Chem 277B Spring 2024 Tutorial 0

Outline

- Introduction & Discussion Format
- Installing Python through Anaconda and IDE setups
- Introduction to Jupyter Notebook
- Introduction to Python basics
- Introduction to Numpy & Matplotlib
- Introduction to Clean Coding

---

1. Discussion Formats

- Time: Mon 5-7pm PST
- Format: Teresa's OH (30-60 mins) then tutorial walkthrough
- Attendance: Mandatory via zoom poll + submission of attempted tutorial notebooks (Deadline Wednesday 11:59PM)
- Email: honamnguyen@berkeley.edu (Nam); please include **Chem 277B** in the subject line

---

2. Installing Python through Anaconda

2.1 Why Anaconda?

- Anaconda simplifies package management and deployment.
- It's especially useful for scientific computing, ensuring you have all the necessary libraries.

2.2 Installation Guide

- Anaconda Installation Guide
- A useful GitHub page

2.3 Setting up your environment

Anaconda manages all packages (even Python) with "environment". Environments are seperated, which means different environments can contain different Python, different packages, if properley set up. This will minimize user's efforts to manage the dependencies between packages.

Once conda is installed, an environment named `base` is automatically created. You can see the name of the current environment in your terminal (MacOS or Linux) or Anaconda prompt (Windows). For example:

```
(base) [honamnguyen@Nams-Air ~]
```

Terminal app in MacOS:

Anaconda prompt in Windows:

For this class, it is recommended to create a new environment with the following procedure (typing the corresponding command in terminal/prompt) and use this environment to finish all the tutorial/homework assignments.

- Create a new enviornment:

  `conda create -n [env_name] python=[version]`

  Here, you can specify `env_name` to `chem277b` and `version` to `3.10`. e.g.

  `conda create -n chem277b python=3.10`

- Activate environment

  `conda activate chem277b`

- Install packages

  `pip install numpy scipy scikit-learn matplotlib pandas seaborn ipykernel jupyterlab`

  (you can also use conda install but I find it's not always up-to-date and there are lot of channels you have to specify)

  - **NumPy**: Fundamental package for scientific computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on them.
  - **Pandas**: Package for data manipulation and analysis in Python

  - **Scipy**: Package for scientific computing based on NumPy, providing high-level functions to perform mathematical operations, such as optimization, integration, eigenvalue problems.
  - **Matplotlib**: Package for data visualization.
  - **Seaborn**: Package based on Matplotlib, offering a high-level interface for drawing attractive and informative statistical graphics.
  - **Scikit-learn**: Package for machine learning
  - **ipykernel**: Package that provides the IPython kernel for Jupyter, allowing Jupyter notebooks to interact with Python code.
  - **jupyterlab**: similar to jupyter notebook but you can open multiple notebooks at once and more
- Add your environment to Jupyter notebook

  `python -m ipykernel install --user --name=chem277b`
- Deactivate your environment after done working

  `conda deactivate`

2.4 Summary of command lines to set up

`conda create -n chem277b python=3.10`

`conda activate chem277b`

`pip install numpy scipy scikit-learn matplotlib pandas seaborn ipykernel jupyterlab`

`python -m ipykernel install --user --name=chem277b`

---

3. Setting up and Using Jupyter Notebook

Jupyter Notebook is an open-source web application that allows you to create and share documents containing live code, equations, visualizations, and narrative text. It's particularly popular among data scientists and researchers for its interactive nature.

3.1. Why Jupyter Notebook?

- **Interactive Computing**: Jupyter allows for real-time code execution, making it easier to test and debug code.
- **Rich Display System**: You can embed images, videos, LaTeX, and more.
- **Integration with Big Data Tools**: Jupyter can be integrated with big data tools like Apache Spark.
- **Extensibility**: Jupyter can be extended with various plugins and supports more than 40 programming languages.

3.2. Launching Jupyter Notebook

- **Through Anaconda Navigator**:
  - Open Anaconda Navigator.
  - Click on the "Jupyter Notebook" icon.

- **Using the Command Line**:
  - Open your terminal or command prompt.
  - Make sure you are at the `base` environment. If not, use the following command: `conda deactivate` or `conda activate base`
  - Navigate to your desired directory using `cd your_directory_name`.
  - Type `jupyter lab` or `jupyter notebook` and press Enter. For Linux/MacOS, the default web browser should pop up automatically. For Windows, you may need to manually go to http://localhost:8888/

3.3. Working with Cells

- **Types of Cells**:
  - **Code Cells**: Where you write and execute your Python code.
  - **Markdown Cells**: For writing text, explanations, or embedding images and videos. This cell, for instance, is a markdown cell! Here is a quick reference for basic Markdown syntax: https://www.markdownguide.org/cheat-sheet/
- **Basic Operations**:
  - **Creating a Cell**: Click on the "+" button in the toolbar or press `B` (for below) or `A` (for above) when a cell is selected.
  - **Executing a Cell**: Click on the "Run" button in the toolbar or press `Shift + Enter`.
  - **Changing Cell Type**: Use the dropdown in the toolbar or press `M` for markdown and `Y` for code.
  - **Deleting a Cell**: Press `D` twice.

