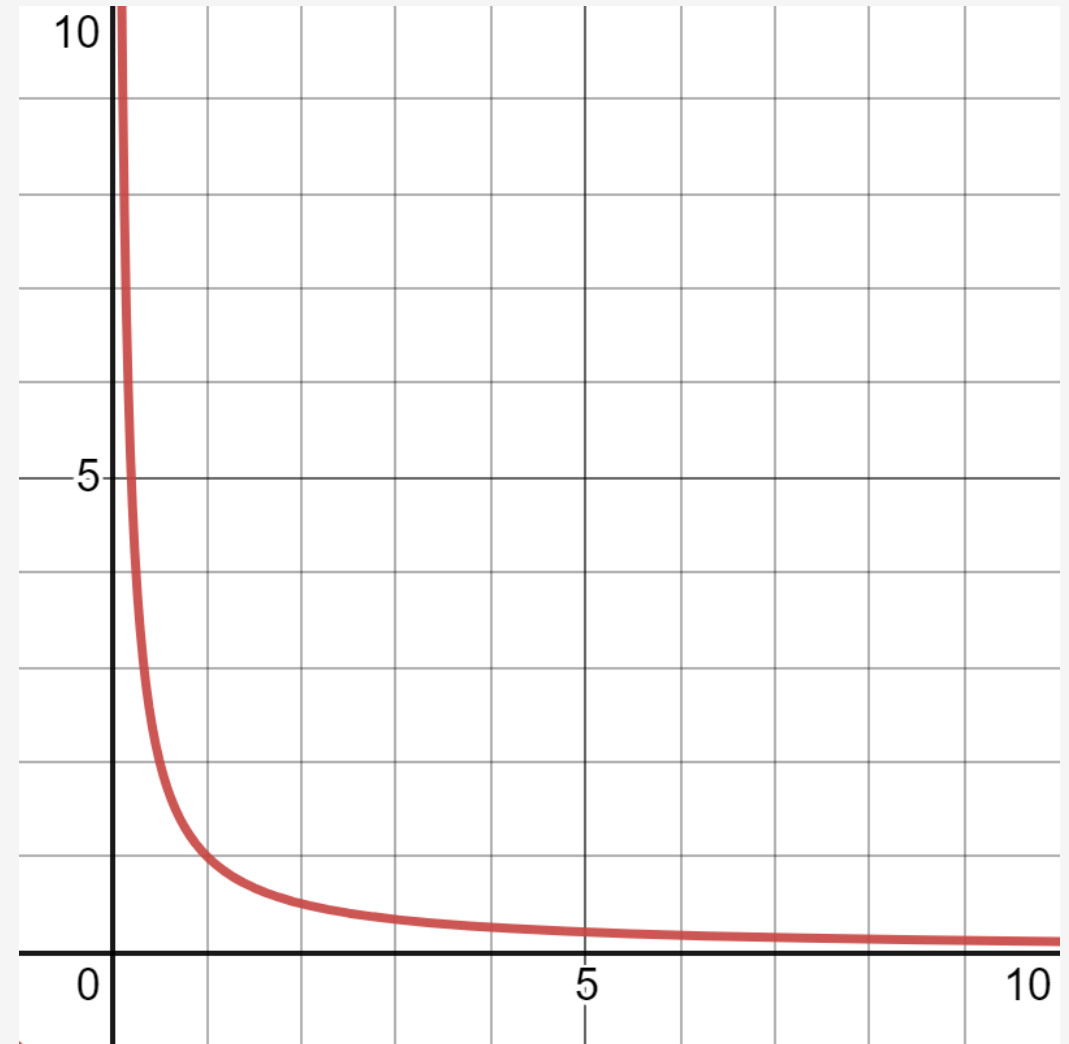
Limits – Euler's Number

Euler's Number is actually derived from a limit using this function which is plotted to the right.

$$f(x) = \left(1 + \frac{1}{x}\right)^x$$

If we keep increasing x we do start to approach a number 2.71828.... which is what we call Euler's number e.

$$e = \lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x$$

$$e = 2.71828182845905$$
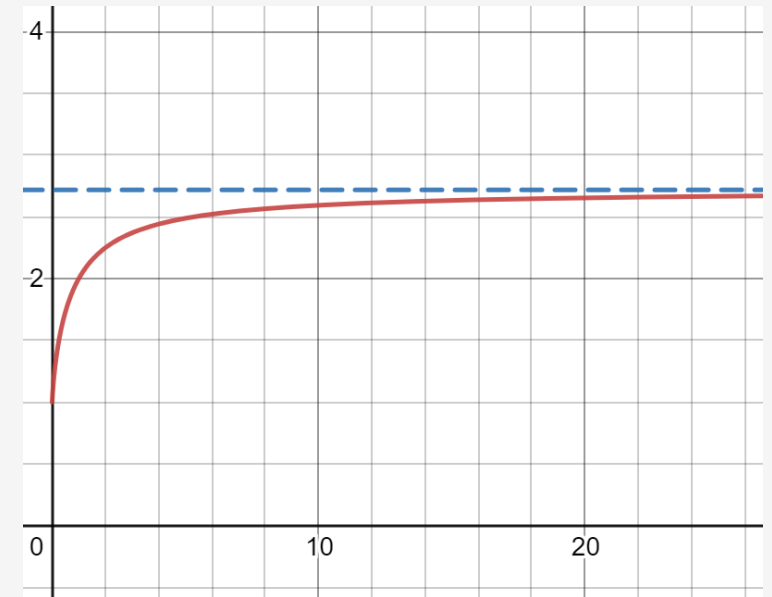

```python
from sympy import *

n = symbols('n')
f = (1 + (1/n))**n
result = limit(f, n, oo)

print(result) # E
print(result.evalf()) # 2.71828182845905
```

Derivatives

A derivative tells the slope of a function, and it is useful to measure the rate of change at any point in a function.

Why do we care about derivatives? They are often used in machine learning and other mathematical algorithms, especially with gradient descent.

When the slope is 0, that means we are at the minimum or maximum of an output variable.

This concept will be useful later when we do linear regression, logistic regression, and neural networks.
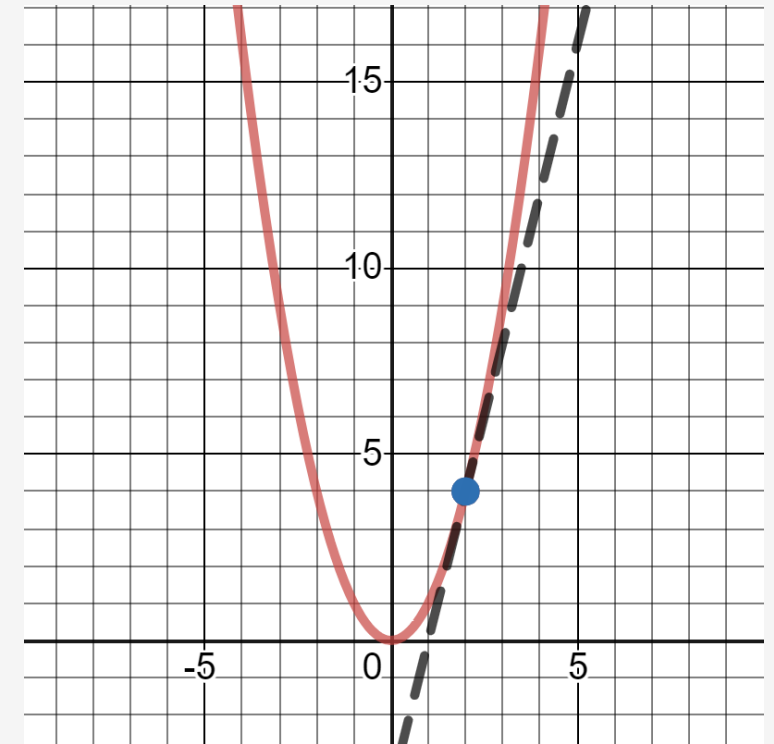
Derivatives – Measuring "Steepness"

A derivative provides the slope at a given x value, so what is the slope for this function at $x = 2$?

$f(x) = x^2$

We can measure "steepness" at any point in the curve, and we can visualize this with a tangent line.

Think of a **tangent line** as a straight line that "just touches" the curve at a given x value, and it also provides the slope at that point.

How do you think we can estimate this slope?
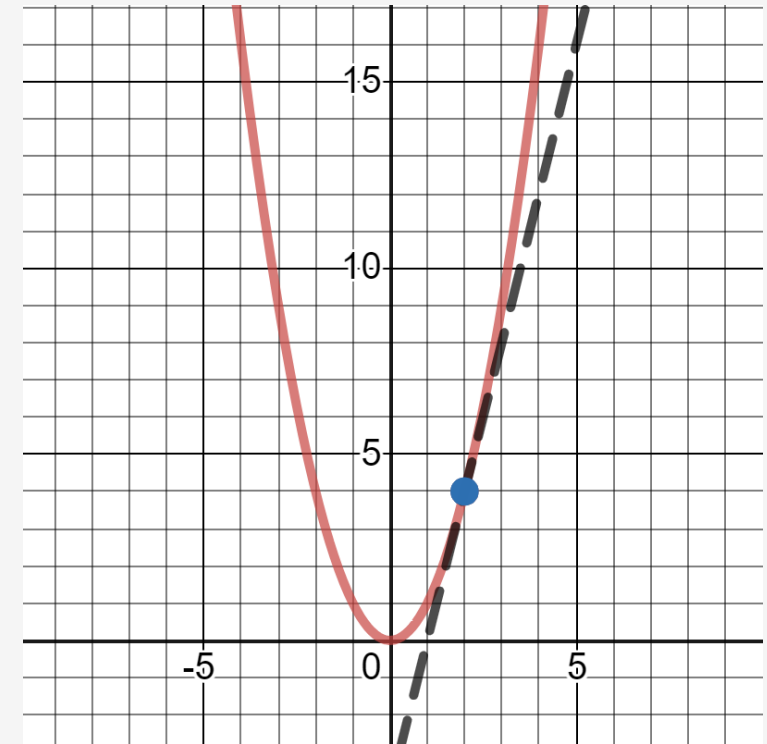
What is the slope for $f(x) = x^2$ at $x = 2$?

Derivatives – Measuring "Steepness"

Take $x = 2$ and a nearby value $x = 2.1$.

When passed to the function $f(x) = x^2$ these two x-values will yield the following values.

$f(x) = x^2$

$f(2) = 4$

$f(2.1) = 4.41$

The resulting line that passes through these two points has a slope of 4.41.

You can quickly calculate the slope between two points using the simple rise-over-run formula.

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{4.41 - 4}{2.1 - 2} = 4.1$$
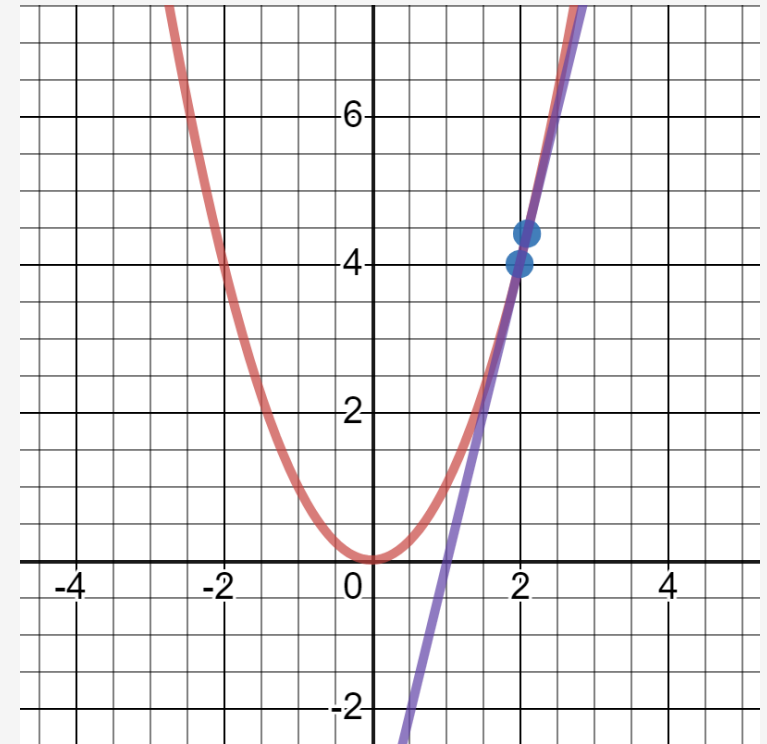
We can estimate slope at a given point by drawing a line through a close neighboring point.

Derivatives – Measuring "Steepness"

Here is how we implement this slope estimator in Python, where `derivative_x()` returns the slope and `my_function()` is the parabola function.

