In [ ]:
```
Name: Cayden Barnett
Cource: Math141 (Calculus 1)
Section: 012
```

Lab 01 - New User's Tour

To take the tour, click in each executable cell. A code cell is any of the ones with **[#]** beside them. Type the appropriate code, and execute the cell. You execute a cell by pressing **Shift + Enter**. Note that simply pressing **Enter** will go to the next line in the executable cell and will not execute the code.

This lab uses Python with the libraries NumPy, SymPy, and Matplotlib to perform calculations and give insight into the many topics covered in Calculus. These libraries provide powerful tools for mathematical computation, symbolic mathematics, and visualization.

We'll first import the necessary libraries:

In [4]:
```
# Import necessary libraries
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt

# Define x as a symbolic variable
x = sp.Symbol('x')
```

Python as a Calculator

Python's most basic function is to act as a conventional calculator. For example, we can use Python to simplify the expression

$$\frac{3(2+\frac{1}{2})^2}{\frac{1}{5}-\frac{2}{3}}$$

We can simply input the expression into an executable cell exactly as we would a calculator. One thing to be careful about in Python is you must use `*` to represent multiplication and `**` for exponentiation (not `^` which is used for bitwise XOR in Python).

In [5]:
```
(3*(2+1/2)**2)/(1/5 - 2/3)
```

Out[5]:
```
-40.17857142857143
```

Python will compute this using floating-point arithmetic. To get exact results with fractions, we can use SymPy's `sp.Rational()` function:

In [6]:
```
result = (3*(2+sp.Rational(1,2))**2)/(sp.Rational(1,5) - sp.Rational(2,3))
display(result)
```

$$-\frac{1125}{28}$$

Since the expression only contains rational numbers, SymPy gives the output as a rational number. If we wanted a decimal approximation of the output, we could use the `float()` function or the `sp.N()` function in SymPy:

In [7]:
```
float(result)  # Convert to decimal
```

Out[7]:
```
-40.17857142857143
```

In [8]:
```
sp.N(result, 10)  # Decimal with 10 digits of precision
```

Out[8]: -40.17857143

A second way in which we could get a decimal approximation for a rational expression is to make at least one of the numbers a floating-point number. This means, make one of the numbers a decimal.

In [9]:
```
(3.0*(2+1/2)**2)/(1/5 - 2/3)
```

Out[9]:
```
-40.17857142857143
```

Predefined Functions and Constants

Python with SymPy has many of the common math functions and constants predefined. Most of these can be referenced using the `sp.function_name()` syntax. Note how SymPy has no issue calculating $\sqrt{100}$, $\sqrt[3]{54}$, $\cos(\pi)$ and $\arcsin(1/2)$.

In [10]:
```
sp.sqrt(100)
```

Out[10]: 10

In [11]:
```
(54)**(sp.Rational(1,3))
```

Out[11]: $3\sqrt[3]{2}$

In [12]:
```
sp.cos(sp.pi)
```

Out[12]: -1

In [13]:
```
sp.asin(sp.Rational(1,2))
```

Out[13]: $\frac{\pi}{6}$

To add more cells to your notebook in Visual Studio Code:

1. Click the **+ Code** button at the top of the notebook to insert a new code cell below the currently selected cell.
2. You can also right-click on any cell and choose **Insert Cell Above** or **Insert Cell Below** from the context menu.

Go ahead and add two new code cells. Use these cells to approximate the values of $\sqrt[3]{54}$ and $\arcsin(1/2)$ to 5 digits of precision.

In [43]:
```
cb54 = (54)**(sp.Rational(1/3))

result2 = sp.asin(1/2)

display(sp.N(cb54, 5))
display(sp.N(result2, 5))
```

3.7798

0.5236

You can also add a Markdown cell to your notebook in Visual Studio Code:

1. Click the **+ Markdown** button at the top of the notebook to insert a new Markdown cell below the currently selected cell.
2. Markdown cells allow you to write formatted text, including Markdown, HTML, and LaTeX for math expressions.
3. To add a Markdown cell at the very beginning of the notebook, use the **Insert Cell Above** option by right-clicking the first cell.

Insert a Markdown cell at the beginning of the notebook. In the Markdown cell, type your name, course, and section number. This must be included in every lab submission.

Assigning Expressions to Names

Python allows us to assign an expression to a name so that it can easily be referenced throughout the entire worksheet. For example, we can assign $\pi/2$ to the variable a by doing the following.

