trigonometry-through-python-2

April 17, 2025

0.1 Trigonometry with Python

0.1.1 What's Trigonometry?

Trigonometry is a part of math that focuses on the connection between angles and the lengths of sides in triangles, especially right-angled triangles (which have one 90° angle). It's a useful tool for solving problems involving shapes, heights, distances, and many real-world situations.

Why learn it? Trigonometry helps us: - Measure the height of buildings or mountains
- Understand waves and sound
- Track planets and build computer graphics
- Even make beautiful animations and art!

In Python, we'll use:

- `math`: for basic trig functions like `sin()`, `cos()`, and `tan()`
- `numpy`: to handle angles and do fast number crunching
- `matplotlib`: to draw cool plots of trig functions and shapes

---

[129]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
```

0.1.2 Degrees vs Radians

When working with trigonometry in Python (or most programming languages), it's important to know that **Python's math functions use radians, not degrees**.

What's the difference?

- **Degrees** are what we usually use in everyday math (like 90°, 180°).
- **Radians** are what Python uses internally for trigonometric functions like `sin()`, `cos()`, and `tan()`.

For reference: - 180° = radians
- 360° = 2 radians
- 90° = /2 radians

So, if you plug `math.sin(90)` into Python, you **won't get 1**, because 90 is treated as **90 radians**, not degrees.

How to convert Python's `math` module has built-in functions to help:

- **Degrees Radians**:

  `math.radians(degrees)`

- **Radians Degrees**:

  `math.degrees(radians)`

[186]:
```python
angle_deg = 90
angle_rad = math.radians(angle_deg)
print(f"{angle_deg}° = {angle_rad:.2f} radians")
```

```
90° = 1.57 radians
```

[188]:
```python
print(f"{angle_rad} radians = {math.degrees(angle_rad)}°")
```

```
1.5707963267948966 radians = 90.0°
```

0.1.3 Trig Basics in Code

In trigonometry, the three main functions are **sine (sin)**, **cosine (cos)**, and **tangent (tan)**. These functions relate the angles of a right triangle to the ratios of its sides. In Python, we can use the `math` module to calculate these values for any angle. Just remember: these functions use **radians**, not degrees, so we often need to convert degrees to radians using `math.radians()` before using them.

[193]:
```python
angle_deg = 30
angle_rad = math.radians(angle_deg)

sin_val = math.sin(angle_rad)
cos_val = math.cos(angle_rad)
tan_val = math.tan(angle_rad)

print(f"sin({angle_deg}°) = {sin_val:.3f}")
print(f"cos({angle_deg}°) = {cos_val:.3f}")
print(f"tan({angle_deg}°) = {tan_val:.3f}")
```

```
sin(30°) = 0.500
cos(30°) = 0.866
tan(30°) = 0.577
```

Create a Table of Values In trigonometry, it's helpful to know the sine and cosine values for common angles like 0°, 30°, 45°, 60°, and 90°. These values represent the ratios of the sides of a right triangle for each angle. By creating a table of these values, we can quickly reference the sine and cosine for these angles without needing to calculate them each time.

In Python, we can create a table using a loop that calculates the sine and cosine for each angle. Here's how we can do it:

- For each angle (in degrees), we first convert it to radians (since Python uses radians for trig functions).
- Then, we calculate the sine and cosine of that angle using `math.sin()` and `math.cos()`.
- Finally, we display the results in a neat table format.

This makes it easy to see the relationship between angles and their sine/cosine values.

[196]:
```python
print(f"{'Angle (°)':<10}{'sin( )':>10}{'cos( )':>10}")
print("-" * 30)

for angle in range(0, 361, 30):
    rad = math.radians(angle)
    print(f"{angle:<10}{math.sin(rad):>10.3f}{math.cos(rad):>10.3f}")
```