We add .00001 to our x-value to get a *really* close neighboring x-value.

```python
def derivative_x(f, x, step_size):
    m = (f(x + step_size) - f(x)) / ((x + step_size) - x)
    return m


def my_function(x):
    return x ** 2

slope_at_2 = derivative_x(my_function, 2, .00001)

print(slope_at_2)  # prints 4.000010000000827
```
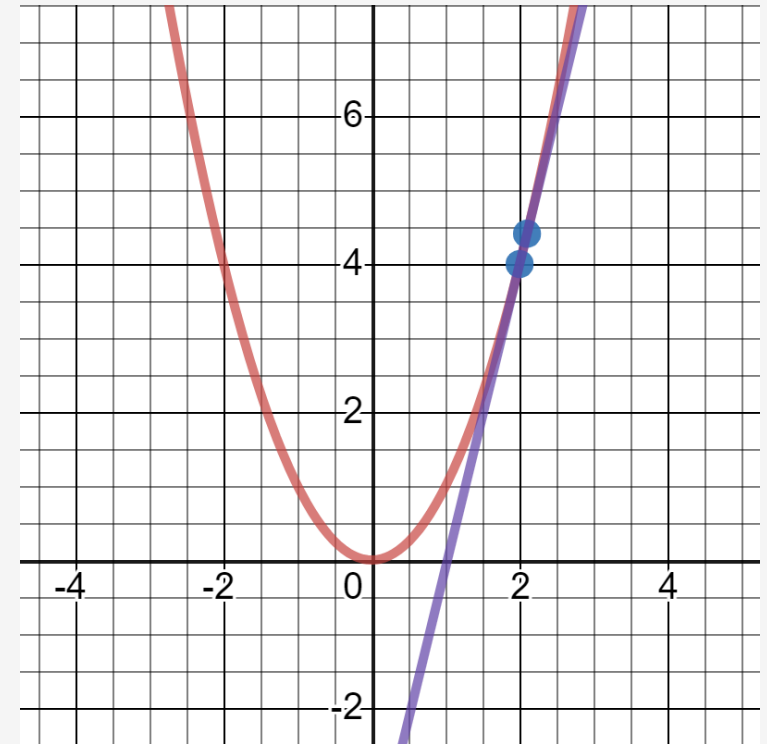

We can estimate slope at a given point by drawing a line through a close neighboring point.

Derivatives – Measuring "Steepness"

A more exact way to use this technique is to use a limit in SymPy.

We will forever make our step size smaller and approaching 0, but never actually reach 0 .

$$\lim_{s \to 0} \frac{(x+s)^2 - x^2}{(x+s) - x}$$

$$\lim_{s \to 0} \frac{(2+s)^2 - 2^2}{(2+s) - 2} = 4$$

```python
from sympy import *

# x and step size
x, s = symbols('x s')

# declare function
f = x**2

# slope between two points with step s
slope_f = (f.subs(x, x + s) - f) / ((x+s) - x)

# substitute 2 for x
slope_2 = slope_f.subs(x, 2)

# calculate slope at x = 2
# by forever approaching a step size of 0
result = limit(slope_2, s, 0)

print(result) # 4
```
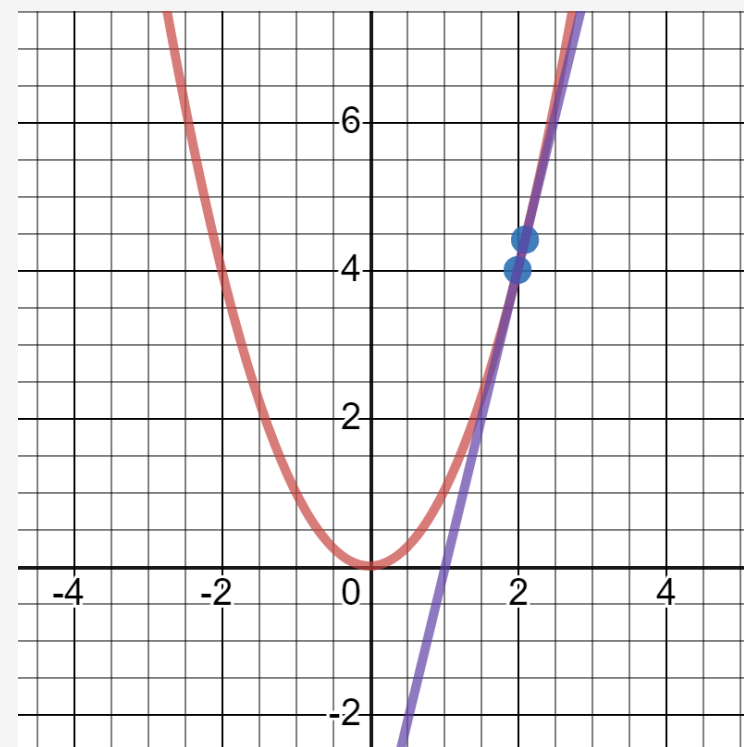

If we forever make the gap size approach 0 without touching 0, we will get our exact slope of 4!

Derivatives – Using SymPy

There is a cleaner way to calculate slope.

By taking a function $f(x)$ we can calculate a derivative function $\frac{d}{dx}f(x)$ that will return a slope for a given x-value.

To avoid textbook pencil-and-paper math work, we can use SymPy's `diff()` to calculate a derivative for a function with respect to x.

After using SymPy to calculate the derivative function to the right, we now can cleanly and quickly calculate slope for any x-value.

$$f(x) = x^2$$

$$\frac{d}{dx}f(x) = 2x$$

```python
from sympy import *

# Declare 'x' to SymPy
x = symbols('x')

# Now just use Python syntax to declare
function
f = x**2

# Calculate the derivative of the function
dx_f = diff(f)
print(dx_f) # prints 2*x
```

Derivatives – Using SymPy

We can now take this derivative function back in Python and calculate the slope for $x = 2$.

Implementing this function as `dx_f()` to the right, we can now calculate the slope for any x-value.

For $x = 2$, we see the slope is 4.

$$f(x) = x^2$$
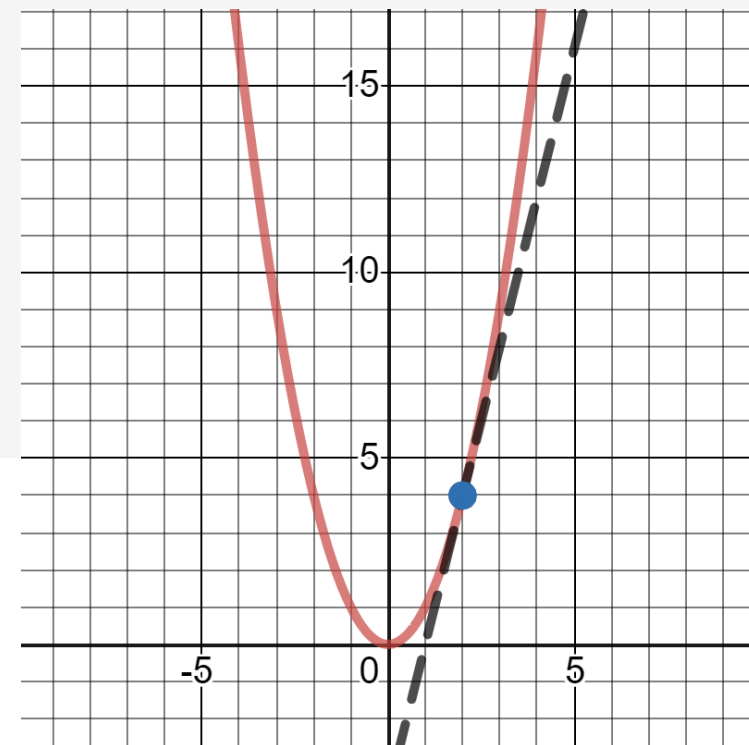
$$\frac{d}{dx}f(x) = 2x$$

$$\frac{d}{dx}f(2) = 4$$

```
# Using Plain Python
def f(x):
    return x ** 2

def dx_f(x):
    return 2 * x

slope_at_2 = dx_f(2.0)

print(slope_at_2)  # prints 4.0
```

Derivatives – Using SymPy

Of course, we can continue using SymPy too.

Just use the derivative function's `subs()` to plug in a specific value for x.

$$f(x) = x^2$$

$$\frac{d}{dx}f(x) = 2x$$

$$\frac{d}{dx}f(2) = 4$$

```python
# Continue Using SymPy
from sympy import *

# Declare 'x' to SymPy
x = symbols('x')

# Now just use Python syntax
# to declare function
f = x**2

# Calculate the derivative of the function
dx_f = diff(f)

# Calculate the slope at x = 2
print(dx_f.subs(x,2)) # prints 4
```
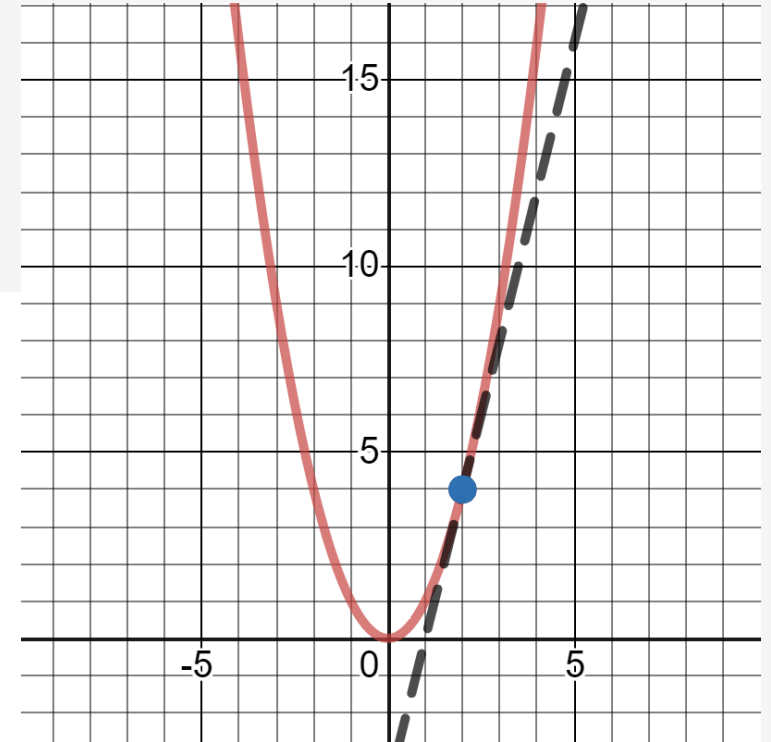
Partial Derivatives

Partial Derivatives are derivatives on functions that have multiple input variables.

Because there is more than one input variable, we have slopes in more than one direction as plotted with this function.


```python
from sympy import *
from sympy.plotting import plot3d

# Declare x and y to SymPy
x,y = symbols('x y')

# Now just use Python syntax to declare
function
f = 2*x**3 + 3*y**3

# plot the function
plot3d(f)
```
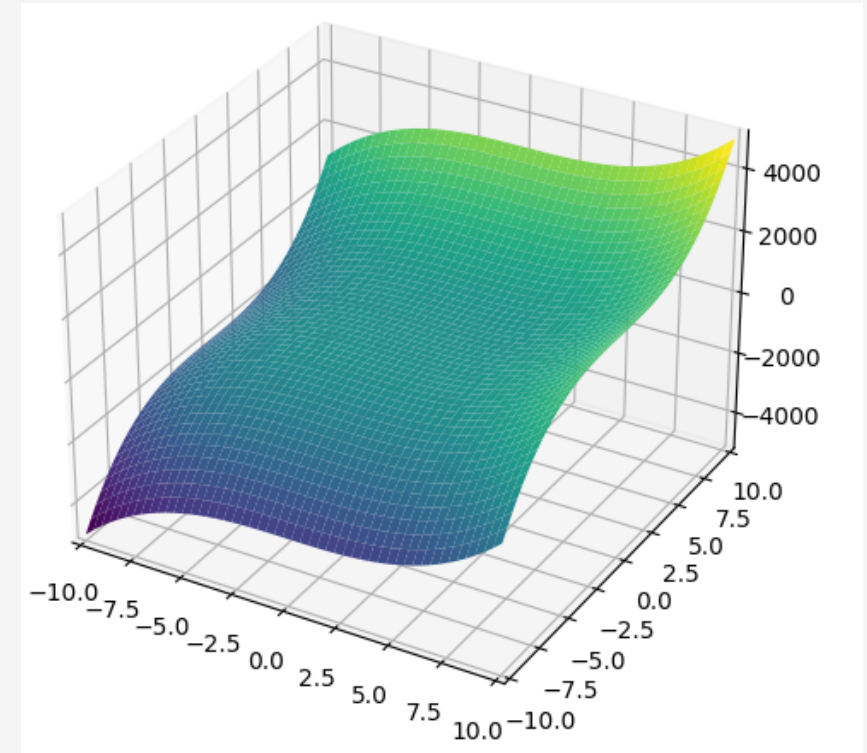
Partial Derivatives

Logically, if there are multiple slopes then we can have multiple derivatives for each variable.

$$f(x, y) = 2x^3 + 3y^3$$

$$\frac{d}{dx}f(x, y) = 6x^2$$

$$\frac{d}{dy}f(x, y) = 9y^2$$

Rather than finding the slope on a 1-dimensional function, we have slopes with respect to multiple variables in several directions.