In [14]:
```
a = sp.pi/2
```

Note that Python suppresses the output whenever you are making an assignment. We

can check that a really has been assigned to $\pi/2$ by entering it into an input cell by itself.

In [15]:
```
a
```

Out[15]: $\frac{\pi}{2}$

Now, whenever we use a throughout the entire worksheet, it will be referring to $\pi/2$. If we wanted to reassign a to something else, all we have to do is set it equal to the new value.

In [16]:
```
a = sp.pi/3
a
```

Out[16]: $\frac{\pi}{3}$

Note that this time, Python did display the new value of a. This is because we both assigned a to its new value and told Python to return a in the same input cell. Python allows multiple lines of code in the same cell and will execute all lines, however, it will only display the last line of code.

In [17]:
```
a
a + 1
a + 2
```

Out[17]: $\frac{\pi}{3} + 2$

If you wish to display multiple lines, you can use the `display()` command.

In [18]:
```
display(a)
display(a + 1)
display(a + 2)
```

$\frac{\pi}{3}$

$1 + \frac{\pi}{3}$

$\frac{\pi}{3} + 2$

Another way is to you use the Python `print()` command. Note that the `display()` command gives a better, visually appealing output.

In [19]:
```
print(a)
print(a + 1)
print(a + 2)
```

```
pi/3
1 + pi/3
pi/3 + 2
```

You can even use **f-strings** to output explanatory text (in string format) when printing your variables. All that is required is placing an `f` in front of the string you'd like to print and using `{}` to insert variables.

In [20]:
```
solution = sp.cos(a)
print(f'The cos({a}) evaluates to {solution}.')
```

```
The cos(pi/3) evaluates to 1/2.
```

Caution: Be careful when naming your variables in Python. If you assign a value to a name that is already used by a function or object from a library (for example, `sp.sin` from SymPy), you will overwrite that function without warning. For instance, if you write `sp.sin = 5`, then `sp.sin` will refer to the number 5 instead of the SymPy sine function. To restore the original function you must restart the kernel, remove the erroneous assignment and re-execute the cells in the notebook.

Creating Functions

We can also create functions in Python. For example, in order to create the function $f(x) = x^2$, we do the following.

In [21]:
```
def f(x):
    return x**2  # Note: use ** for exponentiation in Python, not ^
```

Similar to assignments, Python will not display any output when creating a function. We can call the function after creating it in order to verify that we've created the intended function.

In [22]:
```
def f(x):
    return x**2

f(x)
```

Out[22]: x^2

Now that we have $f(x)$ defined, we can evaluate the function at different inputs.

In [23]:
```
f(4)
```

Out[23]: 16

In [24]:
```
f(x) + 4
```

Out[24]: $x^2 + 4$

In [25]:
```
f(f(x))
```

Out[25]: x^4

What if we wanted to use the variable t in place of x? Python with SymPy requires us to define symbols before using them. We defined x at the beginning of this notebook, and now we'll define t as well.

In [26]:
```
t = sp.Symbol('t')
f(t)
```

Out[26]: t^2

Rather than just doing normal calculations, we can also have SymPy perform calculus.

In [27]:
```
h = sp.Symbol('h')
sp.limit((f(x+h) - f(x))/h, h, 0) # Limit definition of the derivative
```

Out[27]: $2x$

In [28]:
```
sp.diff(f(x), x) # First derivative
```

Out[28]: $2x$

In [29]:
```
sp.diff(f(x), x, 2)  # Second derivative
```

Out[29]: 2

In [30]:
```
sp.integrate(f(x), x) # Indefinite integral of f(x) (Note: Constant of integ
```