3.4. Saving and Exporting

- **Saving**: Click on the floppy disk icon in the toolbar or press `Ctrl + S`.
- **Exporting**: Go to `File > Download as` to export your notebook in various formats like PDF, HTML, or Python (.py).

3.5. Additional Resources

- Jupyter Notebook Beginner Guide

For this class, we ask you to download your work as HTML, and then convert the HTML to PDF for homework submission because direct PDF download could cause problems. (see guide)

---

4. Python Basics

This section is adapted from the `CS231n` Python tutorial by Justin Johnson (http://cs231n.github.io/python-numpy-tutorial/).

Python is a great general-purpose programming language on its own, but with the help of a few popular libraries (numpy, scipy, matplotlib) it becomes a powerful environment for scientific computing.

Some of you may have already worked with Python and the related packages before; for the rest of you, this section will serve as a quick crash course both on the Python programming language and on the use of Python for scientific computing.

If you have previous knowledge in Matlab, the "numpy for Matlab users" page is suggested for reading.

In this tutorial, we will cover:

- Basic Python: Basic data types (Containers, Lists, Dictionaries, Sets, Tuples), Functions, Classes
- Numpy: Arrays, Array indexing, Datatypes, Array math, Broadcasting
- Matplotlib: Plotting, Subplots, Setting labels and legends

4.1. Data Types

Python has several built-in data types. Here are some of the most commonly used ones:

Integers

Whole numbers, e.g., `-3, -2, -1, 0, 1, 2, 3...`

In [1]:
```
x = 5
print(type(x))
```

```
<class 'int'>
```

Floats

Decimal numbers, e.g., `-3.5, -2.25, 0.0, 1.1, 2.2...`

In [2]:
```
y = 3.14
print(type(y))
```

```
<class 'float'>
```

In [3]:
```
print(x + 1)   # Addition;
print(x - 1)   # Subtraction;
print(x * 2)   # Multiplication;
print(x / 2)   # Division;
print(x // 2)  # Division, take only integer;
print(x ** 2)  # Exponentiation;
```

```
6
4
10
2.5
2
25
```

In [4]:
```
x += 1  # Equivalent to x = x + 1
print(x)
x *= 2  # Equivalent to X = x * 2
print(x)
```

```
6
12
```

Booleans

Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (`&&`, `||`, etc.):

In [5]:
```
t, f = True, False
print(type(t))
```

```
<class 'bool'>
```

In [6]:
```
print(t and f) # Logical AND;
print(t or f)  # Logical OR;
print(not t)   # Logical NOT;
print(t != f)  # Logical XOR;
```

```
False
True
False
True
```

Strings

Sequence of characters, e.g., `"Hello, World!"`

In [7]:
```
greeting = 'Welcome to'       # String literals can use single quotes
class_code = "CHEM277B"    # or double quotes; it does not matter.
print(greeting, class_code)

# string concatenation
welcome = greeting + ' ' + class_code  # use "+" operator for string concatena
print(welcome)
```

```
Welcome to CHEM277B
Welcome to CHEM277B
```

Format strings

In [8]:
```
welcome2 = '%s %s %d/%d' % (greeting, 'CHEM', 147,247)  # Format string style
print(welcome2)

welcome3 = "{} {} {}/{}".format(greeting, 'CHEM', 147, 247) # use .format func
print(welcome3)
```

```
Welcome to CHEM 147/247
Welcome to CHEM 147/247
```

f-string is a preferred way to format strings in Python.

In [9]:
```
code1 = 277
code2 = '278'

# f-string begins with 'f' and variables to format in curly braces
welcome_fstr = f"{greeting} CHEM{code1}/{code2}"
print(welcome_fstr)
```

```
Welcome to CHEM277/278
```

4.2. Containers

4.2.1 Lists

Lists are ordered collections of items (which can be of any type). They are mutable, meaning their contents can change.

Creating a List

In [10]:
```
# create list with brakets
elements = ["hydrogen", "carbon", "oxygen", "nitrogen"]
print(elements)
print("Length:", elements)
```

```
['hydrogen', 'carbon', 'oxygen', 'nitrogen']
Length: ['hydrogen', 'carbon', 'oxygen', 'nitrogen']
```

In [11]:
```
# create list with list() function
numbers = list(range(5))
print(numbers)
chars = list("abcde")
print(chars)
```

```
[0, 1, 2, 3, 4]
['a', 'b', 'c', 'd', 'e']
```

Accessing Elements

In [12]:
```
first_element = elements[0]  # Indexing starts from 0
print(first_element)

second_element = elements[1]
print(second_element)

last_element = elements[-1] # Also indexing from backward
print(last_element)

second_last_element = elements[-2]
print(second_last_element)
```

```
hydrogen
carbon
nitrogen
oxygen
```