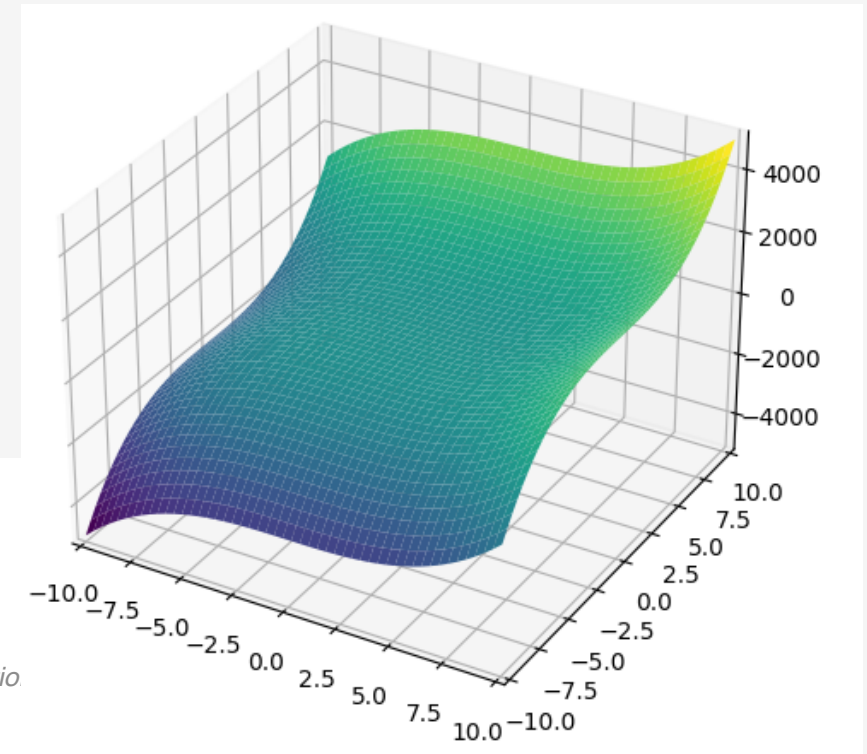
```python
from sympy import *
from sympy.plotting import plot3d

# Declare x and y to SymPy
x,y = symbols('x y')

# Now just use Python syntax to declare functio
f = 2*x**3 + 3*y**3

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)

print(dx_f) # prints 6*x**2
print(dy_f) # prints 9*y**2

# plot the function
plot3d(f)
```
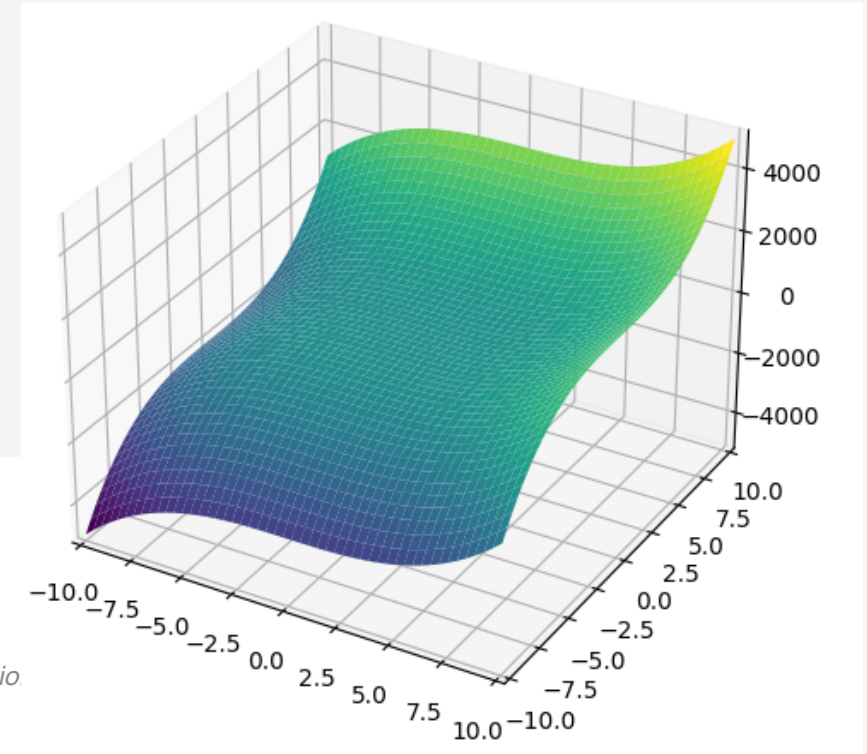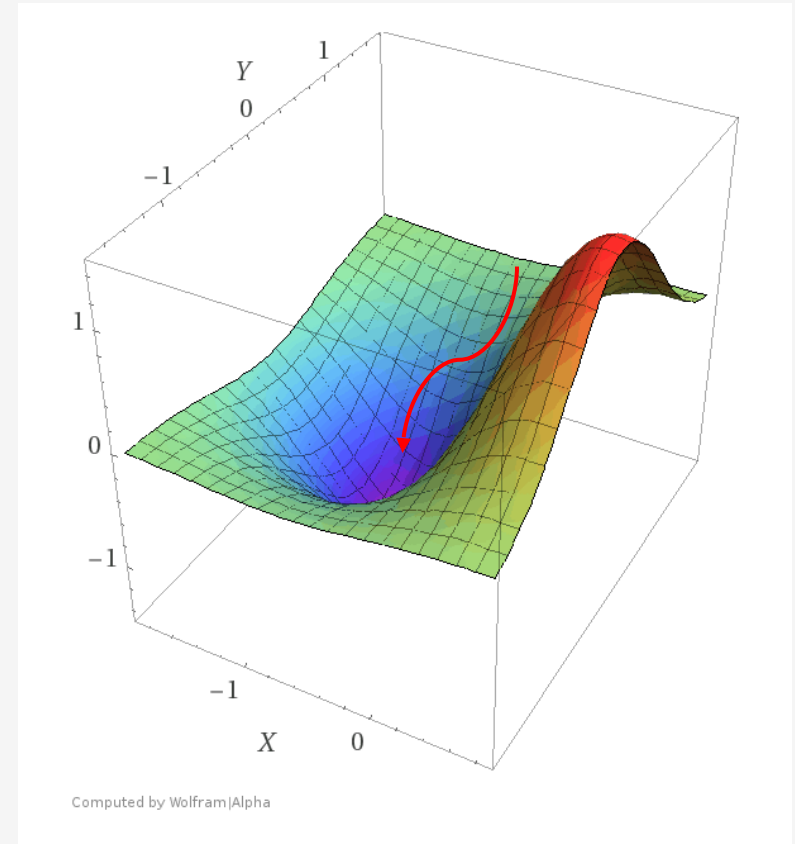
Partial Derivatives

$$f(x, y) = 2x^3 + 3y^3$$

$$\frac{d}{dx}f(x, y) = 6x^2$$

$$\frac{d}{dy}f(x, y) = 9y^2$$

For `(x,y)` values `(1,2)`, the slope with respect to *x* is 6 and the slope with respect to *y* is 36.

$$\frac{d}{dx}f(1, 2) = 6(1)^2 = 6$$

$$\frac{d}{dy}f(1, 2) = 9(2)^2 = 36$$


```python
from sympy import *
from sympy.plotting import plot3d

# Declare x and y to SymPy
x,y = symbols('x y')

# Now just use Python syntax to declare function
f = 2*x**3 + 3*y**3

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)

print(dx_f) # prints 6*x**2
print(dy_f) # prints 9*y**2

# plot the function
plot3d(f)
```
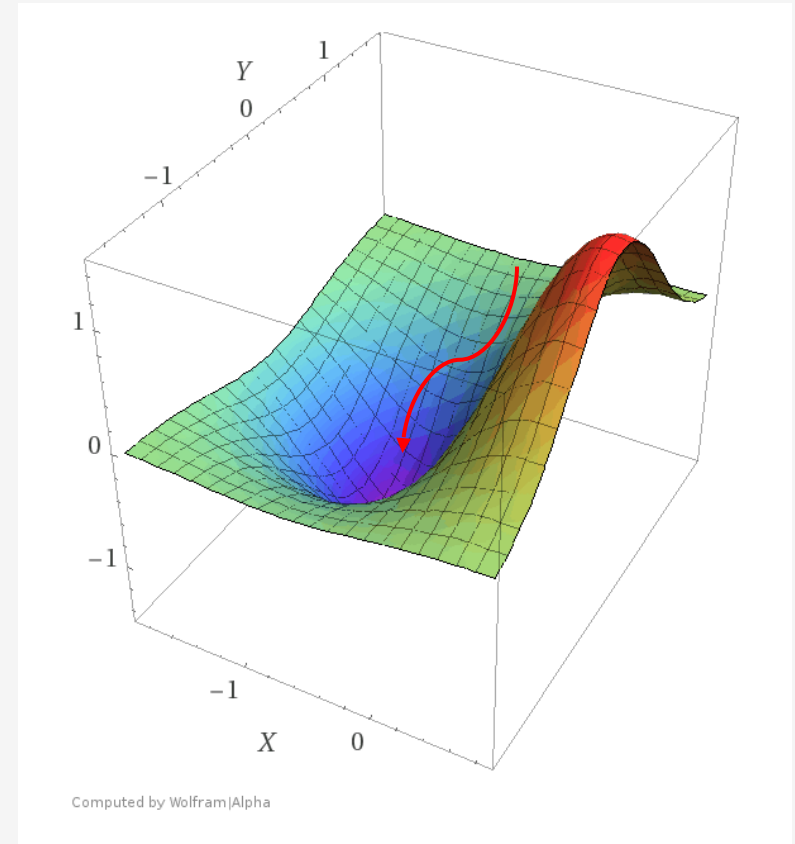
Gradient Descent - Making Calculus Useful
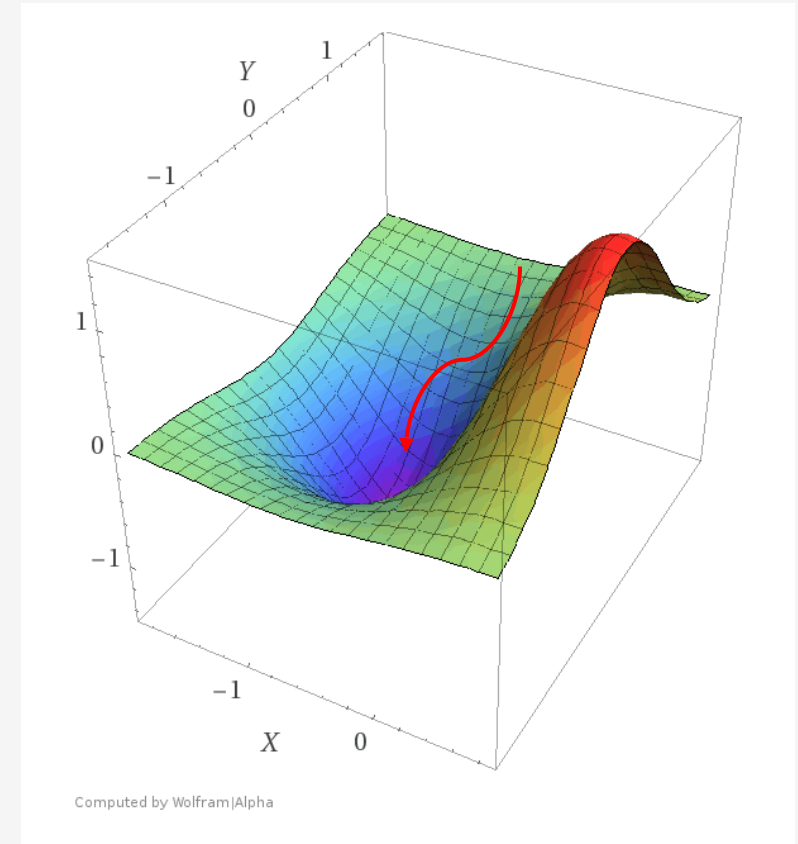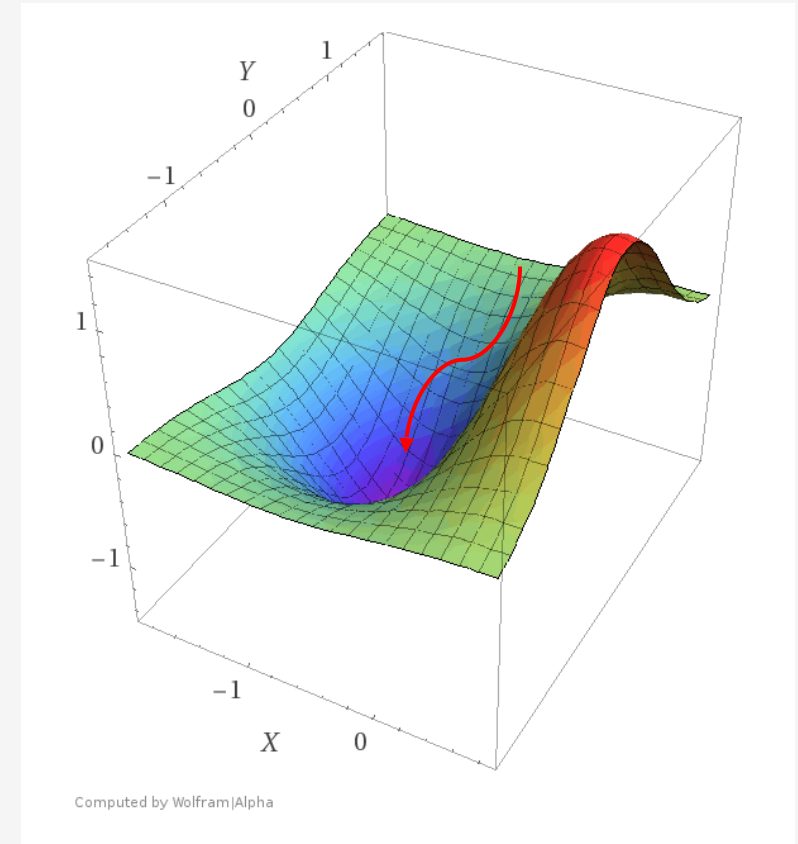