Out[30]: $\frac{x^3}{3}$

Plotting

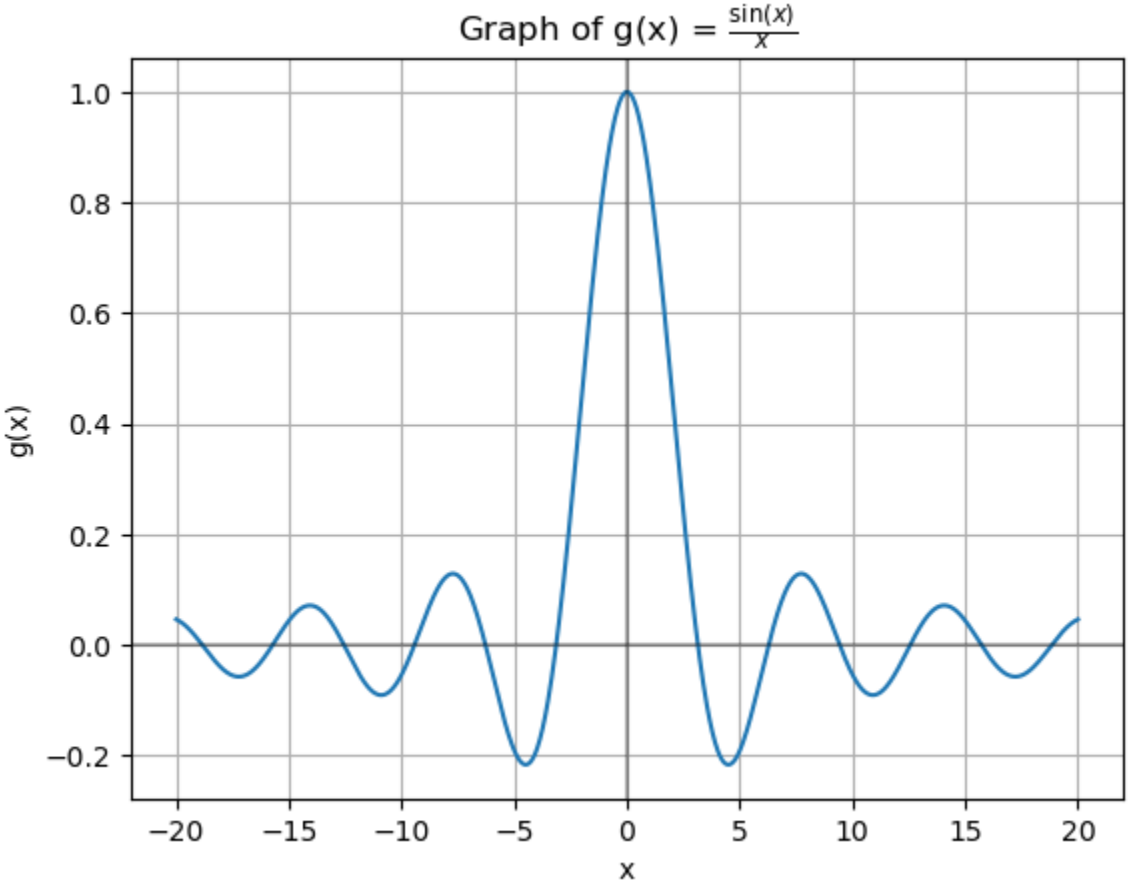
The last part of Python which we will explore in this tour is its ability to plot graphs of functions using Matplotlib. First, use the cell below to create the function $g(x) = \sin(x)/x$.

In [31]:
```
# Define the function g(x)
def g(x):
    return sp.sin(x)/x
```

Now, plot $g(x)$ using Matplotlib. We need to convert our symbolic function (SymPy) to a numerical one (NumPy) for plotting.

In [32]:
```
# Convert to a NumPy function
g_np = sp.lambdify(x, g(x), 'numpy')
```