```
Angle (°)     sin( )    cos( )
------------------------------
0              0.000     1.000
30             0.500     0.866
60             0.866     0.500
90             1.000     0.000
120            0.866    -0.500
150            0.500    -0.866
180            0.000    -1.000
210           -0.500    -0.866
240           -0.866    -0.500
270           -1.000    -0.000
300           -0.866     0.500
330           -0.500     0.866
360           -0.000     1.000
```

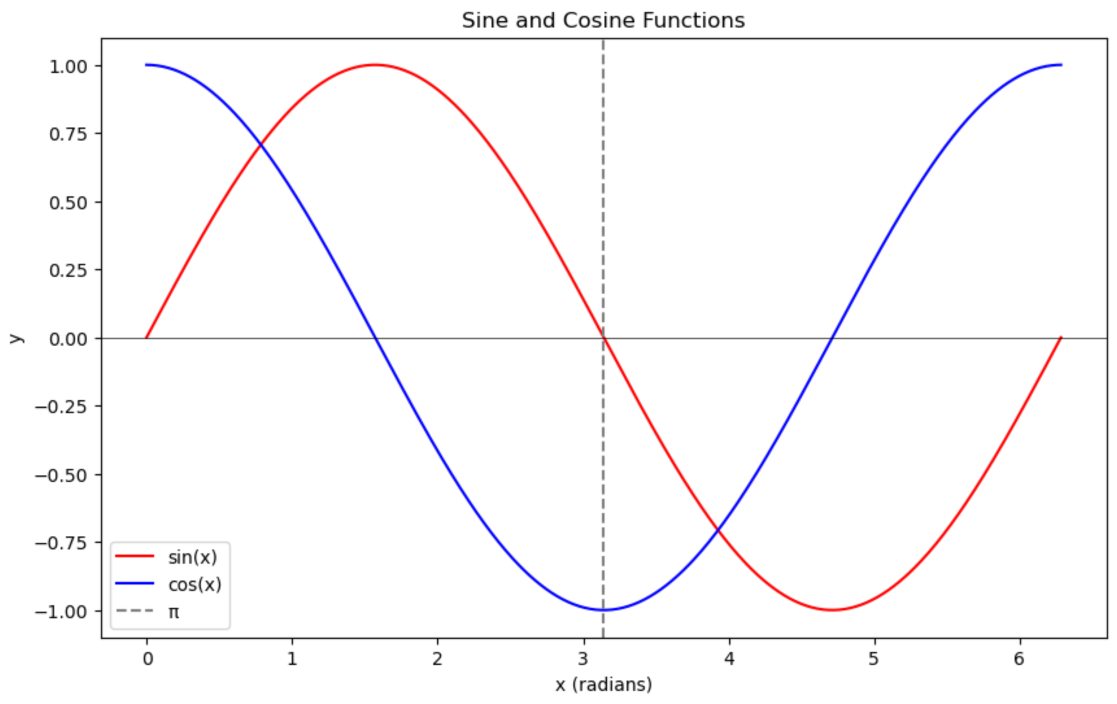
0.1.4 Visualizing Trig Functions

Visualizing sine and cosine functions helps us understand how they repeat in a smooth, wave-like pattern. By plotting these functions on a graph, we can see their behavior over a full cycle (from 0 to $2 \cdot pi$ radians). This gives us a clear picture of how the values of sine and cosine change as the angle increases, creating oscillating waves that repeat every $2 \cdot pi$ radians.

[203]:
```python
x = np.linspace(0, 2 * np.pi, 500)  # 0 to 2  radians
sin_y = np.sin(x)
cos_y = np.cos(x)

plt.figure(figsize=(10, 6))