Adding Elements

In [13]:
```
elements.append("sulfur")
print(elements)
```

```
['hydrogen', 'carbon', 'oxygen', 'nitrogen', 'sulfur']
```

Popping Elements

In [14]:
```
last_element = elements.pop()     # Remove and return the last element of the
print(elements)
print(last_element)
```

```
['hydrogen', 'carbon', 'oxygen', 'nitrogen']
sulfur
```

Concaternation

In [15]:
```
other_elements = ['sodium', 'zinc']
elements = elements + other_elements
print(elements)
```

```
['hydrogen', 'carbon', 'oxygen', 'nitrogen', 'sodium', 'zinc']
```

Slicing

In [16]:
```
print(elements)          # Now printing the list
print(elements[2:4])     # Get a slice from index 2 to 4 (exclusive)
print(elements[2:])      # Get a slice from index 2 to the end
print(elements[:2])      # Get a slice from the start to index 2 (exclusive)
print(elements[:])       # Get a slice of the whole list
print(elements[:-1])     # Slice indices can be negative
print(elements[::2])     # Slice with jumping index
elements[2:4] = ['fluorine', 'chlorine'] # Assign a new sublist to a slice
print(elements)
```

```
['hydrogen', 'carbon', 'oxygen', 'nitrogen', 'sodium', 'zinc']
['oxygen', 'nitrogen']
['oxygen', 'nitrogen', 'sodium', 'zinc']
['hydrogen', 'carbon']
['hydrogen', 'carbon', 'oxygen', 'nitrogen', 'sodium', 'zinc']
['hydrogen', 'carbon', 'oxygen', 'nitrogen', 'sodium']
['hydrogen', 'oxygen', 'sodium']
['hydrogen', 'carbon', 'fluorine', 'chlorine', 'sodium', 'zinc']
```

4.2.2 Tuples

Tuples are **ordered non-mutable** collections of objects, which means they cannot be modified once created.

In [17]:
```
# Create a tuple with parentheses or bulit-in tuple method
tup = (1, 2, 3)
tup2 = tuple([1, 2, 3])

print(tup[0])     # indexing
print(len(tup))   # get the length
```

```
1
3
```

4.2.3 Dictionaries

Dictionaries store (key, value) pairs, like bookmark and content of the book. You can use them like this:

In [18]:
```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['cat'])       # Get an entry from a dictionary
print('cat' in d)     # Check if a dictionary has a given key
```

```
cute
True
```

In [19]:
```
d['fish'] = 'wet'    # Set an entry in a dictionary
print(d['fish'])
```

```
wet
```

In [20]:
```
# print(d['monkey'])  # KeyError: 'monkey' not a key of d
```

In [21]:
```
print(d.get('monkey', 'N/A'))  # Get an element with a default
print(d.get('fish', 'N/A'))    # Get an element with a default
```

```
N/A
wet
```

In [22]:
```
del(d['fish'])        # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key
```

```
N/A
```

In [23]:
```
print(d)
d.update({"cat": "hungry", "monkey": "naughty"}) # update a dictionary
print("Updated:", d)
```

```
{'cat': 'cute', 'dog': 'furry'}
Updated: {'cat': 'hungry', 'dog': 'furry', 'monkey': 'naughty'}
```

4.3. Conditional Statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

In [24]:
```
a = 33
b = 200

if b > a:
    print("b is greater than a")
```

```
b is greater than a
```

In [25]:
```
grade = 85

if grade > 90:
    print("A+")
elif grade > 80:
    print("A")
else:
    print("B")
```

```
A
```

4.5. Loops

Python has two primitive loop commands:

For loops

In [26]:
```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
```

```
    print(fruit)
```

```
apple
banana
cherry
```

While loops

In [27]:
```
i = 1
while i < 6:
    print(i)
    i += 1
```

```
1
2
3
4
5
```

Comprehensions

Comprehension is a "pythonic" syntax, making it easier to create containers (list, dictionary)

In [28]:
```
# list
nums = [i for i in range(5)]
print(nums)
nums2 = [i * 2 for i in range(5)]
print(nums2)


# dictionary
numdict = {i: i + 1 for i in range(5)}
print(numdict)
```

```
[0, 1, 2, 3, 4]
[0, 2, 4, 6, 8]
{0: 1, 1: 2, 2: 3, 3: 4, 4: 5}
```

4.5 Function

Functions are blocks of organized and reusable code. They provide better modularity and a higher degree of code reusability.