Computed by Wolfram|Alpha

Gradient Descent - Making Calculus Useful

- **Gradient descent** is a continuous/nonlinear optimization method that uses Calculus derivatives to find a local minimum/maximum.


Computed by Wolfram|Alpha

Gradient Descent - Making Calculus Useful

- **Gradient descent** is a continuous/nonlinear optimization method that uses Calculus derivatives to find a local minimum/maximum.

- This methodology has become highly popular due to the need to optimize thousands and even millions of machine learning variables, like neural networks and their node weight values.

Computed by Wolfram|Alpha

Gradient Descent - Making Calculus Useful

- **Gradient descent** is a continuous/nonlinear optimization method that uses Calculus derivatives to find a local minimum/maximum.

- This methodology has become highly popular due to the need to optimize thousands and even millions of machine learning variables, like neural networks and their node weight values.

- While you can use simulated annealing to do these ML optimizations, gradient descent offers a more guided methodology that scales well with extremely large numbers of variables, at the cost of finding a better optimum.

Computed by Wolfram|Alpha

Gradient Descent - Making Calculus Useful

- **Gradient descent** is a continuous/nonlinear optimization method that uses Calculus derivatives to find a local minimum/maximum.

- This methodology has become highly popular due to the need to optimize thousands and even millions of machine learning variables, like neural networks and their node weight values.

- While you can use simulated annealing to do these ML optimizations, gradient descent offers a more guided methodology that scales well with extremely large numbers of variables, at the cost of finding a better optimum.

- The main drawback of gradient descent is it is very mathy and easily gets stuck in local minima, which is why stochastic approaches are used to add randomness.
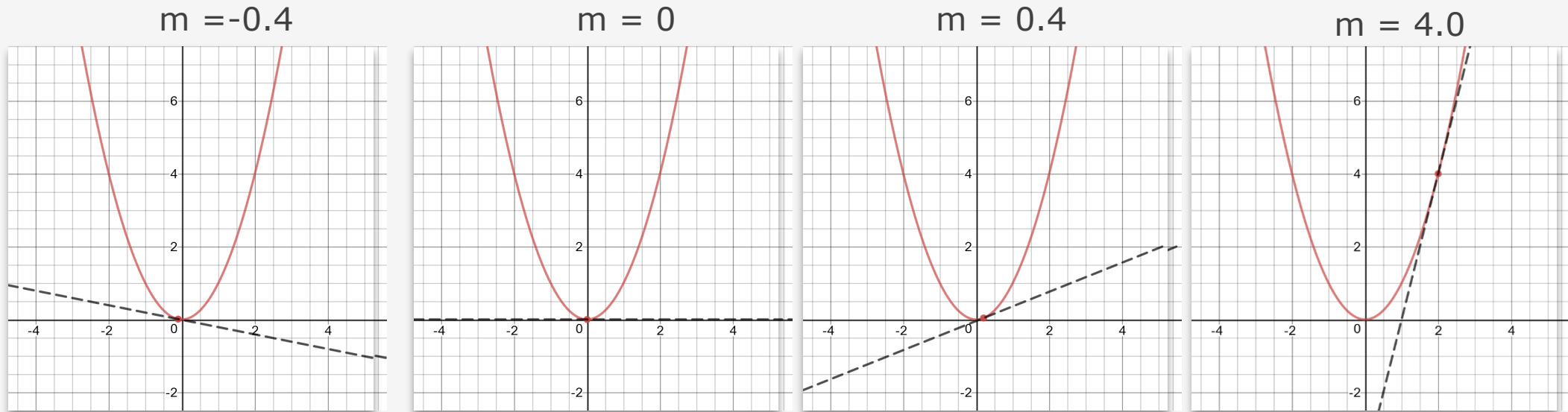

Computed by Wolfram|Alpha
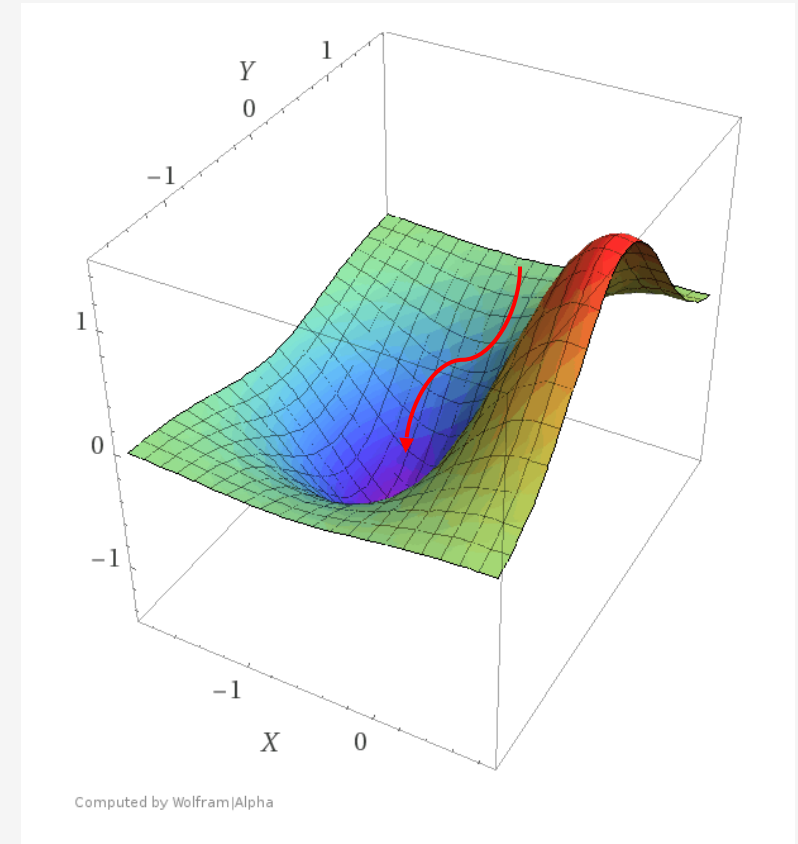
The Essence of Gradient Descent

For a given function, notice how when the slope is zero, we are at a local minimum/maximum!
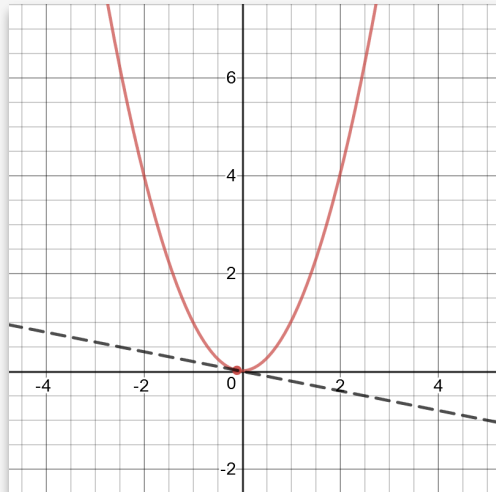
As the slope gets closer to zero, the closer we are to the local minimum /maximum.

Oppositely, the larger/steeper the slope the farther we are from the local minimum.
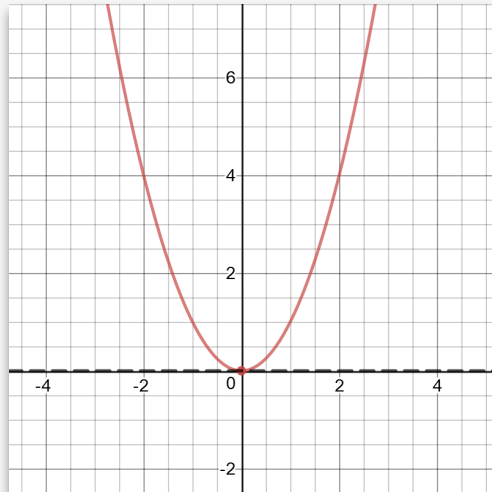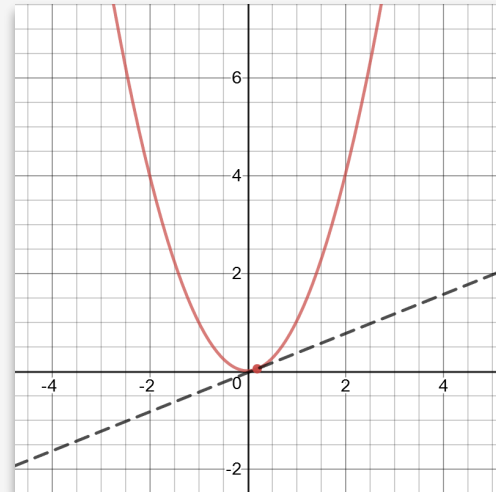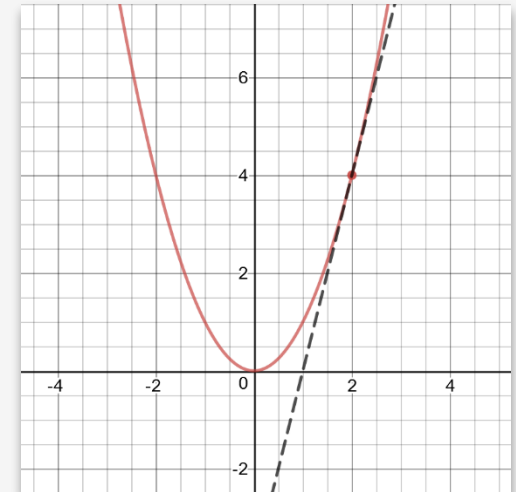

m =-0.4   m = 0   m = 0.4   m = 4.0

https://www.desmos.com/calculator/ruvtlp9zk6

The Essence of Gradient Descent

m =-0.4   m = 0   m = 0.4   m = 4.0

https://www.desmos.com/calculator/ruvtlp9zk6

The Essence of Gradient Descent

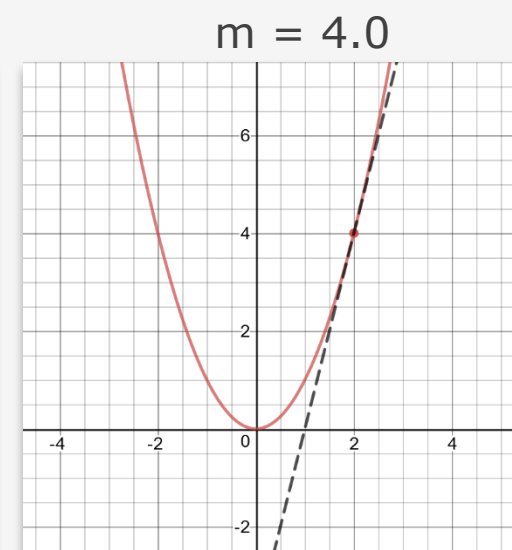Starting our search in a random or arbitrary location, we want it to *step* towards the local minimum.
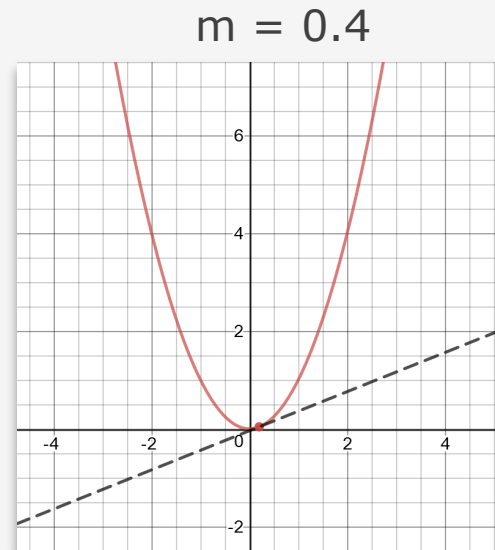
m =-0.4

m = 0

m = 0.4

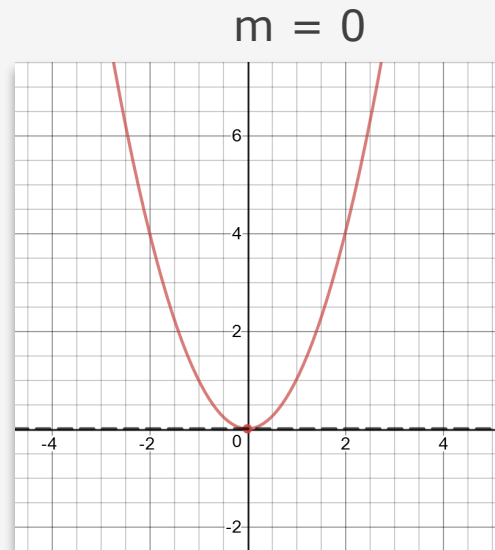m = 4.0

The Essence of Gradient Descent

Starting our search in a random or arbitrary location, we want it to *step* towards the local minimum.