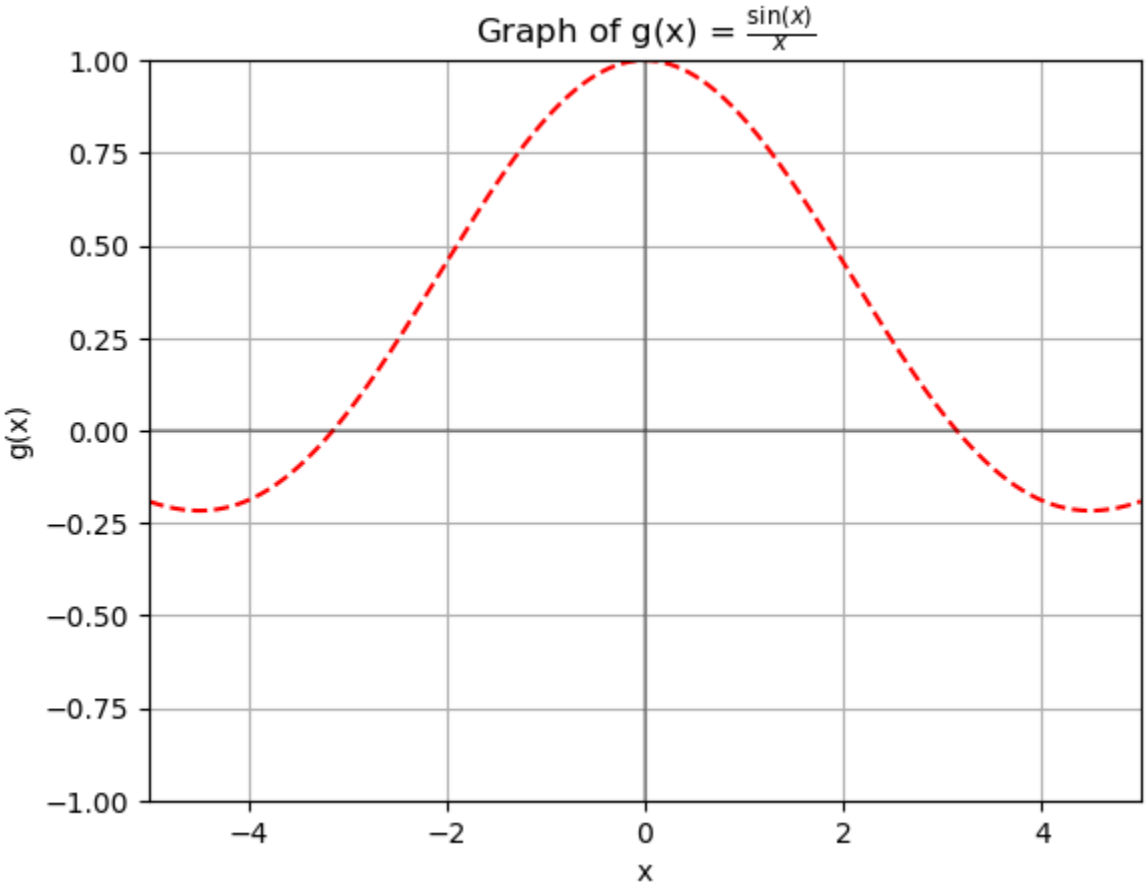
```
# Create x values for plotting
x_vals = np.linspace(-20, 20, 1000)

# Plot g(x) = sin(x)/x
plt.plot(x_vals, g_np(x_vals))

# Add grid and labels
plt.grid(True)
plt.xlabel('x')
plt.ylabel('g(x)')
plt.title('Graph of g(x) = $\\frac{\\sin(x)}{x}$')

# Emphasize x-axis and y-axis
plt.axhline(y=0, color='k', alpha=0.3)
plt.axvline(x=0, color='k', alpha=0.3)

# Show the plot
plt.show()
```



Matplotlib offers many options to customize your plots. Let's modify our previous plot to show the function in a window that goes from $[-5, 5]$ on the x-axis and $[-1, 1]$ on the y-axis, changes the graph color to red, and changes the linestyle to dashed.