# Plot sine and cosine
plt.plot(x, sin_y, label="sin(x)", color="red")
plt.plot(x, cos_y, label="cos(x)", color="blue")
plt.title("Sine and Cosine Functions")
plt.xlabel("x (radians)")
plt.ylabel("y")
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(np.pi, color='gray', linestyle='--', label=" ")
plt.legend()
plt.show()
```



0.1.5 Art Through Trigonometry

Trigonometry isn't just for solving math problems—it can also be used to create **beautiful art**! By using **sine**, **cosine**, and **tangent** functions in parametric equations, we can generate intricate shapes and patterns. These functions describe smooth, repetitive motions, which can be used to create spirals, waves, petals, and even more complex designs. By adjusting parameters like the radius, angle, and color, we can create visually stunning artwork that's both mathematical and creative.

0.1.6 Spiral Using Polar Coordinates

In this example, we use **polar coordinates** to create a beautiful spiral. By defining `r =` (where `r` is the radius and is the angle), we create a **spiral pattern** that expands outward as the angle increases. This is a perfect example of using trigonometric functions to generate smooth, continuous shapes.

Code Breakdown:

1. **Define the Range of** :
   `theta = np.linspace(0, 6 * np.pi, 1000)` generates 1000 equally spaced values for the angle   from 0 to (6), which covers multiple full rotations.

2. **Set the Radius**:
   `r = theta` means the radius grows linearly as the angle increases, creating the spiral effect.

3. **Calculate x and y Coordinates**:
   Using the parametric equations:

   - `x = r * np.cos(theta)`

   - `y = r * np.sin(theta)`
     we calculate the `x` and `y` positions for each value of   to plot the spiral.

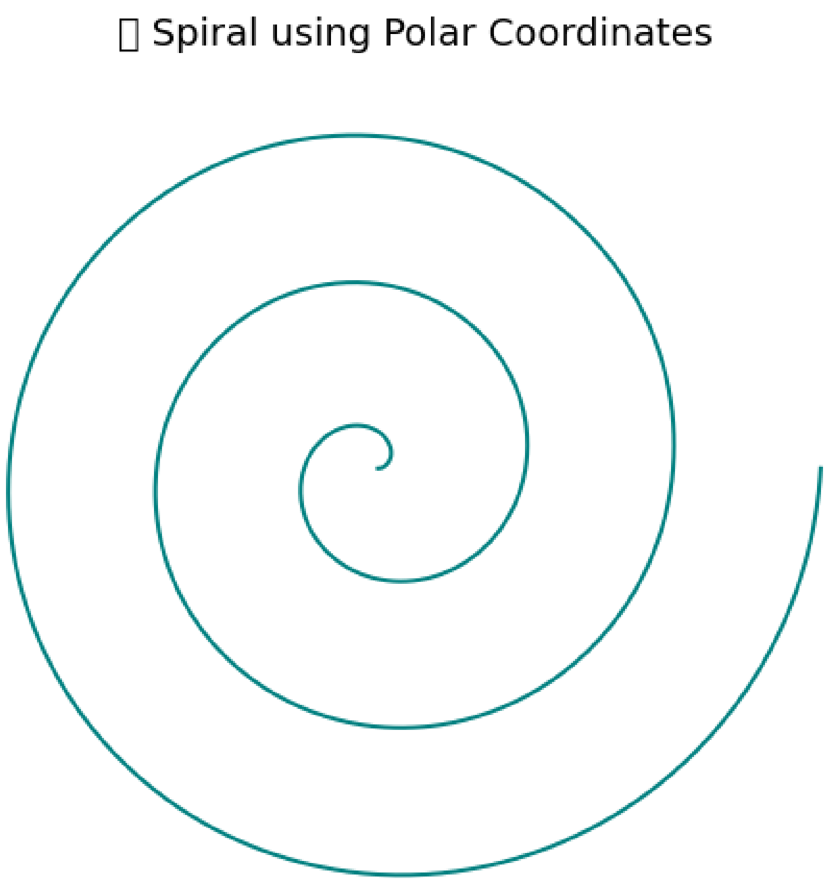
4. **Plot the Spiral**:

   - `plt.figure(figsize=(6, 6))` ensures the plot is square-shaped.
   - `plt.plot(x, y, color="teal")` draws the spiral with a teal-colored line.
   - `plt.title("  Spiral using Polar Coordinates", fontsize=14)` adds a title to the plot.
   - `plt.axis("equal")` ensures the plot has equal scaling for both axes, keeping the spiral symmetrical.
   - `plt.axis("off")` removes the axes for a cleaner look.

[210]:
```python
theta = np.linspace(0, 6 * np.pi, 1000)
r = theta

x = r * np.cos(theta)
y = r * np.sin(theta)