The slope is like a compass providing direction to the local minimum/maximum. We step in directions that make the slope smaller, leading it to 0.

m =-0.4   m = 0   m = 0.4   m = 4.0

https://www.desmos.com/calculator/ruvtlp9zk6
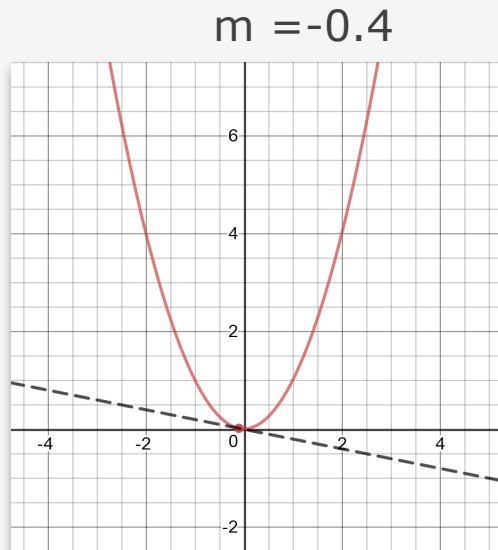
The Essence of Gradient Descent

Starting our search in a random or arbitrary location, we want it to *step* towards the local minimum.

The slope is like a compass providing direction to the local minimum/maximum. We step in directions that make the slope smaller, leading it to 0.

To make this process quicker, we can take bigger steps for bigger slopes, and smaller steps for smaller slopes.

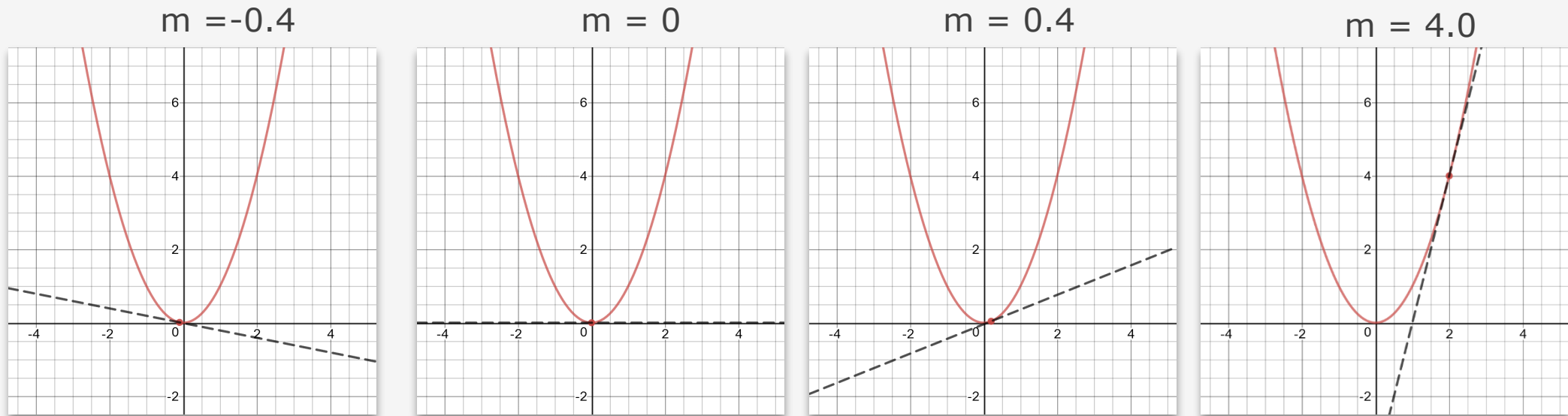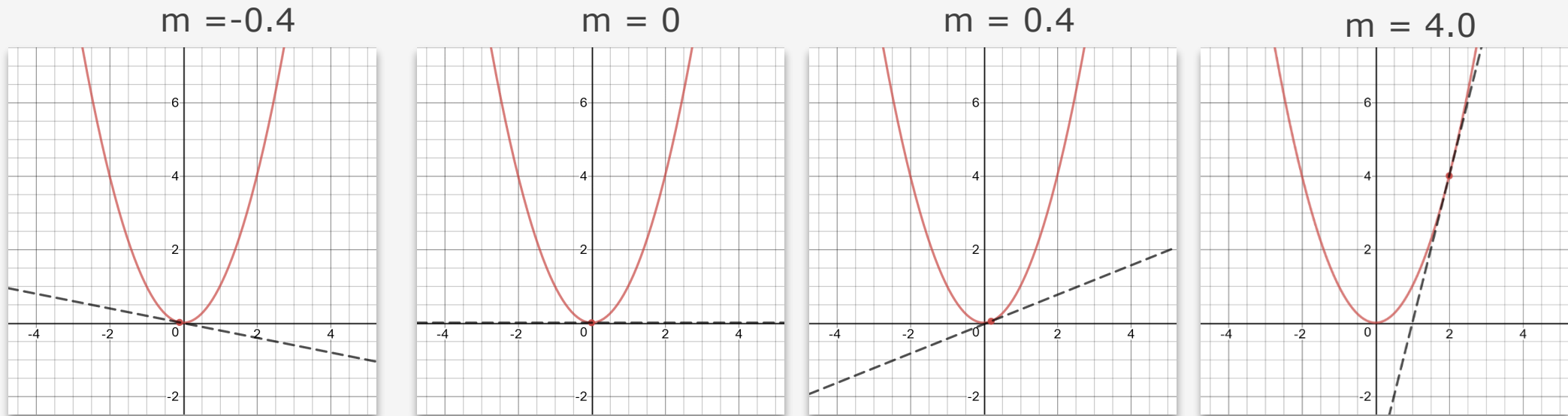m =-0.4   m = 0   m = 0.4   m = 4.0

The Essence of Gradient Descent

Starting our search in a random or arbitrary location, we want it to *step* towards the local minimum.

The slope is like a compass providing direction to the local minimum/maximum. We step in directions that make the slope smaller, leading it to 0.

To make this process quicker, we can take bigger steps for bigger slopes, and smaller steps for smaller slopes.
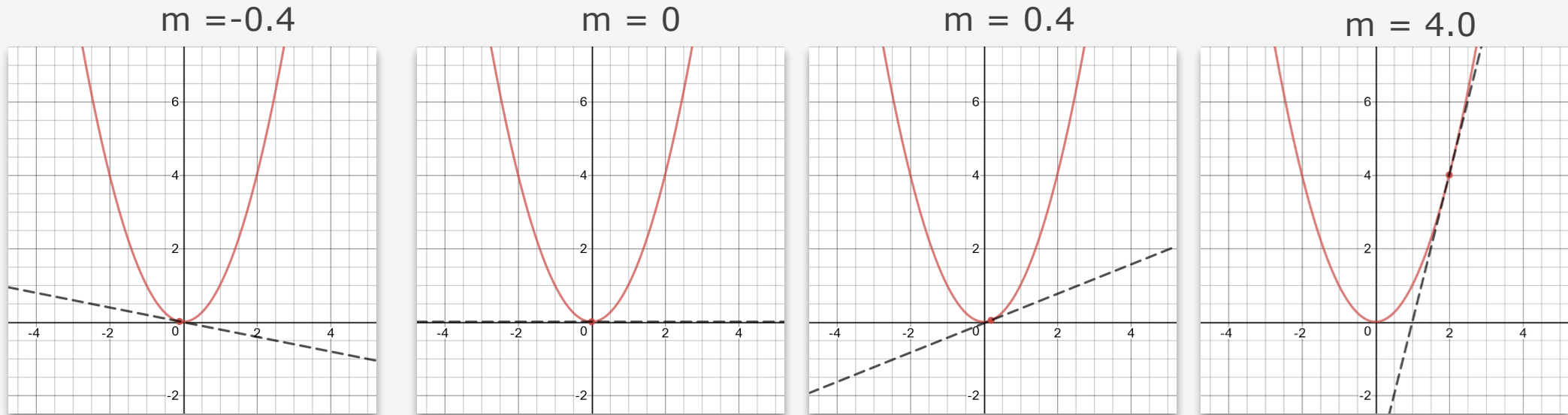
This is the basic idea behind gradient descent.

m =-0.4   m = 0   m = 0.4   m = 4.0

https://www.desmos.com/calculator/ruvtlp9zk6

Gradient Descent with One Variable

Let's walk this through:

We declare our function and derivative

of our function.
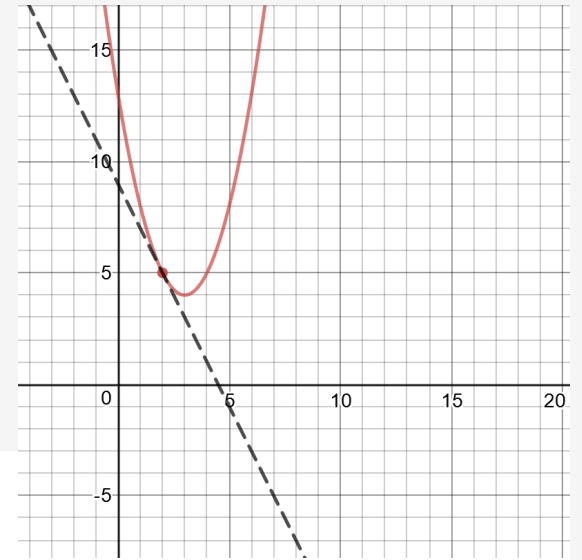

```python
import random

def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)

L = 0.001    # The learning rate
epochs = 100000   # The number of iterations to perform gradient descent

x = random.randint(-15,15)   # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x   # update x by subtracting the (learning rate) * (slope)

print(x, f(x)) # prints 2.999999999999889 4.0
```

Gradient Descent with One Variable

Let's walk this through:

L is our **learning rate**, which scales how small our steps are. Smaller value means smaller steps. Having this too small can make this slow and require more iterations. Having it too large will cause it to keep stepping over the local minimum.

Epochs is a fancy name for number of iterations. You need enough so the search doesn't stop prematurely.


```python
import random


def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)


L = 0.001   # The learning rate
epochs = 100000   # The number of iterations to perform gradient descent

x = random.randint(-15,15)    # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x   # update x by subtracting the (learning rate) * (slope)

print(x, f(x)) # prints 2.99999999999889 4.0
```
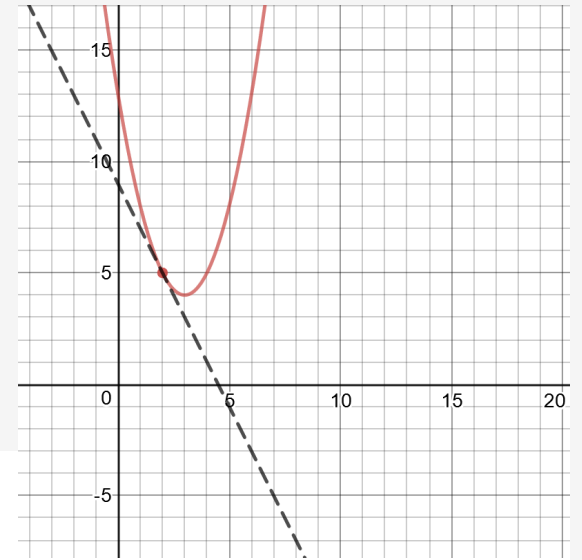
Gradient Descent with One Variable

Let's walk this through:

We start x at a random or arbitrary location. If there are several local minimums, this could have an impact on our answer.