Defining a Function

In [29]:
```
# function with one required argument
def greet(name):
    return f"Hello, {name}!"
```

Calling a Function

In [30]:
```
message = greet("Alice")
print(message)
```

```
Hello, Alice!
```

Default Arguments

In [31]:
```
# function with two required arguments, one of them has default value
def add(a, b=1):
    return a + b

print(add(3))
print(add(3, 4))
print(add(3, b=5))
```

```
4
7
8
```

In [32]:
```
# # non-default argument must follow default argument
# def add(b=1, a):
#     return a + b
```

Argument Lists

In [33]:
```
# arguments wrapped into a tuple if specified with "*"
def arglist(name, *args):
    print(f"Hello, {name}")
    print("Number of arguments:", len(args))
    print("Arguments passed in:")
    for i, arg in enumerate(args):
        print(f"#{i} {arg}")

arglist("Alice")
print('======')
arglist("Eric", "Chemistry", 23, [1, 2])
```

```
Hello, Alice
Number of arguments: 0
Arguments passed in:
======
Hello, Eric
Number of arguments: 3
Arguments passed in:
#0 Chemistry
#1 23
#2 [1, 2]
```

Keyword Arguments

In [34]:
```
# arguments wrapped into a dict if specified with "**"
def argkw(name, **kwargs):
    print(f"Hello, {name}!")
    print("Keyword arguments:")
    for key, value in kwargs.items():
        print(f"{key}={value}")

argkw("Alice")
print("======")
argkw("Eric", age=23, major='Chemistry')
```

```
Hello, Alice!
Keyword arguments:
======
Hello, Eric!
Keyword arguments:
age=23
major=Chemistry
```

4.6 Class

In [35]:
```python
class Greeter:

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)


g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method
g.greet(loud=True)   # Call an instance method
```

```
Hello, Fred
HELLO, FRED!
```

4.7 Additional Resources

- Python Official Documentation
  - Basics
  - Numeric Types
  - String
  - List
  - Dictionary
- W3Schools Python Tutorial

TODO: Can you generate the first 50 elements of the fibonacci sequence?

Extra: Can you write it using both `for` loop and `while` loop?

In [36]:
```
# code your solution here!
prev = 0
curr = 1
print(prev)
print(curr)
for i in range(0, 48):
    new = curr + prev
    print(new)
    prev = curr
    curr = new
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
39088169
63245986
102334155
165580141
267914296
433494437
701408733
1134903170
1836311903
2971215073
4807526976
7778742049
```

In [37]:

```python
# code your solution here!
i = 2
prev = 0
curr = 1
print(prev)
print(curr)
while i < 50:
    new = curr + prev
    print(new)
```

```
    prev = curr
    curr = new
    i = i + 1
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
39088169
63245986
102334155
165580141
267914296
433494437
701408733
1134903170
1836311903
2971215073
4807526976
7778742049
```

---

5. Introduction to NumPy and Matplotlib

In this section, we'll introduce two fundamental libraries for scientific computing and data visualization in Python: `numpy` and `matplotlib`.

5.1. NumPy

NumPy (Numerical Python) is the foundational package for numerical computations in Python. It provides support for large multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

Basic Array Operations

In [38]:
```
import numpy as np

# Creating an array
arr = np.array([1, 2, 3, 4, 5])
print("Array:", arr)

# Basic operations
print("Sum:", np.sum(arr))
print("Mean:", np.mean(arr))
print("Standard Deviation:", np.std(arr))
```

```
Array: [1 2 3 4 5]
Sum: 15
Mean: 3.0
Standard Deviation: 1.4142135623730951
```

Generating Random Data

In [39]:
```
# Generating 10 random numbers between 0 and 1
random_data = np.random.rand(10)
print("Random Data:", random_data)
```

```
Random Data: [0.46359854 0.40734672 0.83967921 0.77615737 0.32521174 0.5219068
8
 0.11546864 0.86364655 0.85667774 0.13157379]
```