plt.figure(figsize=(6, 6))
plt.plot(x, y, color="teal")
plt.title("  Spiral using Polar Coordinates", fontsize=14)
plt.axis("equal")
plt.axis("off")
plt.show()
```



0.1.7 Rose Curve

The **Rose Curve** is a beautiful mathematical shape created by using **sine** functions in parametric equations. The pattern looks like a flower with multiple petals, and the number of petals is determined by the value of `k`. When `k` is an integer, the number of petals is **2k** if `k` is even, and **k** petals if `k` is odd.

Code Breakdown:

1. **Choose the Number of Petals**:
   `k = 5` determines the number of petals. You can experiment with different values for `k` (e.g., 4, 5, 8) to see how the shape changes.
2. **Define the Range of** :
   `theta = np.linspace(0, 2 * np.pi, 1000)` generates 1000 equally spaced values for the angle over one full rotation (0 to (2)).
3. **Set the Radius**:
   `r = np.sin(k * theta)` defines the radius of the curve. The `k` factor controls the number

of petals and the “frequency” of the sine wave. As `k` changes, the number of loops or petals in the rose curve changes.

4. **Calculate x and y Coordinates**:
   Using the parametric equations:

   - `x = r * np.cos(theta)`

   - `y = r * np.sin(theta)`
     we calculate the `x` and `y` positions for each value of to plot the curve.

5. **Plot the Rose Curve**:

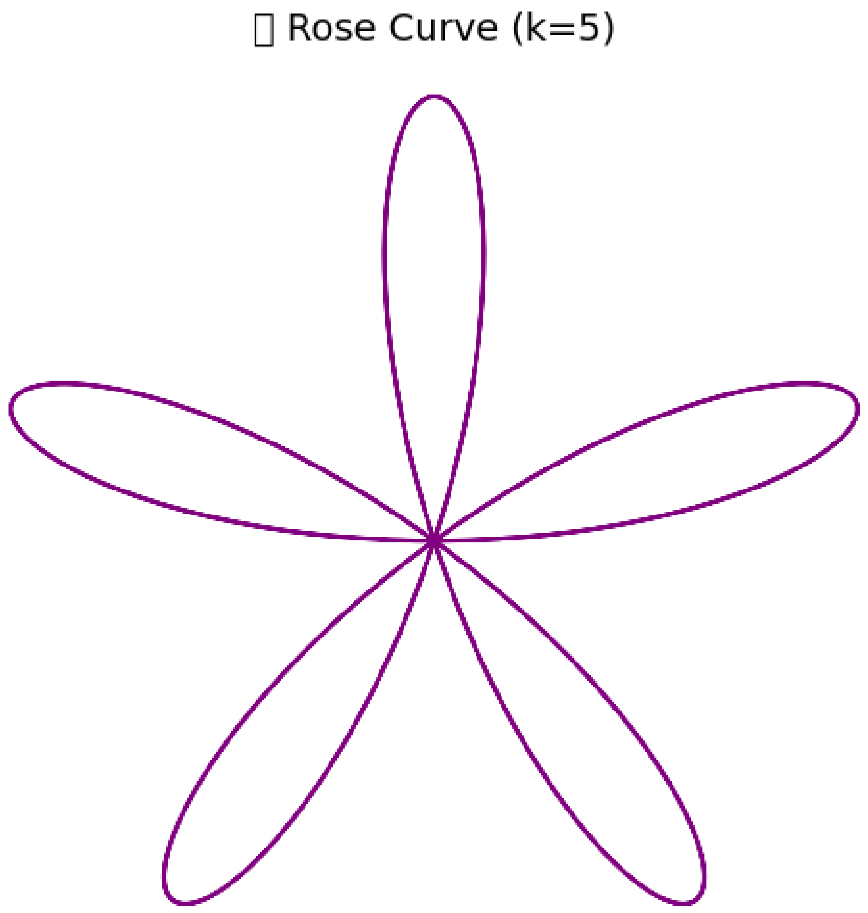
   - `plt.figure(figsize=(6, 6))` ensures the plot is square-shaped for symmetry.
   - `plt.plot(x, y, color="purple")` draws the rose curve with a purple-colored line.
   - `plt.title(f" Rose Curve (k={k})", fontsize=14)` adds a title with the current value of `k`.
   - `plt.axis("equal")` ensures that both axes are scaled equally, preserving the symmetry of the curve.
   - `plt.axis("off")` removes the axes for a cleaner, more artistic look.

[161]:
```python
k = 5  # Try different values (e.g., 4, 5, 8)