Fortunately, this problem only has one local minimum.

```python
import random


def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)

L = 0.001   # The learning rate
epochs = 100000   # The number of iterations to perform gradient descent

x = random.randint(-15,15)    # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x   # update x by subtracting the (learning rate) * (slope)

print(x, f(x)) # prints 2.999999999999889 4.0
```
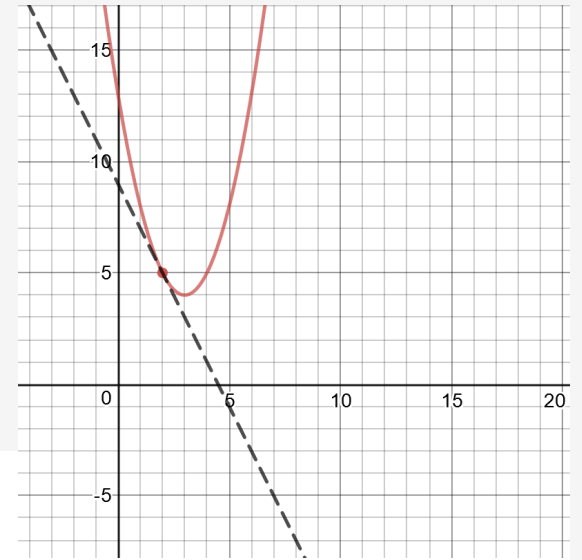
Gradient Descent with One Variable

Let's walk this through:

We use the derivative to calculate the slope and figure out which direction to step *x* towards and by how much.

This means repeatedly subtracting the (slope * learning rate) from *x* to progress towards the minimum.

```python
import random


def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)

L = 0.001   # The learning rate
epochs = 100000   # The number of iterations to perform gradient descent

x = random.randint(-15,15)    # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x   # update x by subtracting the (learning rate) * (slope)

print(x, f(x)) # prints 2.999999999999889 4.0
```
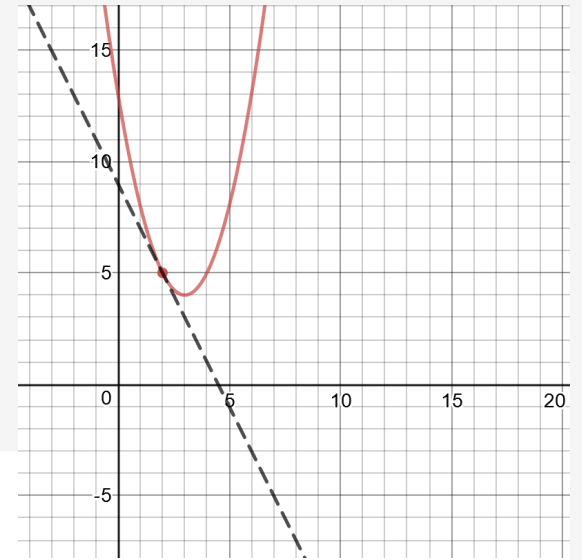
Gradient Descent with One Variable

Let's walk this through:

Finally, if our parameters are right we reach an acceptable local minimum. In this case we can easily eyeball from the graph it's *x=3, y=4*.

Our gradient descent algorithm got close enough at

x=2.999999999999889, y=4.0

```python
import random


def f(x):
    return (x - 3) ** 2 + 4

def dx_f(x):
    return 2*(x - 3)


L = 0.001   # The learning rate
epochs = 100000   # The number of iterations to perform gradient descent

x = random.randint(-15,15)    # start at a random x

for i in range(epochs):
    d_x = dx_f(x) # get slope
    x = x - L * d_x   # update x by subtracting the (learning rate) * (slope)

print(x, f(x)) # prints 2.999999999999889 4.0
```
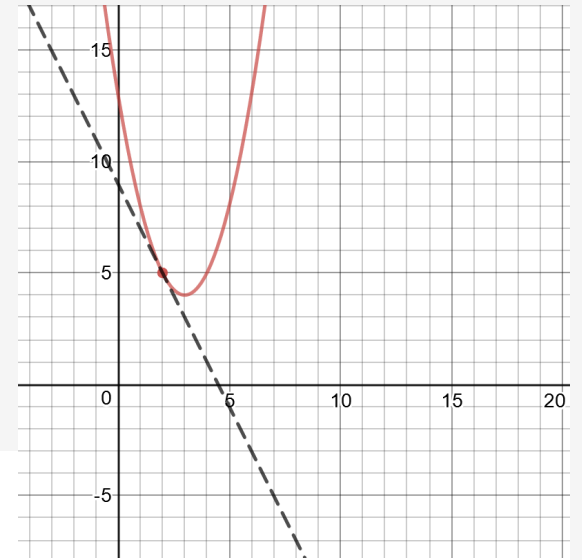
Multivariable Gradient Descent: Partial Derivatives

```python
from sympy import *

# Declare x and y to SymPy
x,y,z = symbols('x y z')

# Now just use Python syntax to declare function
f = (x+10)**2 + (y-3)**2 + (1-z)**2

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)
dz_f = diff(f, z)

print(dx_f) # 2*x + 20
print(dy_f) # 2*y - 6
print(dz_f) # 2*z - 2
```

Multivariable Gradient Descent: Partial Derivatives

Often in machine learning, we are dealing with hundreds, thousands, or millions of variables so we need to learn how to do derivatives for multivariable functions.

```python
from sympy import *

# Declare x and y to SymPy
x,y,z = symbols('x y z')

# Now just use Python syntax to declare function
f = (x+10)**2 + (y-3)**2 + (1-z)**2

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)
dz_f = diff(f, z)

print(dx_f) # 2*x + 20
print(dy_f) # 2*y - 6
print(dz_f) # 2*z - 2
```

Multivariable Gradient Descent: Partial Derivatives

Often in machine learning, we are dealing with hundreds, thousands, or millions of variables so we need to learn how to do derivatives for multivariable functions.

Let's start with a simple nonlinear multivariable function:

```python
from sympy import *

# Declare x and y to SymPy
x,y,z = symbols('x y z')

# Now just use Python syntax to declare function
f = (x+10)**2 + (y-3)**2 + (1-z)**2

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)
dz_f = diff(f, z)

print(dx_f) # 2*x + 20
print(dy_f) # 2*y - 6
print(dz_f) # 2*z - 2
```

Multivariable Gradient Descent: Partial Derivatives

Often in machine learning, we are dealing with hundreds, thousands, or millions of variables so we need to learn how to do derivatives for multivariable functions.

Let's start with a simple nonlinear multivariable function:

$$f(x, y, z) = (x + 10)^2 + (y - 3)^2 + (1 - z)^2$$

```python
from sympy import *

# Declare x and y to SymPy
x,y,z = symbols('x y z')

# Now just use Python syntax to declare function
f = (x+10)**2 + (y-3)**2 + (1-z)**2

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)
dz_f = diff(f, z)

print(dx_f) # 2*x + 20
print(dy_f) # 2*y - 6
print(dz_f) # 2*z - 2
```

Multivariable Gradient Descent: Partial Derivatives

Often in machine learning, we are dealing with hundreds, thousands, or millions of variables so we need to learn how to do derivatives for multivariable functions.

Let's start with a simple nonlinear multivariable function:

$$f(x, y, z) = (x + 10)^2 + (y - 3)^2 + (1 - z)^2$$

This function has several variables, and we will want to isolate just one of the variables and find its slope.

```python
from sympy import *

# Declare x and y to SymPy
x,y,z = symbols('x y z')

# Now just use Python syntax to declare
function
f = (x+10)**2 + (y-3)**2 + (1-z)**2

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)
dz_f = diff(f, z)

print(dx_f) # 2*x + 20
print(dy_f) # 2*y - 6
print(dz_f) # 2*z - 2
```

Multivariable Gradient Descent: Partial Derivatives

Often in machine learning, we are dealing with hundreds, thousands, or millions of variables so we need to learn how to do derivatives for multivariable functions.

Let's start with a simple nonlinear multivariable function:

$$f(x, y, z) = (x + 10)^2 + (y - 3)^2 + (1 - z)^2$$

This function has several variables, and we will want to isolate just one of the variables and find its slope.

$$\frac{d}{dx} f(x, y, z) = 2(x + 10)$$

```python
from sympy import *

# Declare x and y to SymPy
x,y,z = symbols('x y z')

# Now just use Python syntax to declare
function
f = (x+10)**2 + (y-3)**2 + (1-z)**2

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)
dz_f = diff(f, z)

print(dx_f) # 2*x + 20
print(dy_f) # 2*y - 6
print(dz_f) # 2*z - 2
```

Multivariable Gradient Descent: Partial Derivatives

Often in machine learning, we are dealing with hundreds, thousands, or millions of variables so we need to learn how to do derivatives for multivariable functions.

Let's start with a simple nonlinear multivariable function:

$$f(x, y, z) = (x + 10)^2 + (y - 3)^2 + (1 - z)^2$$

This function has several variables, and we will want to isolate just one of the variables and find its slope.

$$\frac{d}{dx} f(x, y, z) = 2(x + 10)$$

This is called a partial derivative, and it helps identify the rate of change for just that variable.

```python
from sympy import *

# Declare x and y to SymPy
x,y,z = symbols('x y z')

# Now just use Python syntax to declare
function
f = (x+10)**2 + (y-3)**2 + (1-z)**2

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)
dz_f = diff(f, z)

print(dx_f) # 2*x + 20
print(dy_f) # 2*y - 6
print(dz_f) # 2*z - 2
```

Multivariable Gradient Descent: Partial Derivatives

Often in machine learning, we are dealing with hundreds, thousands, or millions of variables so we need to learn how to do derivatives for multivariable functions.

Let's start with a simple nonlinear multivariable function:

$$f(x, y, z) = (x + 10)^2 + (y - 3)^2 + (1 - z)^2$$

This function has several variables, and we will want to isolate just one of the variables and find its slope.

$$\frac{d}{dx} f(x, y, z) = 2(x + 10)$$

This is called a partial derivative, and it helps identify the rate of change for just that variable.

Using SymPy, we can calculate partial derivatives for the each variable using $\frac{d}{dx}, \frac{d}{dy},$ and $\frac{d}{dz}$ on the function as shown on the right.

```python
from sympy import *

# Declare x and y to SymPy
x,y,z = symbols('x y z')

# Now just use Python syntax to declare
function
f = (x+10)**2 + (y-3)**2 + (1-z)**2

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)
dz_f = diff(f, z)

print(dx_f) # 2*x + 20
print(dy_f) # 2*y - 6
print(dz_f) # 2*z - 2
```

Multivariable Gradient Descent

We now manage three derivatives and not just one, and use them to adjust the three variables respectively.

Using gradient descent, we roughly minimize (x,y,z) to numbers close to (-10, 3, 1) and closely reach a minimum of 0.
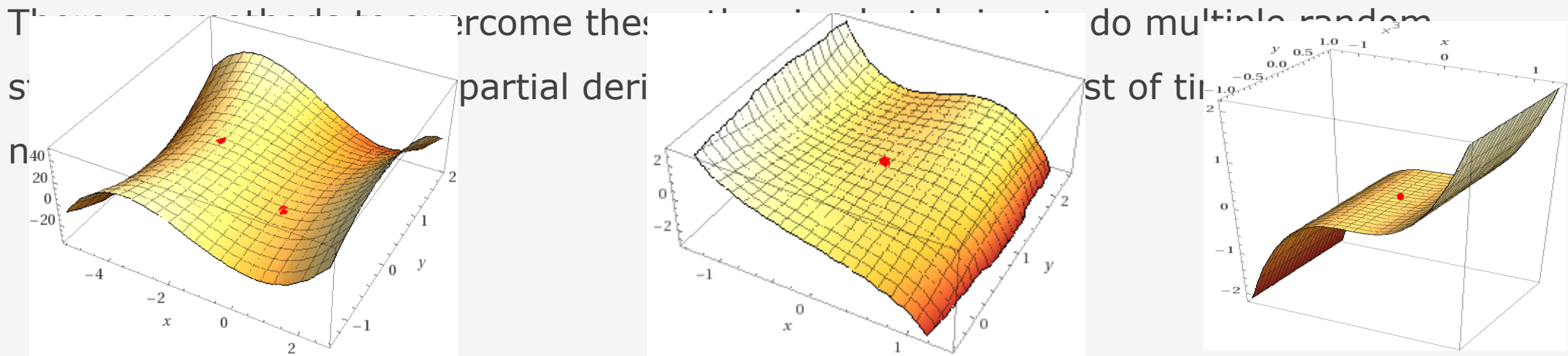
Congrats! You are doing multivariable calculus.

```python
def f(x,y,z):
    return (x+10)**2 + (y-3)**2 + (1-z)**2

def dx_f(x):
    return 2*(x+10)

def dy_f(y):
    return 2*(y-3)

def dz_f(z):
    return 2*(z-1)

L = 0.0001  # The learning rate
epochs = 1000000   # The number of iterations to perform gradient descent

x = 0    # we will find the x for the minimum
y = 0    # we will find the y for the minimum
z = 0    # we will find the z for the minimum

for i in range(epochs):
    d_x = dx_f(x) # get x slope
    d_y = dy_f(y) # get y slope
    d_z = dz_f(z) # get z slope

    x = x - L * d_x   # update x
    y = y - L * d_y   # update y
    z = z - L * d_z   # update z

# -9.999999999995559 2.9999999999988898 0.999999999997224
2.1031154992708616e-23
print(x, y, z, f(x,y,z))
```

When Slopes of 0 are Not Minimums/Maximums

There are some awkward special cases when all variables reach a slope of 0 but are not the local minimum or maximum.

These are called saddle points.

There are methods to overcome these ... but basically ... do multiple random ... partial deri ... st of ti ... n ...

Integrals

The opposite of a derivative is an **integral**, which finds an area under a function's curve.

Let's look at a function that is curvy and hard to find the area under.

$f(x) = x^2$

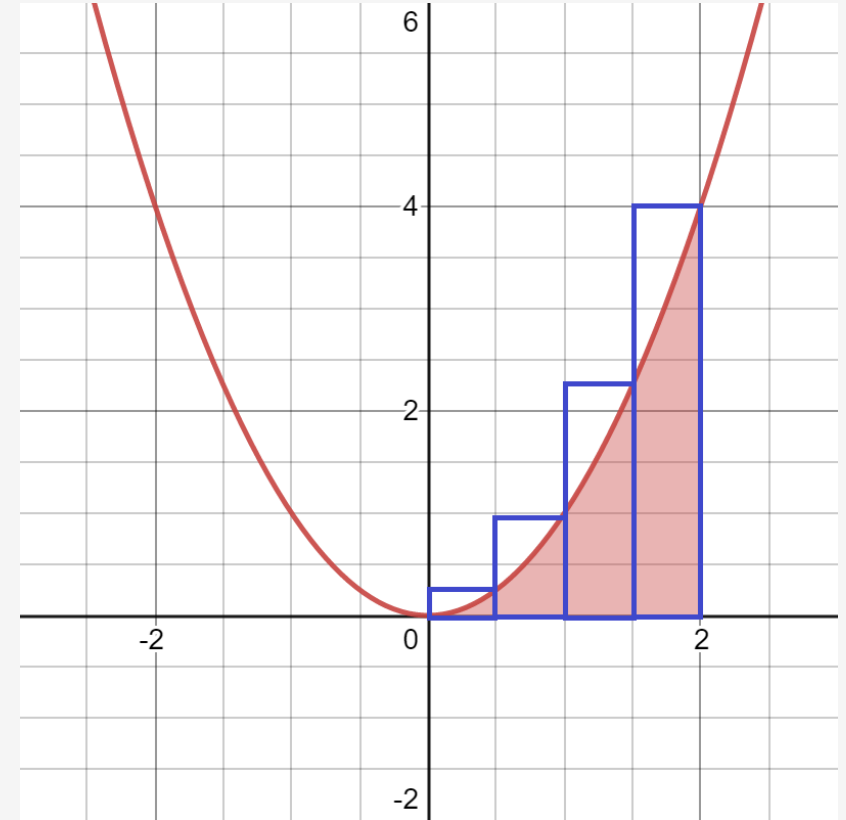I want to find the area between 0 and 2 for this function as shaded in red to the right

How is it done?
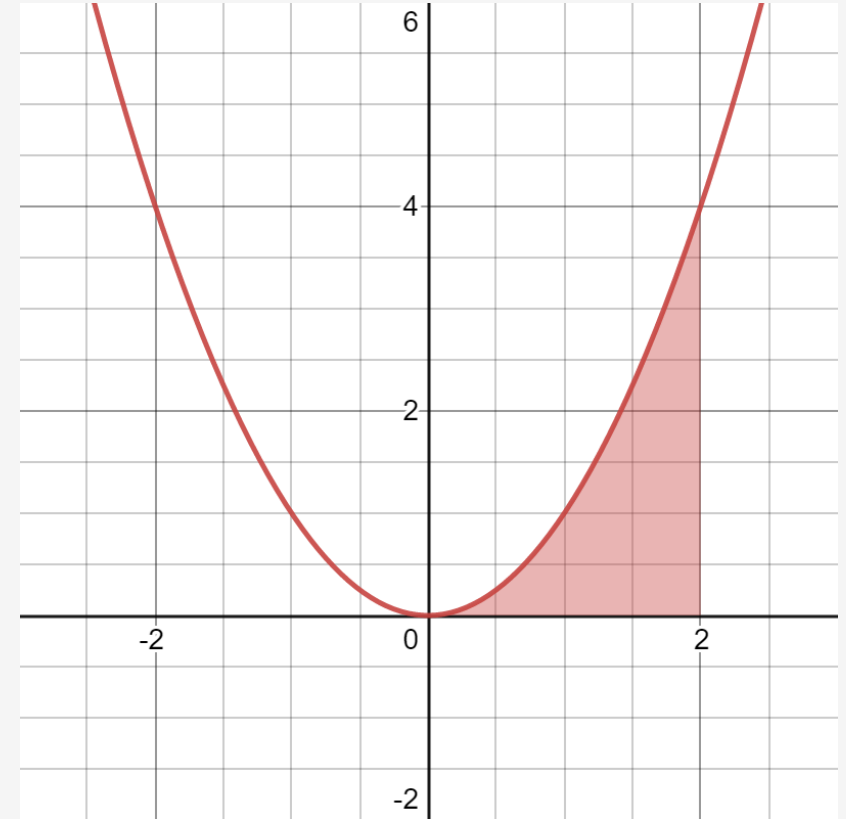
Integrals

What if we were to pack some rectangles of equal width under the curve and sum their areas?

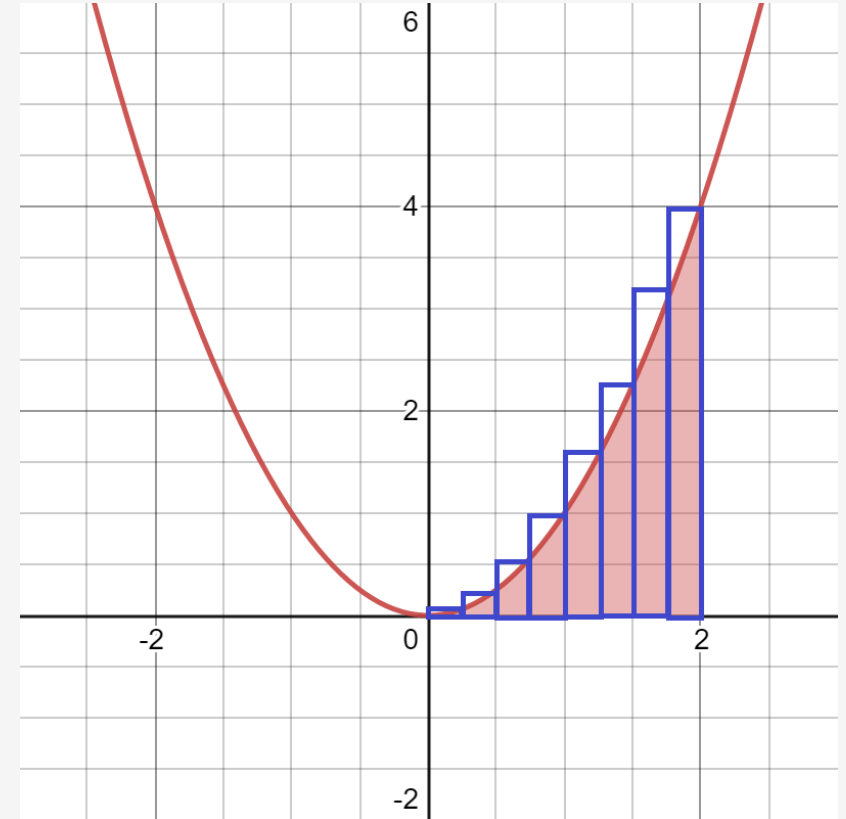Would 4 rectangles give us a good estimate?

Integrals

What if we were to pack some rectangles of equal width under the curve and sum their areas?

Would 4 rectangles give us a good estimate?

What about 8 rectangles?

Is it reasonable to say the more rectangles we pack, the closer the sum of their areas approaches the actual area under the curve?

Integrals – Reimann Sums

We can use Python to approximate the area using 10,000 rectangles and we get a very close answer of 2.666.

```python
def approximate_integral(a, b, n, f):
    delta_x = (b - a) / n
    total_sum = 0

    for i in range(1, n + 1):
        midpoint = 0.5 * (2 * a + delta_x * (2 * i - 1))
        total_sum += f(midpoint)

    return total_sum * delta_x

def my_function(x):
    return x ** 2

area = approximate_integral(a=0, b=2, n=10000,
f=my_function)

print(area)  # prints 2.66666665999999
```
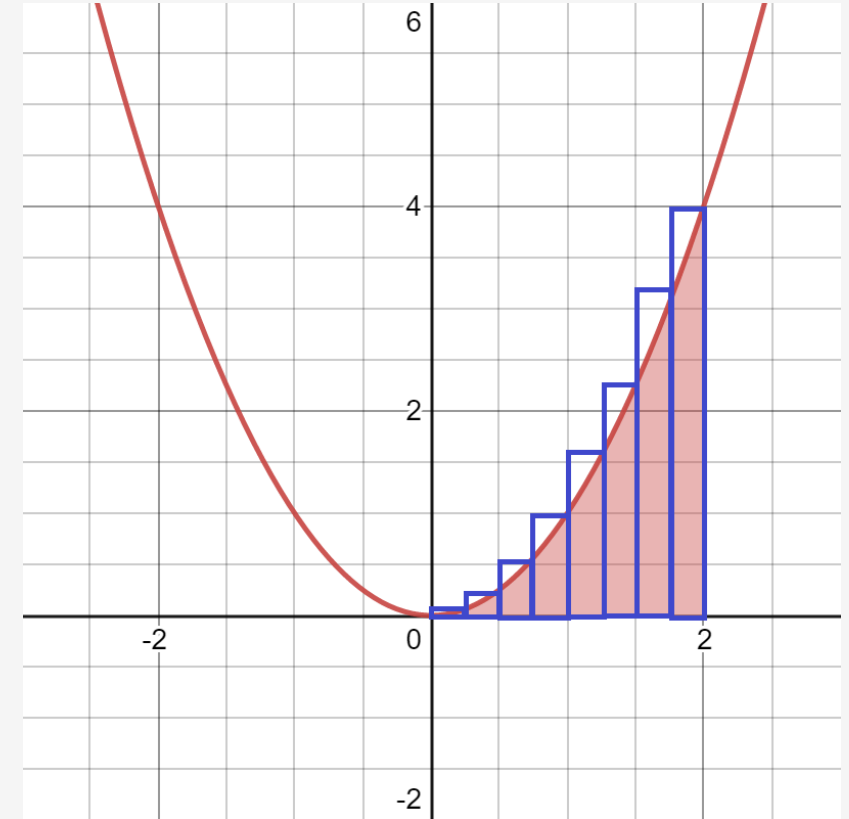
Integrals – Using Limits and Reimann Sum

If you want to extend this idea formally, you can use limits to approach an infinite number of rectangles and see it converges on an exact area of 8/3.

```python
from sympy import *

# Declare variables to SymPy
x, i, n = symbols('x i n')

# Declare function and range
f = x**2
lower, upper = 0, 2

# Calculate width and each rectangle height at index "i"
delta_x = ((upper - lower) / n)
x_i = (lower + delta_x * i)
fx_i = f.subs(x, x_i)

# Iterate all "n" rectangles and sum their areas
n_rectangles = Sum(delta_x * fx_i, (i, 1, n)).doit()

# Calculate the area by approaching the number
# of rectangles "n" to infinity
area = limit(n_rectangles, n, oo)

print(area) # prints 8/3
```
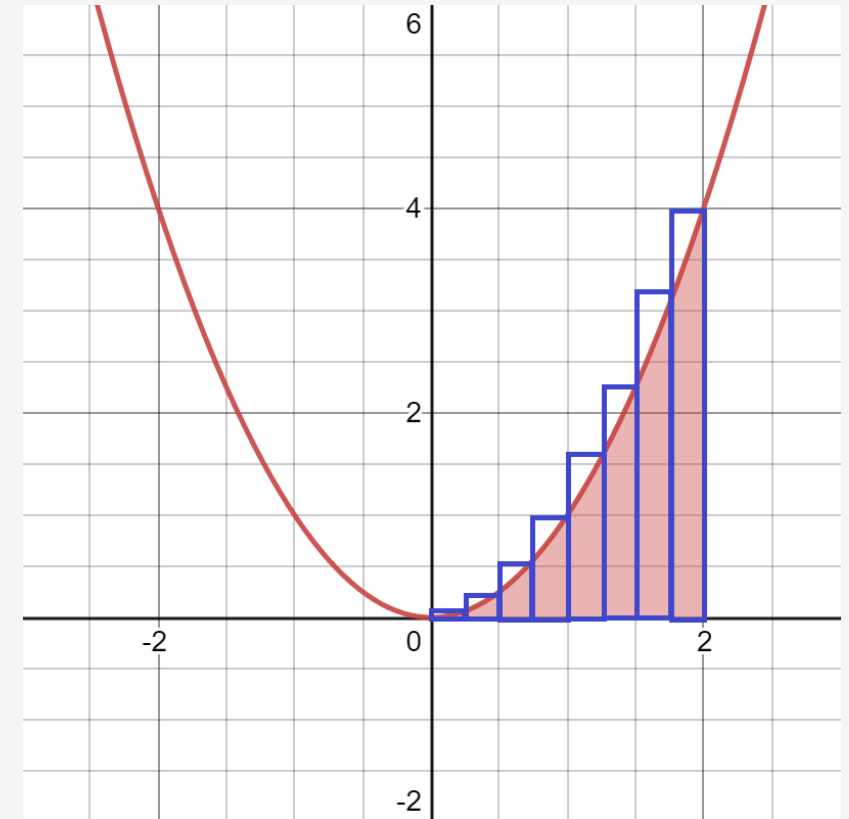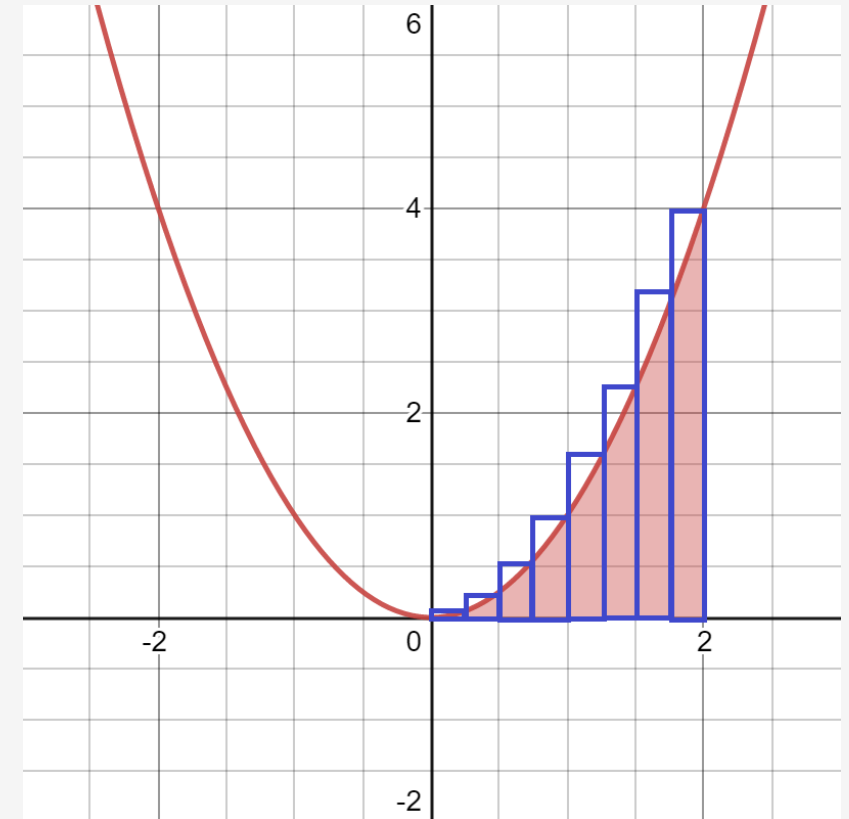
Integrals – Using SymPy

The easiest exact approach you can use is to simply use SymPy's integration function.

```python
from sympy import *

# Declare 'x' to SymPy
x = symbols('x')

# Now just use Python syntax to declare function
f = x**2

# Calculate the integral of the function with respect to x
# for the area between x = 0 and 2
area = integrate(f, (x, 0, 2))

print(area) # prints 8/3
```

, and the

or you.

Applied Example – Integrating the Normal Distribution

```python
import math


def normal_pdf(x: float, mean: float, std_dev: float) -> float:
    return (1.0 / (2.0 * math.pi * std_dev ** 2) ** 0.5) * math.exp(-1.0 * ((x - mean) ** 2 / (2.0 * std_dev
** 2)))


def approximate_integral(a, b, n, f):
    delta_x = (b - a) / n
    total_sum = 0

    for i in range(1, n + 1):
        midpoint = 0.5 * (2 * a + delta_x * (2 * i - 1))
        total_sum += f(midpoint)

    return total_sum * delta_x


p_between_61_and_62 = approximate_integral(a=61, b=62, n=7, f= lambda x: normal_pdf(x,64.43,2.99))

print(p_between_61_and_62)
```
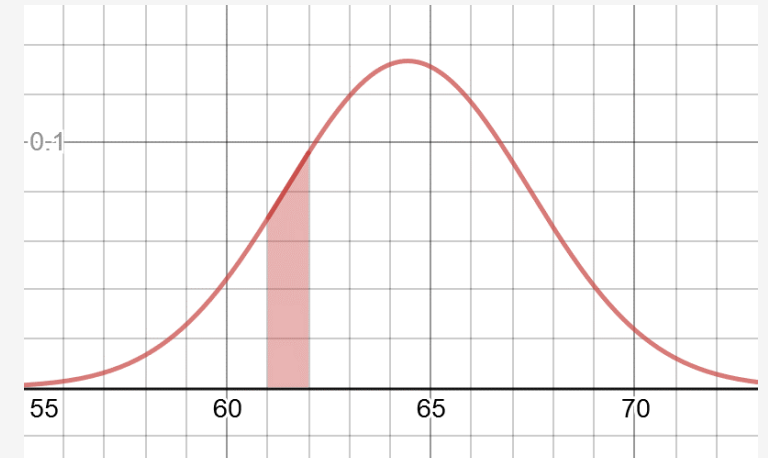
Applied Example – Integrating the Normal Distribution

```python
import math


def normal_pdf(x: float, mean: float, std_dev: float) -> float:
    return (1.0 / (2.0 * math.pi * std_dev ** 2) ** 0.5) * math.exp(-1.0 * ((x - mean) ** 2 / (2.0 * std_dev ** 2)))


def approximate_integral(a, b, n, f):
    delta_x = (b - a) / n
    total_sum = 0

    for i in range(1, n + 1):
        midpoint = 0.5 * (2 * a + delta_x * (2 * i - 1))
        total_sum += f(midpoint)

    return total_sum * delta_x


p_between_61_and_62 = approximate_integral(a=61, b=62, n=7, f= lambda x: normal_pdf(x,64.43,2.99))

print(p_between_61_and_62)
```
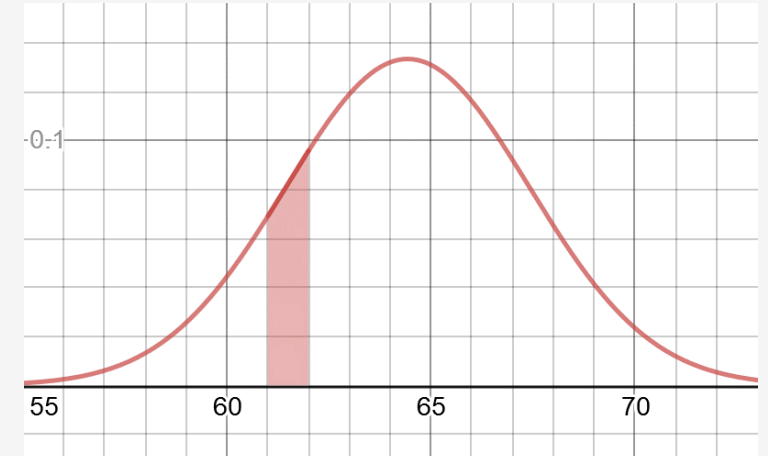

A common use case for integration in data science is finding areas under probability distributions to get probabilities.

Applied Example – Integrating the Normal Distribution

```python
import math


def normal_pdf(x: float, mean: float, std_dev: float) -> float:
    return (1.0 / (2.0 * math.pi * std_dev ** 2) ** 0.5) * math.exp(-1.0 * ((x - mean) ** 2 / (2.0 * std_dev
** 2)))


def approximate_integral(a, b, n, f):
    delta_x = (b - a) / n
    total_sum = 0

    for i in range(1, n + 1):
        midpoint = 0.5 * (2 * a + delta_x * (2 * i - 1))
        total_sum += f(midpoint)

    return total_sum * delta_x


p_between_61_and_62 = approximate_integral(a=61, b=62, n=7, f= lambda x: normal_pdf(x,64.43,2.99))

print(p_between_61_and_62)
```
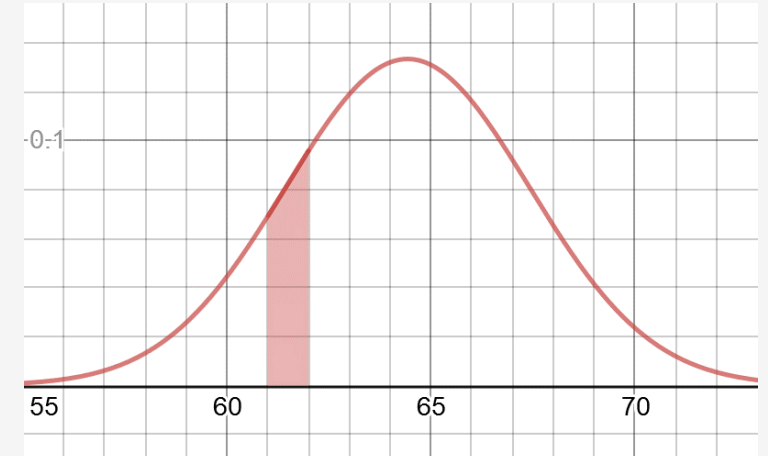

A common use case for integration in data science is finding areas under probability distributions to get probabilities.

Running the code above, we get a probability of 8.25% that any given golden retriever's weight is between 61 and 62 pounds.

Applied Example – Integrating the Normal Distribution

```python
import math


def normal_pdf(x: float, mean: float, std_dev: float) -> float:
    return (1.0 / (2.0 * math.pi * std_dev ** 2) ** 0.5) * math.exp(-1.0 * ((x - mean) ** 2 / (2.0 * std_dev ** 2)))


def approximate_integral(a, b, n, f):
    delta_x = (b - a) / n
    total_sum = 0

    for i in range(1, n + 1):
        midpoint = 0.5 * (2 * a + delta_x * (2 * i - 1))
        total_sum += f(midpoint)

    return total_sum * delta_x


p_between_61_and_62 = approximate_integral(a=61, b=62, n=7, f= lambda x: normal_pdf(x,64.43,2.99))

print(p_between_61_and_62)
```
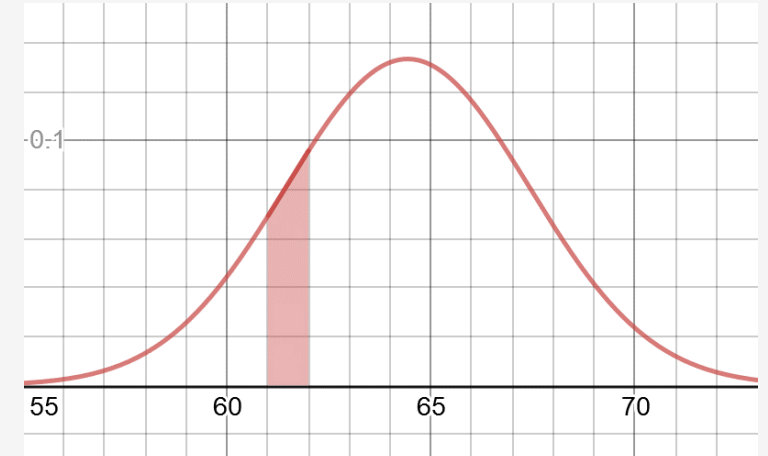

A common use case for integration in data science is finding areas under probability distributions to get probabilities.

Running the code above, we get a probability of 8.25% that any given golden retriever's weight is between 61 and 62 pounds.

Further Resources

3Blue1Brown – Essence of Calculus

https://www.youtube.com/playlist?list=PLZHQObOWTQDMsr9K-rj53DwVRMYO3t5Yr

SymPy Documentation

https://docs.sympy.org/latest/index.html