5.2. Matplotlib

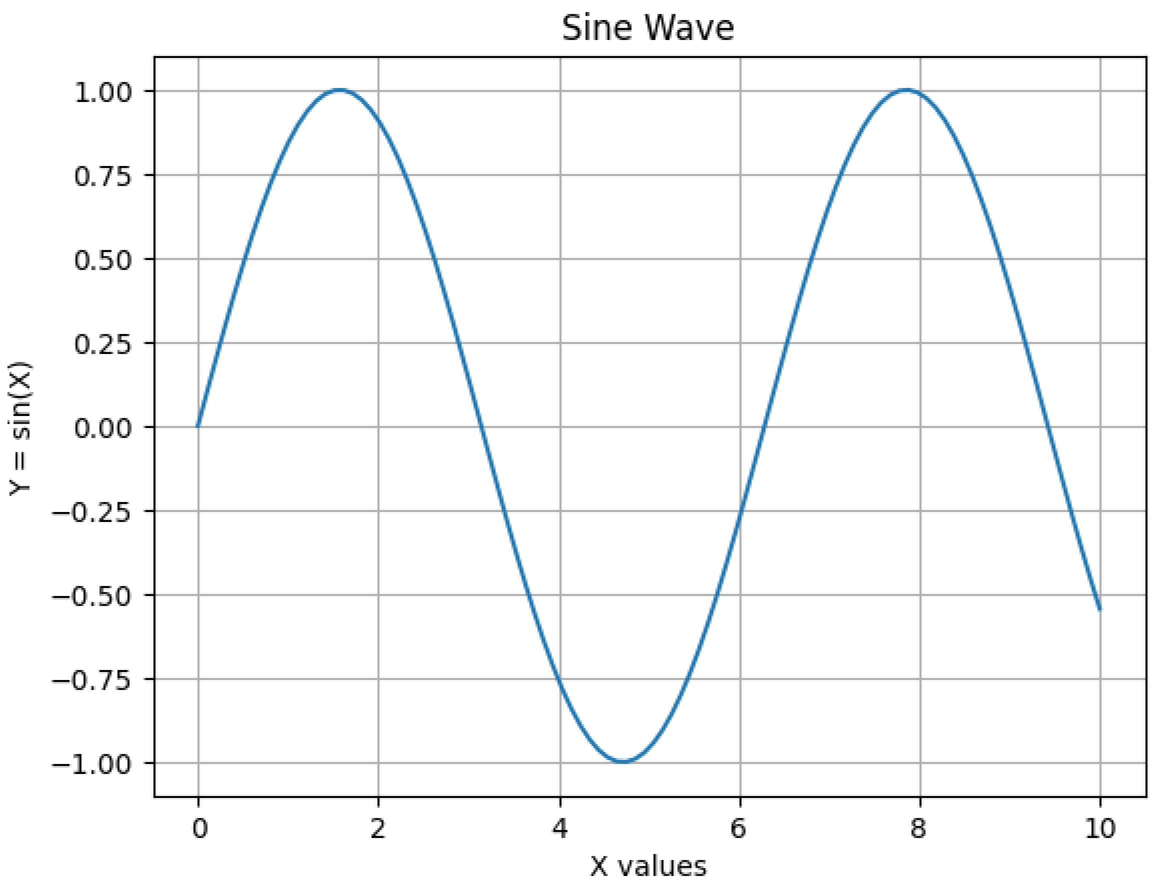
Matplotlib is a plotting library for Python. It provides an object-oriented API for embedding plots into applications that use general-purpose GUI toolkits, such as Tkinter, wxPython, Qt, or GTK. In this tutorial, we'll focus on its basic plotting capabilities.

Basic Plotting with Matplotlib

In [40]:
```
import matplotlib.pyplot as plt

# Generating data
x = np.linspace(0, 10, 100)  # 100 points from 0 to 10
y = np.sin(x)  # sine of each point
```

```
# Creating the plot
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("X values")
plt.ylabel("Y = sin(X)")
plt.grid(True)
plt.show()
```

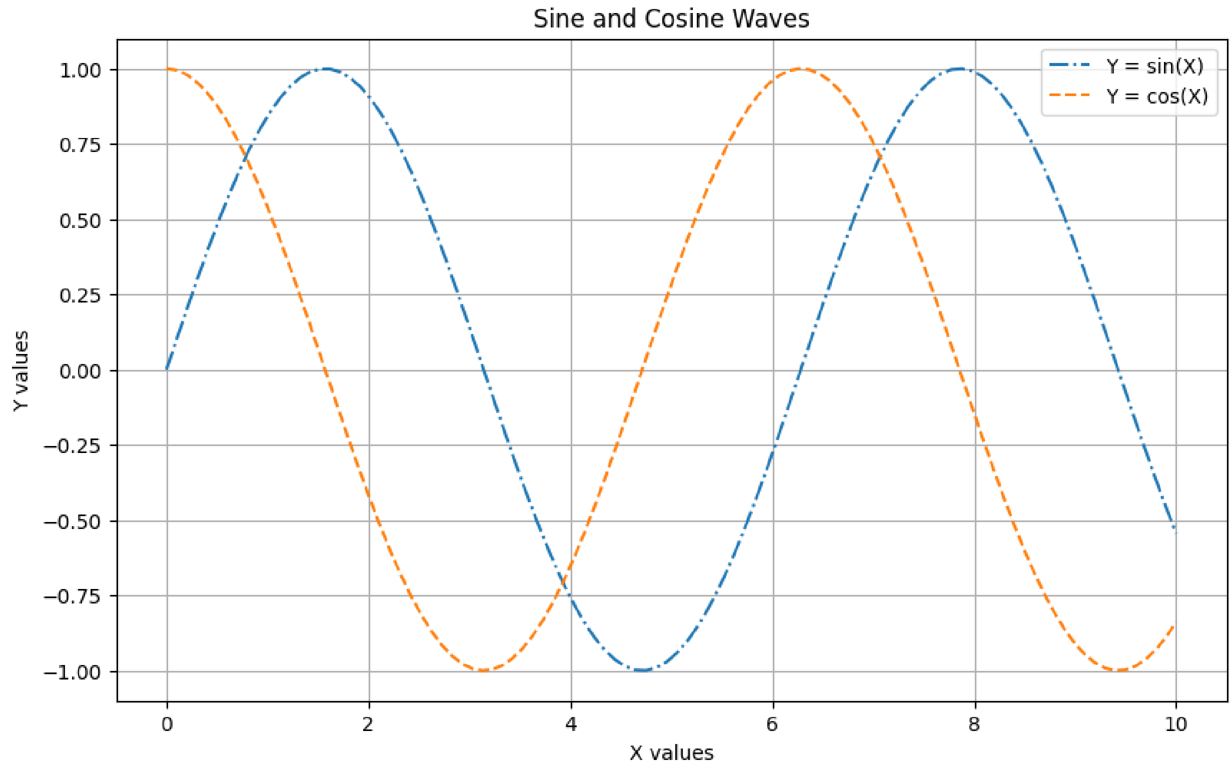


5.3. Combining NumPy and Matplotlib

Let's demonstrate the power of these libraries by combining them in a simple example.