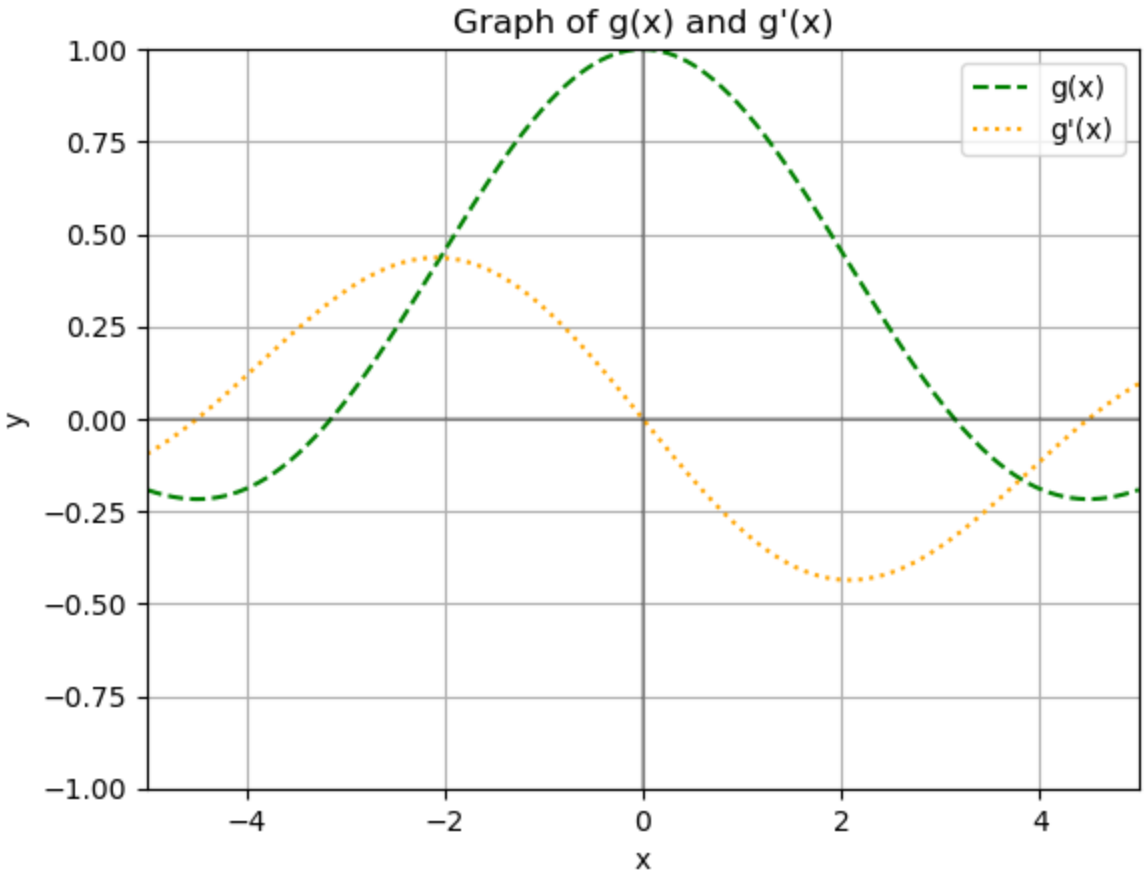
In [33]:
```
# Create x values for plotting
x_vals = np.linspace(-5, 5, 1000)
```

```
# Plot g(x) = sin(x)/x
plt.plot(x_vals, g_np(x_vals), color='red', linestyle='dashed')

# Set the viewing window
plt.xlim(-5, 5)
plt.ylim(-1, 1)

# Add grid and labels
plt.grid(True)
plt.xlabel('x')
plt.ylabel('g(x)')
plt.title('Graph of g(x) = $\\frac{\\sin(x)}{x}$')

# Emphasize x-axis and y-axis
plt.axhline(y=0, color='k', alpha=0.3)
plt.axvline(x=0, color='k', alpha=0.3)

# Show the plot
plt.show()
```



Matplotlib also has the ability to graph multiple functions at once. When plotting multiple functions at the same time, we use multiple `plt.plot()` calls or a single call with multiple arrays. The following code plots both $g(x)$ and its derivative $g'(x)$ from $[-5, 5]$ on the x-axis and $[-1, 1]$ on the y-axis. It also plots $g(x)$ in green with dashes and $g'(x)$ in orange with dots. Additionally, we add a legend to distinguish between the two functions in the graph.

In [34]:
```
# Calculate the derivative of g(x) symbolically
def g_prime(x):
    return sp.diff(g(x), x)

# Convert to a NumPy function
g_prime_np = sp.lambdify(x, g_prime(x), 'numpy')

# Create x values for plotting
x_vals = np.linspace(-5, 5, 1000)

# Plot g(x) and g'(x)
plt.plot(x_vals, g_np(x_vals), color='green', linestyle='dashed', label='g(x
plt.plot(x_vals, g_prime_np(x_vals), color='orange', linestyle='dotted', lab

# Set the viewing window
plt.xlim(-5, 5)
plt.ylim(-1, 1)

# Add grid and labels
plt.grid(True)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Graph of g(x) and g\'(x)')

# Emphasize x-axis and y-axis
plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
plt.axvline(x=0, color='k', linestyle='-', alpha=0.3)

# Add legend
plt.legend(loc='upper right')

# Show the plot
plt.show()
```



Converting Notebook to PDF for Submission

When you finish your lab, submit a PDF version of your notebook to Blackboard for grading. In Visual Studio Code:

1. Click the **Export** button at the top right of the notebook editor and select **Export As PDF**.
2. If you do not see the PDF option, choose **Export As HTML** and then open the HTML file in your browser. Use your browser's **File → Print** menu and select **Save as PDF**.

Upload the resulting PDF to Blackboard.

Note: You do not need to install LaTeX to export your notebook as a PDF. Exporting as HTML and printing to PDF works on any system.