theta = np.linspace(0, 2 * np.pi, 1000)
r = np.sin(k * theta)

x = r * np.cos(theta)
y = r * np.sin(theta)

plt.figure(figsize=(6, 6))
plt.plot(x, y, color="purple")
plt.title(f" Rose Curve (k={k})", fontsize=14)
plt.axis("equal")
plt.axis("off")
plt.show()
```



0.1.8 Trig Heart – A Heart Made of Math

Who knew math could be so **romantic**? In this example, we use trigonometric functions to draw a **heart** shape. Yes, you read that right—math can help you draw love! The heart shape is created using a combination of **sine** and **cosine** functions. Let's see how we can code this love story.

Code Breakdown:

1. **Set the Range for t**:
   `t = np.linspace(0, 2 * np.pi, 1000)` creates 1000 equally spaced points from 0 to (2). This gives us the smooth curve we need to draw our heart.
2. **Calculate the x and y Coordinates**:
   - `x = 16 * np.sin(t) ** 3`: This controls the "horizontal" part of the heart, making it stretch out and curve.
   - `y = 13 * np.cos(t) - 5 * np.cos(2 * t) - 2 * np.cos(3 * t) - np.cos(4 * t)`: This fancy math is what makes the heart shape! The multiple cosine terms add a twist, shaping the curves and giving the heart its smooth, symmetric look.

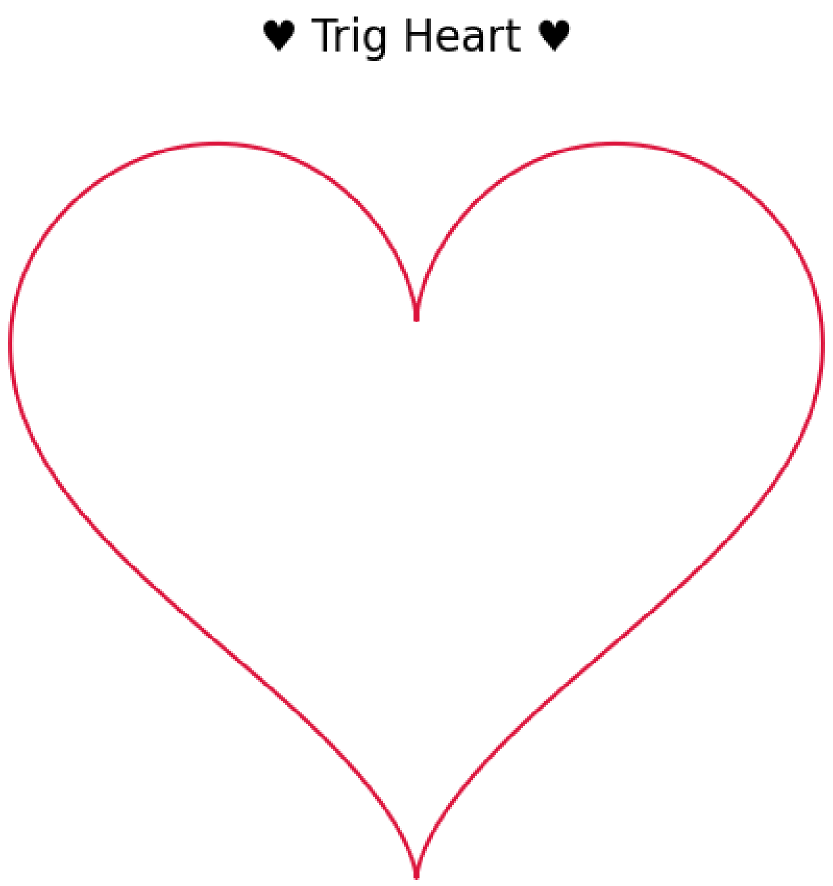
3. **Plot the Heart**:

   - `plt.figure(figsize=(6, 6))` ensures our heart looks just right, with equal scaling on both axes.
   - `plt.plot(x, y, color="crimson")` draws the heart with a deep red (because love is deep, right?).
   - `plt.title("  Trig Heart  ", fontsize=16)` adds a sweet title to the plot.
   - `plt.axis("equal")` makes sure the heart is not stretched or squished.
   - `plt.axis("off")` removes the axes to keep the focus on the heart, because it's all about the love!

[215]:
```
t = np.linspace(0, 2 * np.pi, 1000)

x = 16 * np.sin(t) ** 3
y = 13 * np.cos(t) - 5 * np.cos(2 * t) - 2 * np.cos(3 * t) - np.cos(4 * t)

plt.figure(figsize=(6, 6))
plt.plot(x, y, color="crimson")
plt.title("  Trig Heart  ", fontsize=16)
plt.axis("equal")
plt.axis("off")
plt.show()
```



0.1.9 Trigonometry in Real Life – From Shadows to Heights!

Trigonometry isn't just for the classroom! It's used in many real-world situations, from measuring **heights** to calculating **shadows** and even navigating through space. Let's explore a few cool applications of trigonometry that can be done with just a little bit of code!

1. Measuring the Height of Objects Trigonometry can be used to measure the height of objects like trees, buildings, or mountains without needing to climb them! All you need is the **distance** from the object and the **angle of elevation** to the top of the object. Using these, we can calculate the height.

- **Formula**:
  The height of the object `h` is calculated using the following formula:

$$h = d \cdot \tan(\theta)$$

Where: - `d` is the distance from the object. -   is the angle of elevation.

In the next code example, we will use this formula to calculate the height of a tree based on the distance and angle.

[230]:
```python
import math

# Distance from the tree (in meters)
d = 10  # Change this to any distance

# Angle of elevation (in degrees)
theta = 30  # You can change this to any angle

# Convert angle to radians
theta_rad = math.radians(theta)

# Calculate height
height = d * math.tan(theta_rad)

# Print the result
print(f"The height of the tree is {height:.2f} meters.")
```

```
The height of the tree is 5.77 meters.
```

0.1.10 2. Calculating the Length of a Shadow

Have you ever wondered how long a shadow will be at different times of the day? Trigonometry can help! If you know the **height** of an object and the **angle of elevation** of the sun, you can calculate the **length of the shadow** it casts.

- **Formula**:
  The shadow length `s` is calculated using the following formula:

$$s = \frac{h}{\tan(\theta)}$$

Where: - `h` is the height of the object. -   is the angle of elevation of the sun.

In the next code example, we'll calculate the length of a shadow based on the height of an object and the sun's angle.

[236]:
```python
import math

# Height of the object (in meters)
h = 5  # Change this to the height of the object

# Angle of elevation of the sun (in degrees)
theta = 45  # You can adjust this angle

# Convert angle to radians
theta_rad = math.radians(theta)
```

```
# Calculate shadow length
shadow_length = h / math.tan(theta_rad)

# Print the result
print(f"The length of the shadow is {shadow_length:.2f} meters.")
```

```
The length of the shadow is 5.00 meters.
```

0.1.11 3. Navigating with Angles

When you're out hiking, sailing, or navigating a map, trigonometry can be used to determine your position or calculate distances based on angles. For example, if you know the distance you've traveled in one direction and the angle between two paths, you can calculate the remaining distance or direction.

This is often used in **bearing navigation** where you navigate between two points using angles and distances.

- **Formula**:
  Navigation problems usually require knowledge of several trigonometric formulas, such as:
  - For distance calculation using a known angle and side:

$$d = \frac{s}{\cos(\theta)}$$

    Where:
- s is the known distance.
- is the angle between paths.

[239]:
```
import math

# Known distance (in meters)
s = 100  # You can change this to your traveled distance

# Angle between two paths (in degrees)
theta = 30  # Change this angle to see how it affects the result

# Convert angle to radians
theta_rad = math.radians(theta)

# Calculate remaining distance
d = s / math.cos(theta_rad)

# Print the result
print(f"The remaining distance is {d:.2f} meters.")
```

```
The remaining distance is 115.47 meters.
```

[ ]:
```

```