In [41]:
```
# Generating data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Creating the plot
plt.figure(figsize=(10,6))
plt.plot(x, y1, '-.', label="Y = sin(X)")
plt.plot(x, y2, '--', label="Y = cos(X)")
plt.title("Sine and Cosine Waves")
plt.xlabel("X values")
plt.ylabel("Y values")
plt.legend()
plt.grid(True)
plt.show()
```



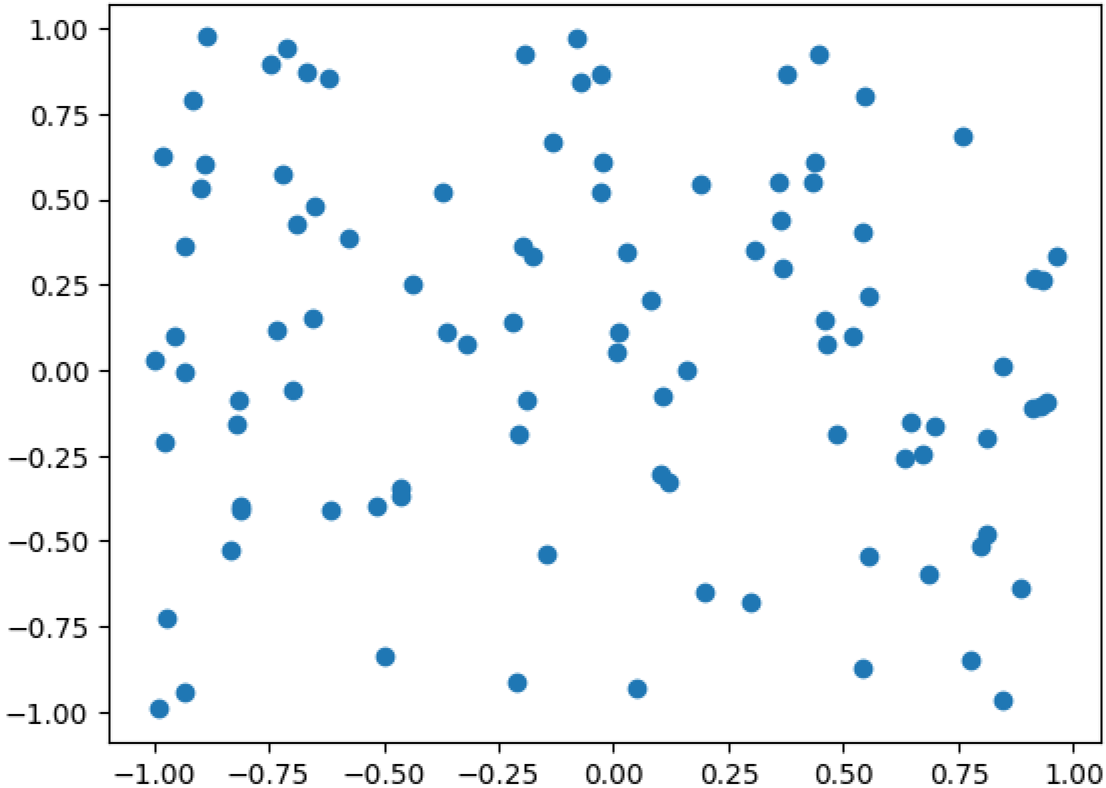
TODO: Can you draw a scatter plot of 100 random points on the x-y plane?

Additional Constraint: x and y values should range from -1 to 1.

Hint: Consider scaling the result from `np.random.rand()` and then use `plt.scatter()`

In [42]:
```
# solution goes here!
xs = np.random.rand(100) * 2 - 1
ys = np.random.rand(100) * 2 - 1
plt.scatter(xs, ys)
```

Out[42]:
```
<matplotlib.collections.PathCollection at 0x1130e9810>
```



Subplots

In [43]:

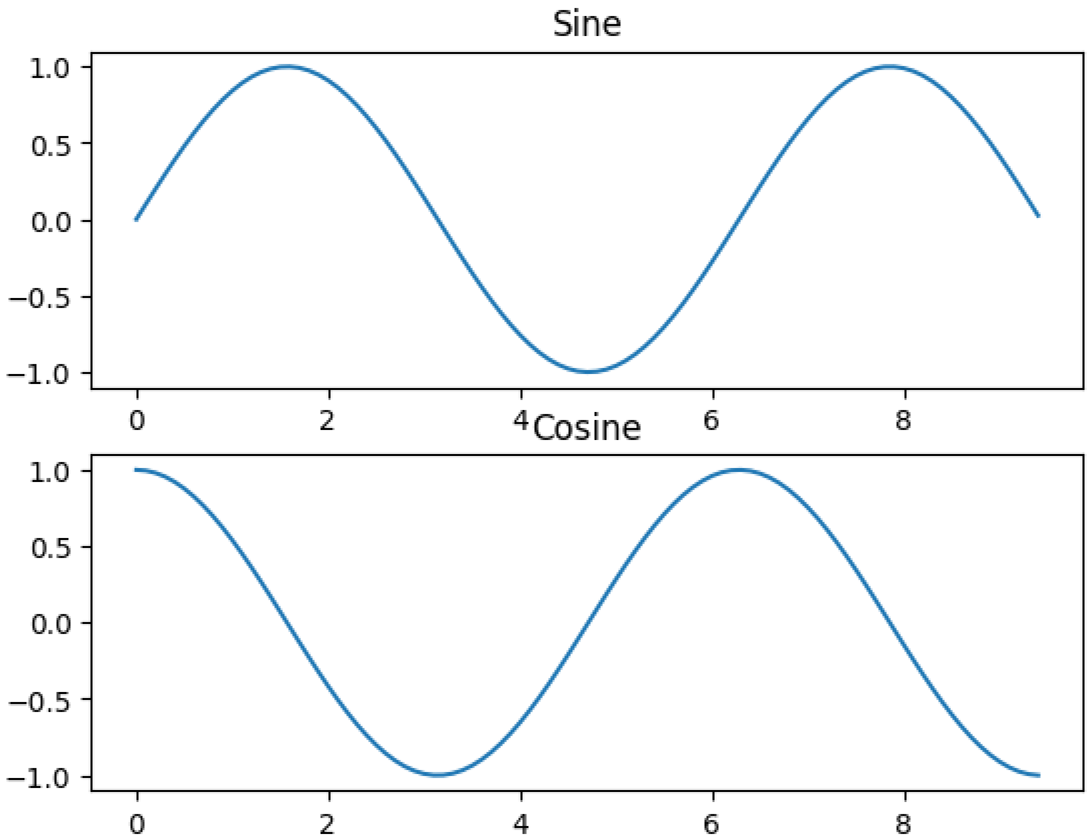
```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1
fig, axes = plt.subplots(2, 1)

# Make the first plot
axes[0].plot(x, y_sin)
axes[0].set_title('Sine')

# Make the second plot
axes[1].plot(x, y_cos)
axes[1].set_title('Cosine')

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



5.4. Additional Resources

- NumPy Official Documentation
- Matplotlib Official Documentation
- Python Data Science Handbook

---

6. Writing Clean Code: PEP 8

Writing clean, readable code is essential for collaboration and maintainability. PEP 8, Python's style guide, provides conventions to help Python developers write code that is clear and consistent. In this section, we'll explore some of these conventions through examples.

6.1. Function Naming and Definition

Functions should have descriptive names, and it's convention to use lowercase with words separated by underscores. This makes the function's purpose clear at a glance.

Good Example

In [44]:
```
def calculate_area(radius: float) -> float:
    """
    Return the area of a circle given its radius.

    Parameters:
    - radius (float): The radius of the circle.

    Returns:
    - float: The area of the circle.
    """
    pi = 3.14159
    return pi * (radius ** 2)
```

Bad Example

In [45]:
```
def ca(r):
    p = 3.14159
    return p * (r ** 2)
```

6.2. Variable Naming

Variable names should be short yet descriptive. Avoid using names that are too generic or too verbose.

Good Example

In [46]:
```
student_names = ["Alice", "Bob", "Charlie"]
```

Bad Example

In [47]:
```
sn = ["Alice", "Bob", "Charlie"]
list_of_names_of_students_in_the_class = ["Alice", "Bob", "Charlie"]
list1 = ["Alice", "Bob", "Charlie"]
```

6.3. Whitespace and Indentation

Proper use of whitespace and indentation is crucial for code readability.

Good Example

In [48]:
```
def greet(name):
    if name:
        return f"Hello, {name}!"
    else:
        return "Hello!"
```

Bad Example

In [50]:
```
# def greet(name):
#   if name:
#   return f"Hello, {name}!"
#   else:
#   return "Hello!"
```

6.4. Comments and Docstrings

While comments can be useful, it's often better to express the intent of your code through clear code and descriptive docstrings. Comments should be used sparingly, i.e

Good Example

In [68]:
```python
def is_prime(number: int) -> bool:
    """
    Check if a number is prime.

    Parameters:
    - number (int): The number to check.

    Returns:
    - bool: True if the number is prime, False otherwise.
    """
    if number <= 1:
        return False
    for i in range(2, number):
        if number % i == 0:
            return False
    return True
```

Bad Example

In [52]:
```
# def is_prime(n):
#     # This function checks if a number is prime
#     if n <= 1:  # If number is less than or equal to 1, it's not prime
#         return False
#     # Loop through numbers from 2 to the given number
#     for i in range(2, n):
#         # If number is divisible by any number in the loop, it's not prime
#         if n % i == 0:
#             return False
#     # If we've gone through the whole loop and didn't return False, it's pri
#     return True
```

6.5. Additional Resources

- PEP 8 Guide
- Real Python - PEP 8

TODO: Can you write a function to output all the prime numbers that are less than 100?

Extra: Can you visualize the number of prime and non-prime numbers in each interval of 10?

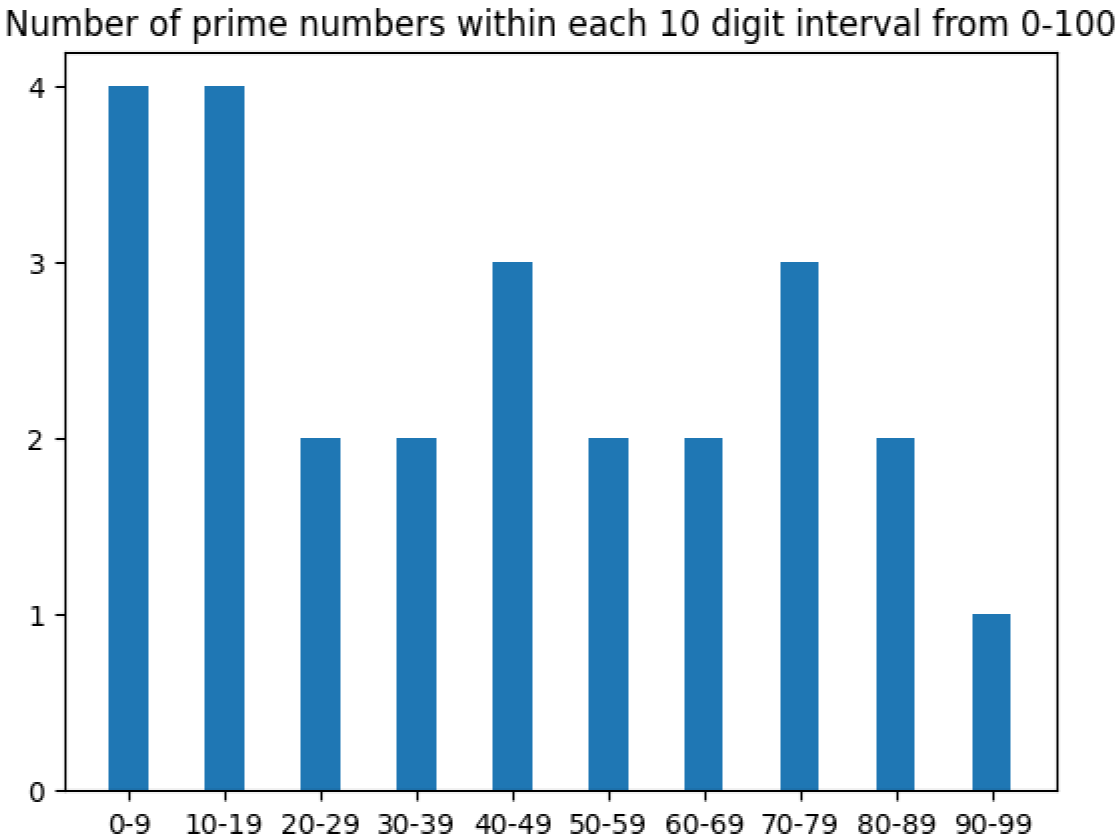
Hint: stacked bar plot using `plt.bar()` can be a good way to visualize the result.

In [71]:
```
xs = np.arange(10, 110, 10)
# plt.bar(xs)
is_prime(7)
```

Out[71]:
```
True
```

In [88]:
```
xs = np.arange(10, 110, 10)
ys = []
labels = []
for i in range(0, 10):
    start = 10 * i
    end = 10 * (i + 1)
    numprime = 0
    for num in range(start, end):
        if is_prime(num):
            numprime += 1
    ys.append(numprime)
    labels.append(f'{start}-{end-1}')
ys = np.array(ys)
plt.bar(xs, ys, width=4, tick_label=labels)
plt.title("Number of prime numbers within each 10 digit interval from 0-100")
plt.yticks(np.arange(0, 5, 1))
```

Out[88]:
```
([<matplotlib.axis.YTick at 0x13230b1c0>,
  <matplotlib.axis.YTick at 0x132308a00>,
  <matplotlib.axis.YTick at 0x13232a110>,
  <matplotlib.axis.YTick at 0x132374310>,
  <matplotlib.axis.YTick at 0x132374dc0>],
 [Text(0, 0, '0'),
  Text(0, 1, '1'),
  Text(0, 2, '2'),
  Text(0, 3, '3'),
  Text(0, 4, '4')])
```



In [ ]:

---

7. Questions?

In [ ]: