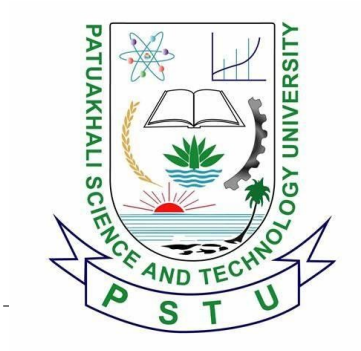


PATUAKHALI SCIENCE AND TECHNOLOGY UNIVERSITY

COURSE CODE CCE 312
Numerical Methods Sessional

SUBMITTED TO:

Prof. Dr. Md Samsuzzaman
Department of Computer and Communication Engineering
Faculty of Computer Science and Engineering

SUBMITTED BY:

Md. Sharafat Karim
ID: **2102024**,
Registration No: **10151**
Faculty of Computer Science and Engineering

Assignment 19
Assignment title: Final Lab Report
Date of submission: 17 Sun, Aug 2025

Contents

1. Python

1.a. Python as a Calculator

We can use python as a calculator to perform basic arithmetic operations. Python can handle addition, subtraction, multiplication, and division just like a standard calculator. Jupyter Notebook or python shell can be used to execute these operations interactively.

To trigger python shell, make sure you have python installed on your system. You can check this by running `python --version` in your terminal or command prompt. If you have python installed, you can start the python shell by typing `python` or `python3` depending on your installation.

And finally to start your shell, type `python` or `python3` in your terminal or command prompt.

1.a.i. *Basic Arithmetic Operations:*

TRY IT! Compute the sum of 1 and 2.

```
In [1]: 1 + 2
Out[1]: 3
```

An **order of operations** is a standard order of precedence that different operations have in relationship to one another. Python utilizes the same order of operations that you learned in grade school. Powers are executed before multiplication and division, which are executed before addition and subtraction. Parentheses, (), can also be used in Python to supersede the standard order of operations.

TRY IT! Compute $\frac{3*4}{(2^2+4/2)}$.

```
In [2]: (3*4)/(2**2 + 4/2)
Out[2]: 2.0
```

> **TIP!** You may have noticed `Out[2]` is the resulting value of the last operation executed. You can use _ symbol to represent this result to break up complicated expressions into simpler commands.

TRY IT! Compute 3 divided by 4, then multiply the result by 2, and then raise the result to the 3rd power.

```
In [3]: 3/4
Out[3]: 0.75

In [4]: _*2
Out[4]: 1.5

In [5]: _**3
Out[5]: 3.375
```

Python has many basic arithmetic functions like `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`, `log10` and `sqrt` stored in a module `math`.

```
In [6]: import math
```

TRY IT! Find the square root of 4.

```
In [7]: math.sqrt(4)
Out[7]: 2.0
```

TRY IT! Compute the $sin(\frac{\pi}{2})$.

```
In [8]: math.sin(math.pi/2)
Out[8]: 1.0
```

TRY IT! Compute e^{log10}.

```
In [9]: math.exp(math.log(10))
Out[9]: 10.000000000000002
```

Note that the `log` function in Python is log_e, or the natural logarithm. It is not log_{10}. If you want to use log_{10}, you need to use `math.log10`.

TRY IT! Compute $e^{\frac{3}{4}}$

```
In [10]: math.exp(3/4)
Out[10]: 2.117000016612675
```

TRY IT! Use the question mark to find the definition of the factorial function

```
In [11]: math.factorial?
```

```
Signature: math.factorial(x, /)
Docstring:
Find x!.

Raise a ValueError if x is negative or non-integral.
Type:      builtin_function_or_method
```

Python will raise an `ZeroDivisionError` when you have expression 1/0, which is infinity, to remind you.

```
In [12]: 1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call
last)
<ipython-input-12-9e1622b385b6> in <module>()
----> 1 1/0

ZeroDivisionError: division by zero
```

You can type `math.inf` at the command prompt to denote infinity or `math.nan` to denote something that is not a number that you wish to be handled as a number. If this is confusing, this distinction can be skipped for now; it will be explained more clearly when it becomes important. Finally, Python can also handle the imaginary number.

TRY IT! $1/\infty$, and $\infty * 2$ to verify that Python handles infinity as you would expect.

```
In [13]: 1/math.inf
Out[13]: 0.0

In [14]: math.inf * 2
Out[14]: inf
```

TRY IT! Compute ∞/∞

```
In [15]: math.inf/math.inf
Out[15]: nan
```

TRY IT! Compute sum 2 + 5i

```
In [16]: 2 + 5j
Out[16]: (2+5j)
```

Note that, in Python imaginary part is using j instead of i to represent.

Another way to represent complex number in Python is to use the complex function.

```
In [17]: complex(2,5)
Out[17]: (2+5j)
```

Python can also handle scientific notation using the letter e between two numbers. For example, $1e6 = 1000000$ and $1e-3 = 0.001$.

TRY IT! Compute the number of seconds in 3 years using scientific notation.

```
In [18]: 3e0*3.65e2*2.4e1*3.6e3
Out[18]: 94608000.0
```

TIP! Every time when we type the function in *math* module, we always type `math.function_name`. Alternatively, there is a simpler way, for example, if we want to use `sin` and `log` from `math` module, we could import them this way: `from math import sin, log`. Then you all you need to do when using these functions is using them directly, for example, `sin(20)` or `log(10)`.

1.a.ii. *Basic Data Types:*

We just learned to use Python as a calculator to deal with different data values. In Python, there are a few data types we need to know, for numerical values, int, float, and complex are the types associated with the values.

- **int**: Integers, such as 1, 2, 3, ...
- **float**: Floating-point numbers, such as 3.2, 6.4, ...
- **complex**: Complex numbers, such as 2 + 5j, 3 + 2j, ...

You can use function *type* to check the data type for different values.

TRY IT! Find out the data type for 1234.

```
In [19]: type(1234)
Out[19]: int
```

TRY IT! Find out the data type for 3.14.

```
In [20]: type(3.14)
Out[20]: float
```

TRY IT! Find out the data type for 2 + 5j.

```
In [21]: type(2 + 5j)
Out[21]: complex
```

Of course, there are other different data types, such as boolean, string and so on, we will introduce them later in the book.

1.a.iii. *Courtesy:*

This page is the summerized form of,

- https://pythonnumericalmethods.studentorg.berkeley.edu/notebooks/chapter01.02-Python-as-A-Calculator.html

To learn more about Python as a calculator, you can refer to the original source.

1.b. Packages and Virtual Environments

1.b.i. *1. `pip` and Virtual Environments:*

It is recommended to use a virtual environment to manage your project's dependencies. This keeps your packages isolated from the global Python installation.

Creating a Virtual Environment:

Open your terminal (Command Prompt on Windows, Terminal on macOS/Linux) on your *projects directory* and run:

```
python -m venv .venv
```

This creates a `.venv` folder in your *project directory*.

Activating the Virtual Environment:

- **Windows:**

```
.venv\Scripts\activate
```

- **macOS/Linux:**

```
source .venv/bin/activate
```

You should see your prompt change, indicating the environment is active.

Installing Packages:

With the virtual environment activated, install packages using:

```
pip install package_name
```

Replace `package_name` with the name of the package you want to install (e.g., `numpy`).

Requirements File:

To manage dependencies, you can create a `requirements.txt` file. This file lists all the packages your project depends on. You can create it by running:

```
pip freeze > requirements.txt
```

Or to create it more efficiently, take a look here,

- https://stackoverflow.com/questions/31684375/automatically-create-file-requirements-txtu

To install all packages listed in `requirements.txt`, run:

```
pip install -r requirements.txt
```

1.b.ii. *2. Importing Third-Party Packages:*

After installation, you can import the package in your Python script or notebook:

```
import package_name
```

For example:

```
import math # pre-installed module
from datetime import datetime # pre-installed module
import random as rnd # pre-installed module

# Using imported modules
print(math.sqrt(16))
print(math.pi)

# Current datetime
now = datetime.now()
print(now)

# Random number
print(rnd.randint(1, 10))

# Common data science imports (third party)
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

1.b.iii. *3. Using Jupyter Notebook:*

Jupyter Notebook is an interactive environment for running Python code.

Installation:

For windows you can install directly from their official website, or anaconda distribution. For macOS/Linux, you can install Jupyter from their respective package managers.

Running Jupyter:

If you are using anaconda, you can launch it from the Anaconda Navigator. For other cases, start Jupyter Notebook by running:

```
jupyter notebook
```

This will open a web interface in your browser. You can create new notebooks and run Python code interactively.

Importing Packages in Jupyter:

You can import packages in a notebook cell just like in a script:

```
import pandas as pd
```

Tip: If you need to install a package directly from a Jupyter notebook in windows, use:

```
!pip install package_name
```

Tip: And for macOS/Linux, you can use your package manager (like `pacman` or `apt` or `brew`) to install Python packages globally, but it's better to stick with virtual environments for project-specific dependencies

1.c. Variables & Data Structures

A quick guide to Python basics for programmers switching from other languages.

1.c.i. *Variables and Data Types:*

```
x = 10           # Integer
y = 3.14         # Float
name = "Alice"   # String
is_active = True # Boolean
print(x, y, name, is_active)
```

```
10 3.14 Alice True
```

1.c.ii. *String Operations:*

```
text = "  Hello, World!  "
print(text.strip())          # Remove whitespace
print(text.count('l'))       # count occurance
print(text.lower())          # Convert to lowercase
print(text.upper())          # Convert to uppercase
print(text.replace("World", "Python"))  # Replace substring
print(text.split(","))       # Split string
print(len(text))             # String length

print("-"*50)

w = "I Love L i n u x !"
print(w[0])
print(w[2:6])
print(w[7::2])

print("-"*50)

# String formatting
name = "Alice"
age = 30
print(f"My name is \"{name}\" and I am {age} years old")  # f-string
print("My name is {} and I am {} years old. Don't you get
it?".format(name, age))  # format method
```

```
Hello, World!
3
  hello, world!
  HELLO, WORLD!
  Hello, Python!
['  Hello', ' World!  ']
17
--------------------------------------------------
I
Love
Linux!
--------------------------------------------------
My name is "Alice" and I am 30 years old
My name is Alice and I am 30 years old. Don't you get it?
```

1.c.iii. *Lists:*

```
fruits = [1, 2, "apple", "banana", "cherry"]
print(fruits)
fruits.append("orange")
print(fruits)
print(fruits[-2:]) # Last 2
print(len(fruits)) # Length

print("-" * 50)

list_2 = ["orange", "mango"]
print(fruits + list_2)

del(list_2[1])
print(fruits + list_2)
print("mango" in list_2)
del(list_2)

print("-" * 50)

print(list("ArchLinux!"))
```

```
[1, 2, 'apple', 'banana', 'cherry']
[1, 2, 'apple', 'banana', 'cherry', 'orange']
['cherry', 'orange']
6
--------------------------------------------------
[1, 2, 'apple', 'banana', 'cherry', 'orange', 'orange', 'mango']
[1, 2, 'apple', 'banana', 'cherry', 'orange', 'orange']
False
--------------------------------------------------
['A', 'r', 'c', 'h', 'L', 'i', 'n', 'u', 'x', '!']
```

1.c.iv. *Tuples:*

```
point = (2, 3)
print(point)
# Tuples are immutable
```

```
(2, 3)
```

1.c.v. *Dictionaries:*

```
person = {"name": "Bob", "age": 25}
print(person)

person["age"] = 26
print(person['age'])

print(person.keys())
print(person.values())
print(len(person))

print('-' * 50)
```

```
print(dict([("UC Berkeley", "USA"), ('Oxford', 'UK')]))
print('UC Berkeley' in dict([("UC Berkeley", "USA"), ('Oxford', 'UK')]))
```

```
{'name': 'Bob', 'age': 25}
26
dict_keys(['name', 'age'])
dict_values(['Bob', 26])
2
--------------------------------------------------
{'UC Berkeley': 'USA', 'Oxford': 'UK'}
True
```

1.c.vi. *Sets:*

```
numbers = {1, 2, 3, 2}
print(numbers) # Duplicates removed
```

```
{1, 2, 3}
```

1.c.vii. *Conditional Statements:*

```
age = 18
if age >= 18:
    print("Adult")
elif age < 5:
    print("child")
else:
    print("Minor")
```

```
Adult
```

1.c.viii. *Loops:*

```
# For loop
fruits = [1, 2, "apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

print("-" * 50)

# While loop
count = 0
while count < 3:
    print(count)
    count += 1
```

```
1
2
apple
banana
cherry
--------------------------------------------------
0
1
2
```

1.c.ix. *Exception Handling:*

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

```
Cannot divide by zero!
```

1.c.x. *List Comprehensions:*

Efficient way to create lists - very useful for data processing.

```
# List comprehensions
squares = [x**2 for x in range(5)]
print(squares)

# List comprehension with condition
even_squares = [x**2 for x in range(10) if x % 2 == 0]
print(even_squares)

# Dictionary comprehension
square_dict = {x: x**2 for x in range(5)}
print(square_dict)

# Set comprehension
unique_squares = {x**2 for x in range(-3, 4)}
print(unique_squares)
```

```
[0, 1, 4, 9, 16]
[0, 4, 16, 36, 64]
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
{0, 9, 4, 1}
```

1.c.xi. *Regular Expressions:*

Pattern matching for text processing and data cleaning. To learn about regular expressions, refer to the Regular Expressions site, cause it's awesome.

```
import re
text = "Contact us at john@example.com or call 123-456-7890"

# Find email addresses
emails = re.findall(r'\w+@\w+\.\w+', text)
print('Emails:', emails)
```

```
Emails: ['john@example.com']
```

```
# Find phone numbers
phones = re.findall(r'\d{3}-\d{3}-\d{4}', text)
print('Phones:', phones)
```

```
Phones: ['123-456-7890']
```

```
# Replace phone numbers with a placeholder
clean_text = re.sub(r'\d{3}-\d{3}-\d{4}', '[PHONE]', text)
print('Cleaned text:', clean_text)
```

```
Cleaned text: Contact us at john@example.com or call [PHONE]
```

```
# Split text into words
words = re.split(r'\s+', text)
print('Words:', words)
```

```
Words: ['Contact', 'us', 'at', 'john@example.com', 'or', 'call',
'123-456-7890']
```

```
# Extract username, domain, and extension from email
pattern = r'(\w+)@(\w+)\.(\w+)'
match = re.search(pattern, text)
if match:
    print('Username:', match.group(1))
    print('Domain:', match.group(2))
    print('Extension:', match.group(3))
```

```
Username: john
Domain: example
Extension: com
```

1.d. Functions

1.d.i. *Built in Functions:*

```
print(type(len))
```

```
import numpy as np
print(type(np.linspace))
```

```
<class 'builtin_function_or_method'>
<class 'numpy._ArrayFunctionDispatcher'>
```

1.d.ii. *Custom Function:*

```
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))
```

```
Hello, Alice!
```

```
n = 42

def func():
    global n
    print(f'Within function: n is {n}')
    n = 3
    print(f'Within function: change n to {n}')

func()
print(f'Outside function: Value of n is {n}')
```

```
Within function: n is 42
Within function: change n to 3
Outside function: Value of n is 3
```

1.d.iii. *Nested Functions:*

```
import math

def my_dist_xyz(x, y, z):
    """
    x, y, z are 2D coordinates contained in a tuple
    output:
    d - list, where
        d[0] is the distance between x and y
        d[1] is the distance between x and z
        d[2] is the distance between y and z
    """

    def my_dist(x, y):
        """
        subfunction for my_dist_xyz
        Output is the distance between x and y,
        computed using the distance formula
        """
```

```
        out = math.sqrt((x[0]-y[0])**2+(x[1]-y[1])**2)
        return out

    d0 = my_dist(x, y)
    d1 = my_dist(x, z)
    d2 = my_dist(y, z)

    return [d0, d1, d2]
d = my_dist_xyz((0, 0), (0, 1), (1, 1))
print(d)
d = my_dist((0, 0), (0, 1))
```

```
[1.0, 1.4142135623730951, 1.0]
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
Cell In[10], line 3
      1 d = my_dist_xyz((0, 0), (0, 1), (1, 1))
      2 print(d)
----> 3 d = my_dist((0, 0), (0, 1))

NameError: name 'my_dist' is not defined
```

Here `my_dist` is a nested function, and so it's not defined outside of `my_dist_xyz`.

1.d.iv. *Lambda Functions:*

The syntax for a lambda function is:

```
lambda arguments: expression
```

```
square = lambda x: x**2

print(square(2))
print(square(5))
```

```
4
25
```

EXAMPLE: Sort [(1, 2), (2, 0), (4, 1)] based on the 2nd item in the tuple.

```
sorted([(1, 2), (2, 0), (4, 1)], key=lambda x: x[1])
```

```
[(2, 0), (4, 1), (1, 2)]
```

```
# Lambda functions
square = lambda x: x**2
print(square(5))

# Using lambda with map, filter, sort
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
```

```
print(squared)

# Filter even numbers
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)

# Sort by custom key
students = [("Alice", 85), ("Bob", 90), ("Charlie", 78)]
students.sort(key=lambda x: x[1])  # Sort by grade
print(students)
```

```
25
[1, 4, 9, 16, 25]
[2, 4]
[('Charlie', 78), ('Alice', 85), ('Bob', 90)]
```

1.d.v. *Recursion:*

```
def fibonacci_display(n):
    """Computes and returns the Fibonacci of n,
    a postive integer.
    """
    if n == 1: # first base case
        out = 1
        print(out)
        return out
    elif n == 2: # second base case
        out = 1
        print(out)
        return out
    else: # Recursive step
        out = fibonacci_display(n-1)+fibonacci_display(n-2)
        print(out)
        return out # Recursive call

fibonacci_display(5)
```

```
1
1
2
1
3
1
1
2
5

5
```

1.d.vi. *Divide and conquer (Bonus):*

Check the next section, “Condition & Loops”, if you don’t know about their syntax. For now, let’s focus on the function itself.

Tower of Hanoi:

```
def my_towers(N, from_tower, to_tower, alt_tower):
    """
    Displays the moves required to move a tower of size N from the
    'from_tower' to the 'to_tower'.

    'from_tower', 'to_tower' and 'alt_tower' are uniquely either
    1, 2, or 3 referring to tower 1, tower 2, and tower 3.
    """

    if N != 0:
        # recursive call that moves N-1 stack from starting tower
        # to alternate tower
        my_towers(N-1, from_tower, alt_tower, to_tower)

        # display to screen movement of bottom disk from starting
        # tower to final tower
        print("Move disk %d from tower %d to tower %d."\
                  %(N, from_tower, to_tower))

        # recursive call that moves N-1 stack from alternate tower
        # to final tower
        my_towers(N-1, alt_tower, to_tower, from_tower)

my_towers(3, 1, 3, 2)  # Move a tower of size 3 from tower 1 to tower 3
using tower 2 as an auxiliary
```

```
Move disk 1 from tower 1 to tower 3.
Move disk 2 from tower 1 to tower 2.
Move disk 1 from tower 3 to tower 2.
Move disk 3 from tower 1 to tower 3.
Move disk 1 from tower 2 to tower 1.
Move disk 2 from tower 2 to tower 3.
Move disk 1 from tower 1 to tower 3.
```

Quick Sort:

```
def my_quicksort(lst):

    if len(lst) <= 1:
        # list of length 1 is easiest to sort
        # because it is already sorted

        sorted_list = lst
    else:

        # select pivot as teh first element of the list
        pivot = lst[0]

        # initialize lists for bigger and smaller elements
        # as well those equal to the pivot
        bigger = []
        smaller = []
        same = []
```

```
        # loop through list and put elements into appropriate array

        for item in lst:
            if item > pivot:
                bigger.append(item)
            elif item < pivot:
                smaller.append(item)
            else:
                same.append(item)

        sorted_list = my_quicksort(smaller) + same +
my_quicksort(bigger)

    return sorted_list

# Example usage
unsorted_list = [3, 6, 8, 10, 1, 2, 1]
sorted_list = my_quicksort(unsorted_list)
print(sorted_list)  # Output: [1, 1, 2, 3, 6, 8, 10]
```

```
[1, 1, 2, 3, 6, 8, 10]
```

1.e. Condition & Loops

1.e.i. *Conditional Statements:*

```
age = 18
if age >= 18:
    print("Adult")
elif age < 5:
    print("child")
else:
    print("Minor")
```

```
Adult
```

```
def my_adder(a, b, c):
    """
    Calculate the sum of three numbers
    author
    date
    """

    # Check for erroneous input
    if not (
        isinstance(a, (int, float))
        or isinstance(b, (int, float))
        or isinstance(c, (int, float))
    ):
        raise TypeError("Inputs must be numbers.")
    # Return output
    return a + b + c

# Testing the my_adder function
result = my_adder(1, 2, 3)
print(f"Result: {result}")  # Returns 6
```

```
Result: 6
```

```
# Testing error handling (uncomment to test)
# try:
#     result = my_adder(1, '2', 3)
#     print(result)
# except TypeError as e:
#     print(f"Error: {e}")  # Raises TypeError: Inputs must be numbers.
```

1.e.ii. *Ternary Operator:*

Basic syntax for a ternary operator in Python:

```
value_if_true if condition else value_if_false
```

```
is_student = True
person = 'student' if is_student else 'not student'
print(person)
```

```
student
```

1.e.iii. *Loops:*

```python
# For loop
fruits = [1, 2, "apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

```
1
2
apple
banana
cherry
```

```python
# While loop
count = 0
while count < 3:
    print(count)
    count += 1
```

```
0
1
2
```

```python
# List
print("Looping through a list:")
for item in fruits:
    print(item)
```

```
Looping through a list:
1
2
apple
banana
cherry
```

```python
# Tuple
my_tuple = (10, 20, 30)
print("Looping through a tuple:")
for item in my_tuple:
    print(item)
```

```
Looping through a tuple:
10
20
30
```

```python
# Dictionary (keys)
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print("Looping through a dictionary (keys):")
for key in my_dict:
    print(key, ":", my_dict[key])
```

```
Looping through a dictionary (keys):
name : Alice
age : 25
city : New York
```

```python
# Dictionary (items)
print("Looping through a dictionary (items):")
```

```
for key, value in my_dict.items():
    print(key, ":", value)
```

```
Looping through a dictionary (items):
name : Alice
age : 25
city : New York
```

```
# range() generates a sequence of numbers.
# Syntax: range(start, stop, step)
# - start: The first number (default is 0)
# - stop: The sequence ends before this number (required)
# - step: Difference between each number (default is 1)

# Example 1: range(5) -> [0, 1, 2, 3, 4]
print("Example 1: range(5)")
for i in range(5):
    print(i)
```

```
Example 1: range(5)
0
1
2
3
4
```

```
# Example 2: range(2, 7) -> [2, 3, 4, 5, 6]
print("Example 2: range(2, 7)")
for i in range(2, 7):
    print(i)
```

```
Example 2: range(2, 7)
2
3
4
5
6
```

```
# Example 3: range(1, 10, 2) -> [1, 3, 5, 7, 9]
print("Example 3: range(1, 10, 2)")
for i in range(1, 10, 2):
    print(i)
```

```
Example 3: range(1, 10, 2)
1
3
5
7
9
```

```
# Converting range to a list
numbers = list(range(5))  # [0, 1, 2, 3, 4]
print("Converting range to list:")
print(numbers)
```

```
Converting range to list:
[0, 1, 2, 3, 4]
```

1.e.iv. *List Comprehensions:*

Efficient way to create lists - very useful for data processing.

```
# Basic list comprehension
squares = [x**2 for x in range(5)]
print("Basic list comprehension:")
print(squares)
```

```
Basic list comprehension:
[0, 1, 4, 9, 16]
```

```
# List comprehension with condition
even_squares = [x**2 for x in range(10) if x % 2 == 0]
print("List comprehension with condition:")
print(even_squares)
```

```
List comprehension with condition:
[0, 4, 16, 36, 64]
```

```
# Dictionary comprehension
square_dict = {x: x**2 for x in range(5)}
print("Dictionary comprehension:")
print(square_dict)
```

```
Dictionary comprehension:
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

```
# Dictionary comprehension from a list of tuples
tuples = [('a', 1), ('b', 2), ('c', 3)]
tuple_dict = {key: value for key, value in tuples}
print("Dictionary comprehension from tuples:")
print(tuple_dict)
```

```
Dictionary comprehension from tuples:
{'a': 1, 'b': 2, 'c': 3}
```

```
# Set comprehension
unique_squares = {x**2 for x in range(-3, 4)}
print("Set comprehension:")
print(unique_squares)
```

```
Set comprehension:
{0, 9, 4, 1}
```

1.e.v. *Iterators and Generators:*

Memory-efficient ways to work with large datasets.

```
# Generator function
def fibonacci_generator(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# Using generator
fib = fibonacci_generator(10)
```

```
print("Using generator:")
print(list(fib))
```

```
Using generator:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
# Generator expression
squares_gen = (x**2 for x in range(5))
print("Generator expression:")
print(list(squares_gen))
```

```
Generator expression:
[0, 1, 4, 9, 16]
```

```
# Using range (which is an iterator)
print("Using range as iterator:")
for i in range(3):
    print(i)
```

```
Using range as iterator:
0
1
2
```

```
# enumerate - useful for getting index and value
fruits = ["apple", "banana", "cherry"]
print("Using enumerate:")
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
```

```
Using enumerate:
0: apple
1: banana
2: cherry
```

```
# zip - combining multiple iterables
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]
print("Using zip:")
for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

```
Using zip:
Alice is 25 years old
Bob is 30 years old
Charlie is 35 years old
```

1.f. File Handling

```
# Writing to a file
with open("sample.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a sample file.\n")

# Reading from a file
with open("sample.txt", "r") as file:
    content = file.read()
    print(content)

# Reading line by line
with open("sample.txt", "r") as file:
    for line in file:
        print(line.strip())

# Working with CSV-like data
import csv

# Writing CSV
data = [["Name", "Age"], ["Alice", 25], ["Bob", 30]]
with open("people.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(data)

# Reading CSV
with open("people.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)

# Delete the created files
import os
os.remove("sample.txt")
os.remove("people.csv")
```

```
Hello, World!
This is a sample file.

Hello, World!
This is a sample file.
['Name', 'Age']
['Alice', '25']
['Bob', '30']
```

1.g. OOP

OOP (Object Oriented Programming) is a programming paradigm that uses objects and classes to structure code. It allows for encapsulation, inheritance, and polymorphism, making it easier to manage complex software systems.

```
# 0. Class Definition
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        return f"Hi, I'm {self.name} and I'm {self.age} years old"

    def have_birthday(self):
        self.age += 1

# Create objects
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

print(person1.introduce())
person1.have_birthday()
print(f"After birthday: {person1.age}")
```

```
Hi, I'm Alice and I'm 25 years old
After birthday: 26
```

```
# 1. Class and Object
class Animal:
  def __init__(self, name):
    self.name = name  # Attribute

  def speak(self):      # Method
    return f"{self.name} makes a sound."

# Create an object
dog = Animal("Dog")
print(dog.speak())
```

```
Dog makes a sound.
```

```
# 2. Inheritance
class Dog(Animal):
  def speak(self): # Override the method
    return f"{self.name} barks."

my_dog = Dog("Buddy")
print(my_dog.speak())
```

```
Buddy barks.
```

```
# 3. Encapsulation
class BankAccount:
  def __init__(self, balance):
```

```
    self.__balance = balance  # Private attribute

  def deposit(self, amount):
    self.__balance += amount

  def get_balance(self):
    return self.__balance

account = BankAccount(100)
account.deposit(50)
print(account.get_balance())
```

```
150
```

```
# 4. Polymorphism
class Cat(Animal):
  def speak(self):
    return f"{self.name} meows."

animals = [Dog("Rex"), Cat("Whiskers")]
for animal in animals:
  print(animal.speak())
```

```
Rex barks.
Whiskers meows.
```

```
# 5. Abstraction

from abc import ABC, abstractmethod

class Vehicle(ABC):
  @abstractmethod
  def drive(self):
    pass

class Car(Vehicle):
  def drive(self):
    return "Car is driving"

car = Car()
print(car.drive())
```

```
Car is driving
```

```
# 6. Class vs Instance Attributes
class Counter:
  count = 0  # Class attribute

  def __init__(self):
    Counter.count += 1

print(Counter.count)
c1 = Counter()
c2 = Counter()
print(Counter.count)
```

```
0
2
```

```
# 7. Static and Class Methods
class MathUtils:
  var = "MathUtils"

  @staticmethod
  def add(a, b):
    return a + b

  @classmethod
  def description(cls):
    return f"This is a math utility class: {cls.var}"

print(MathUtils.add(2, 3))
print(MathUtils.description())
```

```
5
This is a math utility class: MathUtils
```

1.h. Error Handling

1.h.i. *Introduction:*

Error handling is an essential part of programming in Python. It allows you to anticipate and manage errors that may occur during the execution of your code.

```
try:
    x = 5 / 0
except ZeroDivisionError:
    print("Error: Division by zero is not allowed!")
```

```
Error: Division by zero is not allowed!
```

1.h.ii. *Types of Errors:*

Built in:

```
try:
    x = 5 / 2
except ZeroDivisionError:
    print("Error: Division by zero is not allowed!")
else:
    print("No errors occurred!")
```

```
No errors occurred!
```

```
try:
    x = 5 / 0
except ZeroDivisionError:
    print("Error: Division by zero is not allowed!")
finally:
    print("This code will always be executed!")
```

```
Error: Division by zero is not allowed!
This code will always be executed!
```

Custom exceptions:

```
class CustomError(Exception):
    pass

try:
    raise CustomError("This is a custom error!")
except CustomError as e:
    print(f"Error: {e}")
```

```
Error: This is a custom error!
```

1.i. Socket Programming

Socket programming is a way to connect two nodes on a network to communicate with each other. One socket (node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.

1.i.i. *Simple Calculator:*

Let's build a simple calculator using socket programming. The server will perform basic arithmetic operations like addition, subtraction, multiplication, and division based on the client's request.

First let's import `socket` and `threading` libraries. The `socket` library provides low-level networking interface, while the `threading` library allows us to handle multiple clients simultaneously.

```
import socket
import threading
```

Now a basic function to calculate the result based on the operation requested by the client.

```
def calculate(expression):
    """Evaluate a simple arithmetic expression"""
    try:
        if not all(c in "0123456789+-*/. " for c in expression):
            return "Error: Invalid characters in expression."

        result = eval(expression)
        return str(result)
    except Exception as e:
        return f"Error: {e}"
```

Basic server (single client):

First let's build a basic server that can handle a single client. The server will listen for incoming connections, receive the operation and numbers from the client, perform the calculation, and send back the result.

```
def start_server(host="127.0.0.1", port=9999):
    """Start the calculator server"""
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print(f"Calculator server listening on {host}:{port}")
        conn, addr = s.accept()
        with conn:
            print("Connected by", addr)
            while True:
                data = conn.recv(1024)
                if not data:
```

```
                    break
                expression = data.decode("utf-8")
                result = calculate(expression)
                conn.sendall(result.encode("utf-8"))
```

Basic Server (Multiple Clients):

We can enhance the server to handle multiple clients simultaneously using threading. Each client connection will be handled in a separate thread.

```
def start_threaded_server(host="127.0.0.1", port=9999):
    """Start the calculator server with threading support"""
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen()
    print(f"Threaded calculator server listening on {host}:{port}")

    def handle_client(conn, addr):
        print("Connected by", addr)
        with conn:
            while True:
                data = conn.recv(1024)
                if not data:
                    break
                expression = data.decode("utf-8")
                result = calculate(expression)
                conn.sendall(result.encode("utf-8"))
        print("Disconnected from", addr)

    try:
        while True:
            conn, addr = server_socket.accept()
            client_thread = threading.Thread(target=handle_client,
args=(conn, addr))
            client_thread.daemon = True
            client_thread.start()
    except KeyboardInterrupt:
        print("\nShutting down server.")
    finally:
        server_socket.close()
```

And finally call the server function to start the server.

```
start_threaded_server()
```

Client Code:

And finally the client code to connect to the server and send requests.

```
def client_program(host="127.0.0.1", port=9999):
    """Start a simple calculator client"""

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
```

```
        try:
            s.connect((host, port))
            print(f"✅ Connected to calculator server at {host}:{port}")

            while True:
                expression = input("Enter expression (or 'exit' to
quit): ")
                if expression.lower() == "exit":
                    print("👋 Exiting calculator client.")
                    break

                s.sendall(expression.encode("utf-8"))
                data = s.recv(1024)
                print(f"Result: {data.decode('utf-8')}")

        except ConnectionRefusedError:
            print(
                f"❌ Could not connect to server at {host}:{port}. Is
the server running?"
            )
        except Exception as e:
            print(f"❌ An error occurred: {e}")
```

2. Numerical Methods

2.a. Numpy

2.a.i. *Introduction:*

Numpy is a powerful library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

First of all, let's import it,

```
import numpy as np
```

> If you are facing problems then most probably numpy isn't installed in your system. Follow the official documentation to install it: Numpy Installation

2.a.ii. *Array:*

We can create a numpy array using the `numpy.array()` function. Here's an example,

```
x = np.array([1, 2, 3, 4, 5])
x
```

```
array([1, 2, 3, 4, 5])
```

```
y = np.array([[6, 7, 8], [1, 2, 3]])
y
```

```
array([[6, 7, 8],
       [1, 2, 3]])
```

```
# Also we can do some operations on the array. Like,
print(y.shape)
print(y.size)
```

```
(2, 3)
6
```

```
# Just like lists, we can access the elements of the array using
indexing.
print(y)
print("-" * 50)

print(y[0, 1])
print(y[0][1])

print(y[0])

print("-" * 50)

print(y[:, 1:]) # This will return all rows and columns from index 1 to
the end

print("-" * 50)
```

```
print(y[:, [0, 2]]) # This will return the first and third columns of
the array
```

```
[[6 7 8]
 [1 2 3]]
--------------------------------------------------
7
7
[6 7 8]
--------------------------------------------------
[[7 8]
 [2 3]]
--------------------------------------------------
[[6 8]
 [1 3]]
```

```
a = np.arange(0, 7)
print(a)

a[3] = 7
print(a)

a[:3] = 5
print(a)
```

```
[0 1 2 3 4 5 6]
[0 1 2 7 4 5 6]
[5 5 5 7 4 5 6]
```

```
x = [1, 4, 9, 16]
np.sqrt(x)
```

```
array([1., 2., 3., 4.])
```

2.a.iii. *Arrange:*

Arrange is used to create a sequence of numbers.

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.arange(10, 20, 2)
```

```
array([10, 12, 14, 16, 18])
```

2.a.iv. *Linspace:*

Linespace is used to create evenly spaced numbers over a specified range.

```
np.linspace(0, 1, 5)
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

2.a.v. *Zero or One:*

Let's generate arrays with zero or one!

```
np.zeros(5)
```

```
array([0., 0., 0., 0., 0.])
```

```
np.zeros((2, 3)) # This will create a 2x3 array filled with zeros
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
np.ones((2, 5)) # This will create a 2x5 array filled with ones
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

```
np.empty(5) # This will create an array filled with uninitialized values
```

```
array([0., 0., 0., 0., 0.])
```

2.a.vi. *Matrix operations:*

```
b = np.zeros((2, 2))
b[0, 0] = 1
b[0, 1] = 2
b[1, 0] = 3
b[1, 1] = 4
b
```

```
array([[1., 2.],
       [3., 4.]])
```

```
b + 2
```

```
array([[3., 4.],
       [5., 6.]])
```

```
b - 2
```

```
array([[-1.,  0.],
       [ 1.,  2.]])
```

```
b * 2
```

```
array([[2., 4.],
       [6., 8.]])
```

```
b / 2
```

```
array([[0.5, 1. ],
       [1.5, 2. ]])
```

```
b ** 2
```

```
array([[ 1.,  4.],
       [ 9., 16.]])
```

```
b = np.array([[1, 2], [3, 4]])
d = np.array([[3, 4], [5, 6]])
```

```
b + d
```

```
array([[ 4,  6],
       [ 8, 10]])
```

```
b - d
```

```
array([[-2, -2],
       [-2, -2]])
```

```
b * d # it's not matrix multiplication, it's element-wise multiplication
```

```
array([[ 3,  8],
       [15, 24]])
```

```
b / d
```

```
array([[0.33333333, 0.5       ],
       [0.6       , 0.66666667]])
```

```
b.T # This will transpose the array
```

```
array([[1, 3],
       [2, 4]])
```

2.a.vii. *Comparing:*

```
x = np.array([1, 2, 4, 5, 9, 3])
y = np.array([0, 2, 3, 1, 2, 3])
```

```
x > 3
```

```
array([False, False,  True,  True,  True, False])
```

```
x > y
```

```
array([ True, False,  True,  True,  True, False])
```

```
z = x[x > 3]
z # Values in x that are greater than 3
```

```
array([4, 5, 9])
```

```
x[x > 3] = 0
x # Now all values in x that are greater than 3 are set to 0
```

```
array([1, 2, 0, 0, 0, 3])
```

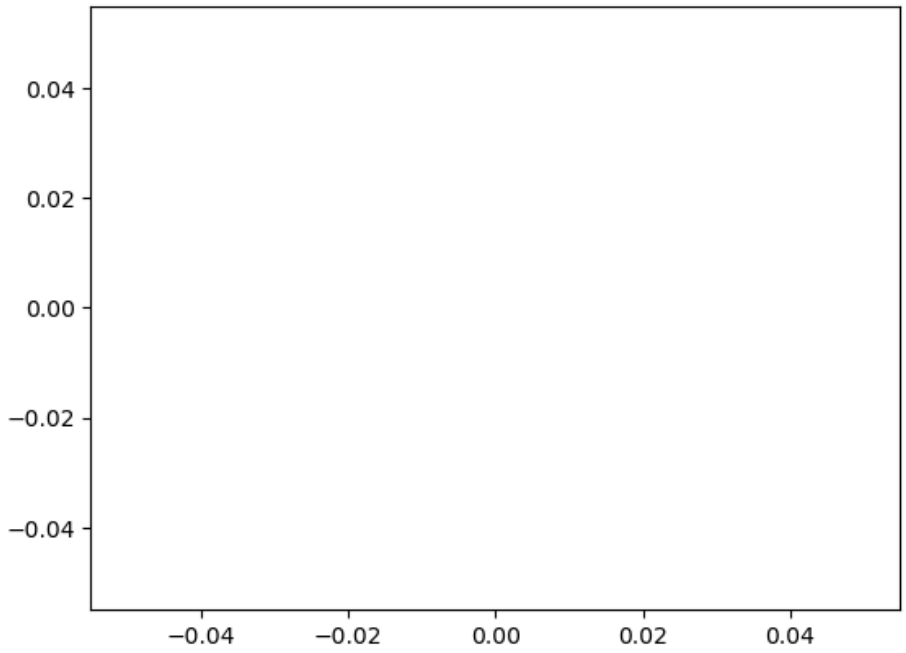
2.b. Mastering Matplotlib!

```
import matplotlib.pyplot as plt
```

2.b.i. *Line Plots:*

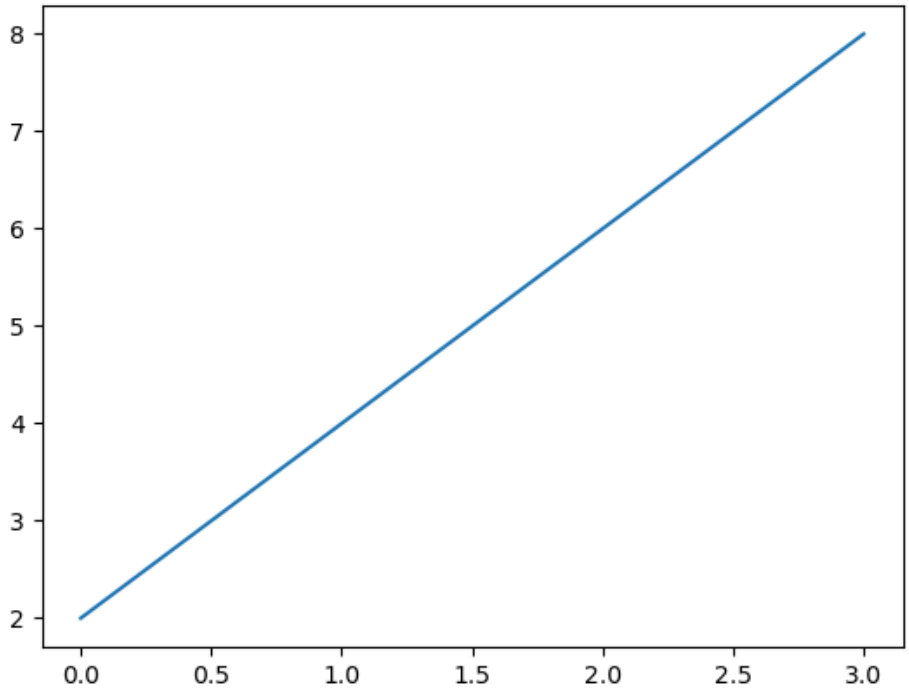
```
plt.plot()
```

```
[]
```



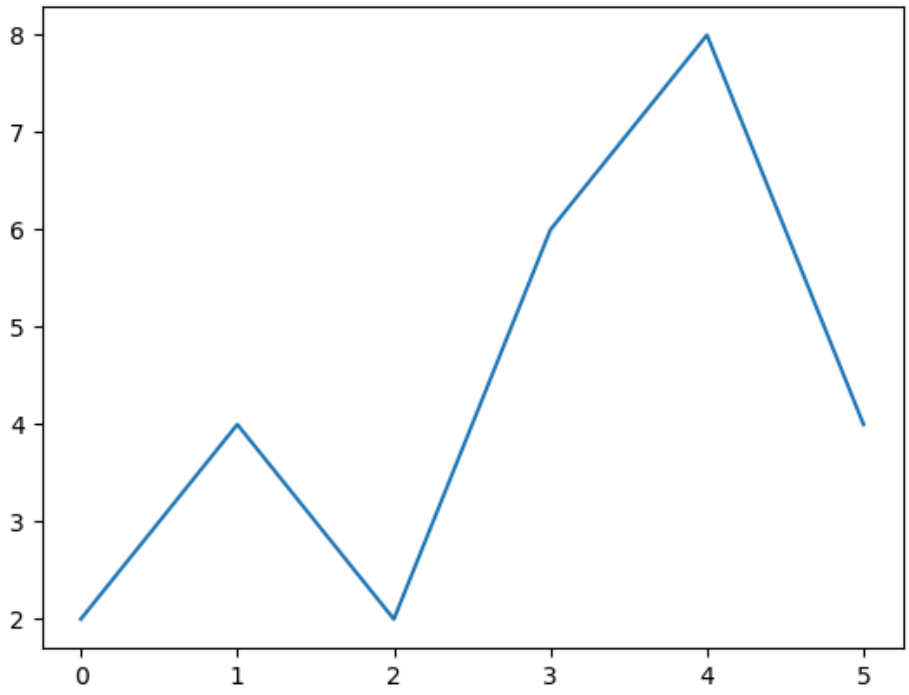

```
plt.plot([2,4,6,8])
```

```
[<matplotlib.lines.Line2D at 0x7f17628f5810>]
```



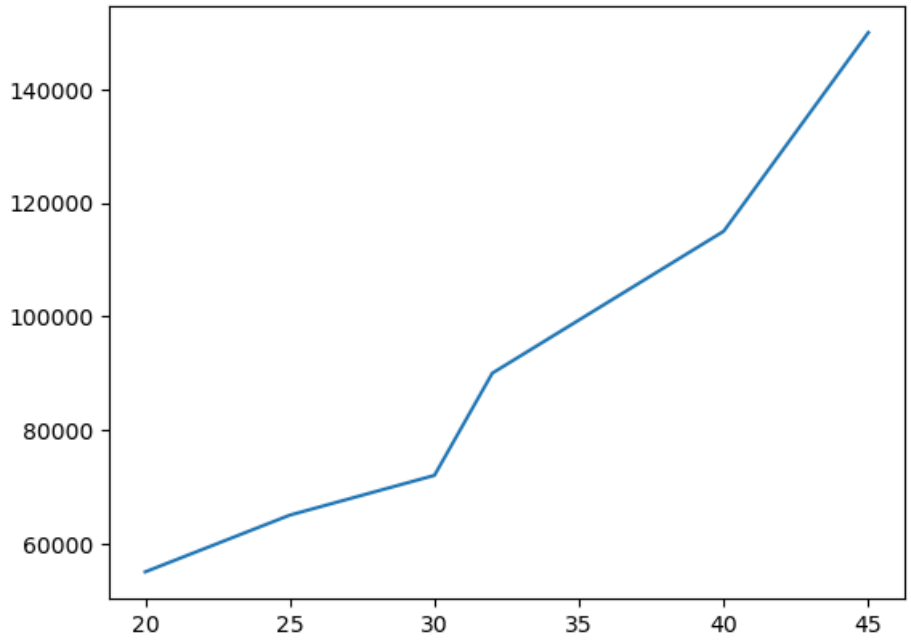

```
plt.plot([2,4,2,6,8,4])
```

```
[<matplotlib.lines.Line2D at 0x7f1762be1f90>]
```

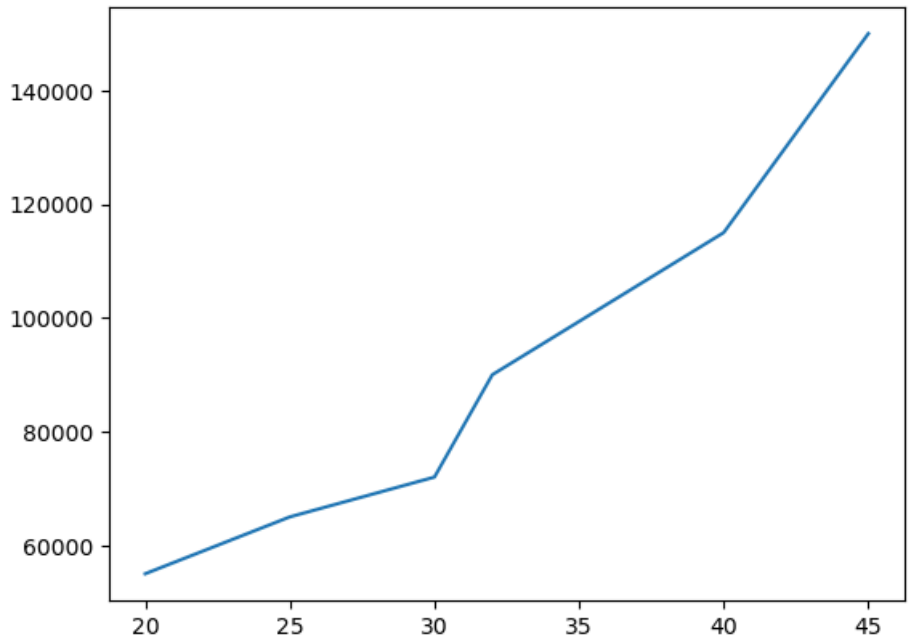



```
salaries=[55000,65000,72000,90000,115000,150000]
ages = [20,25,30,32,40,45]
plt.plot(ages, salaries)
```

```
[<matplotlib.lines.Line2D at 0x7f176299ccd0>]
```




```
plt.plot(ages, salaries)
plt.show()
```

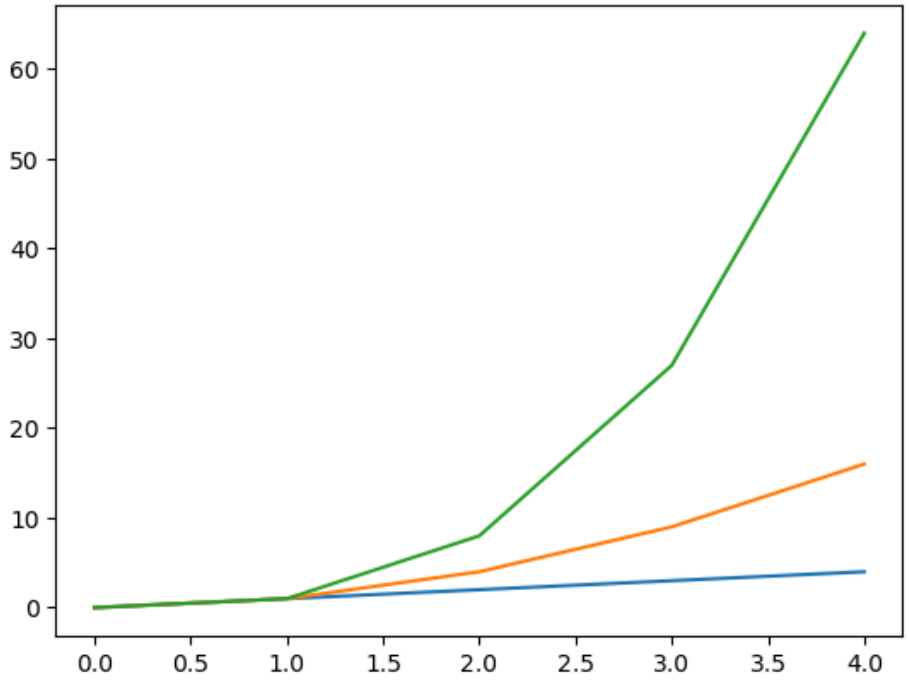



```
import numpy as np
nums = np.arange(5)
nums
```

```
array([0, 1, 2, 3, 4])
```

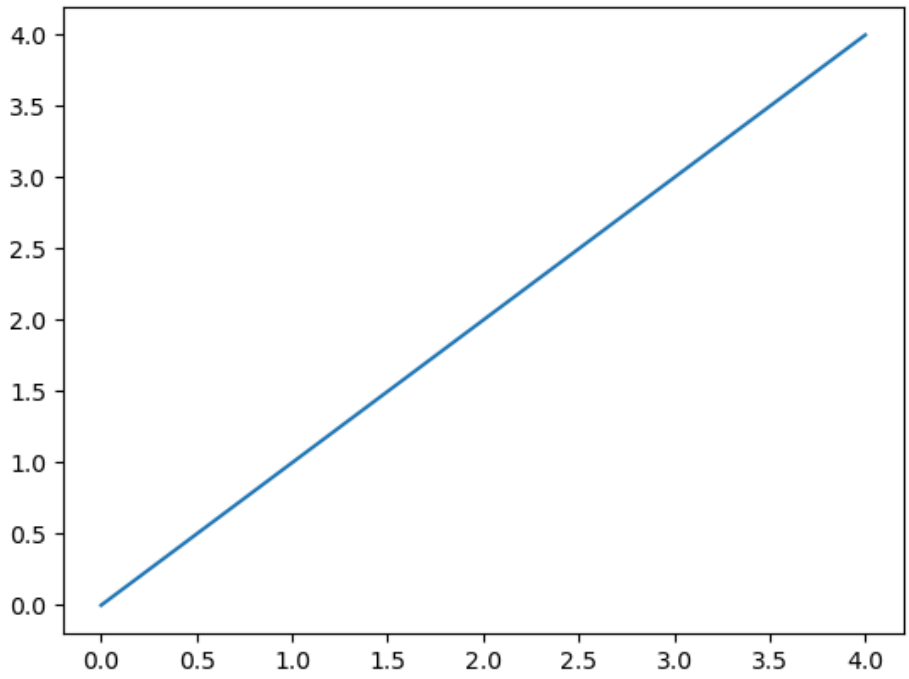
```
plt.plot(nums,nums)
plt.plot(nums, nums*nums)
plt.plot(nums, nums**3)
```

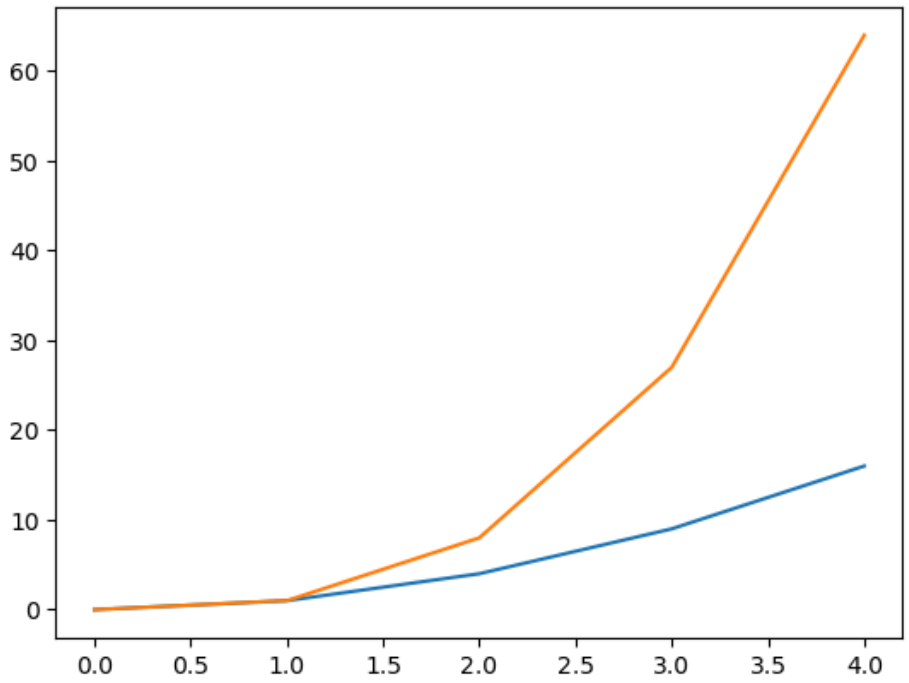
```
[<matplotlib.lines.Line2D at 0x7f1762a87c50>]
```




```
plt.figure()
plt.plot(nums,nums)
plt.figure()
plt.plot(nums, nums*nums)
plt.plot(nums, nums**3)
```

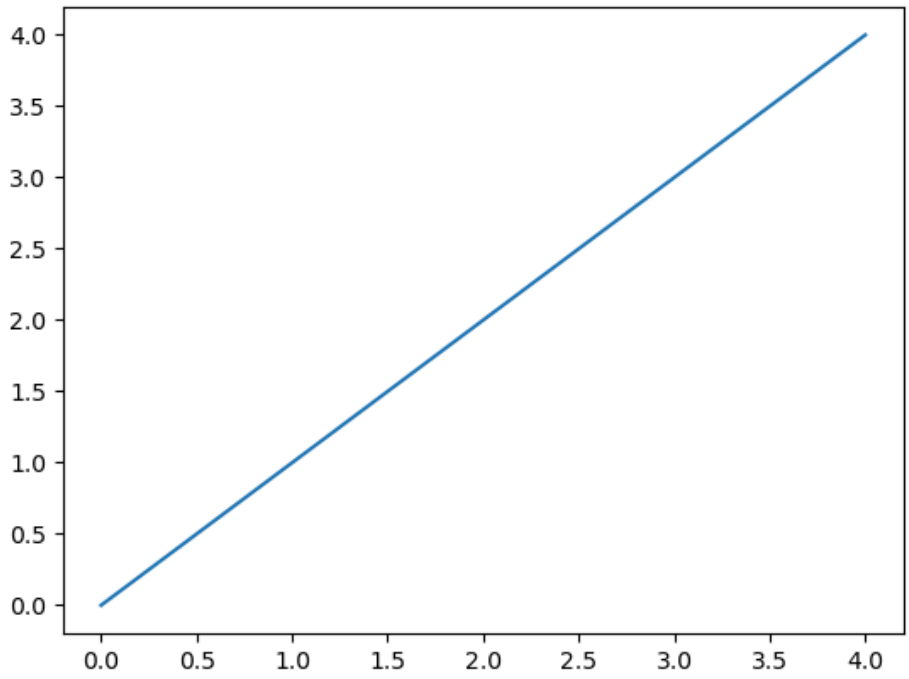
```
[<matplotlib.lines.Line2D at 0x7f1762822e90>]
```

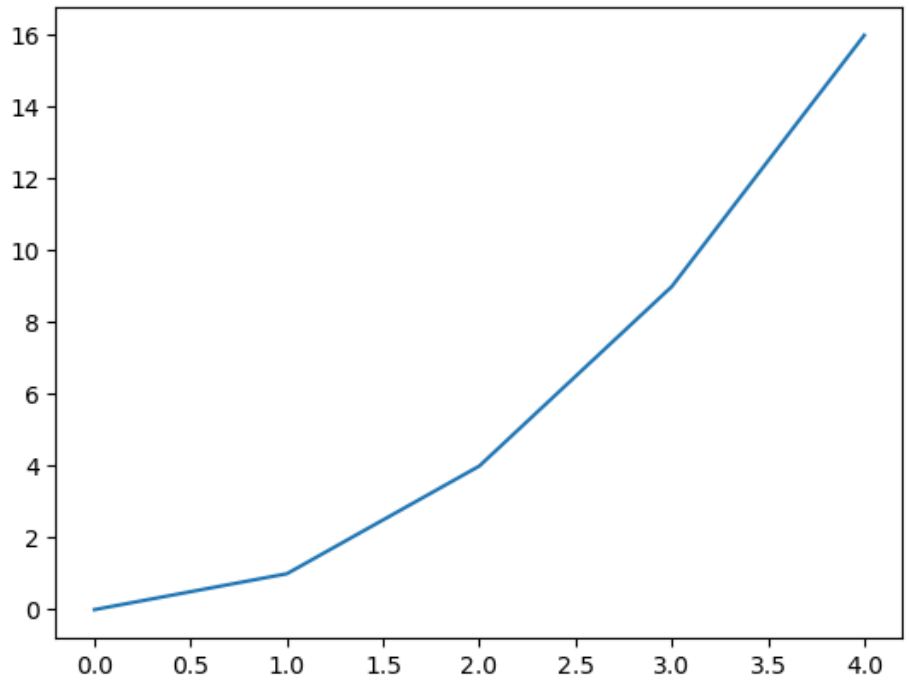


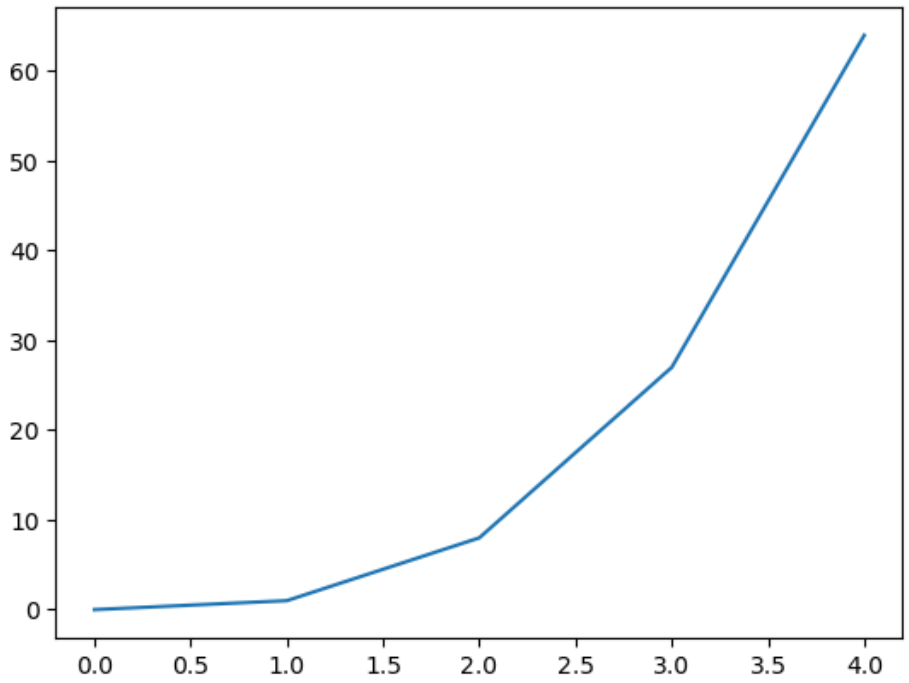



```
plt.figure()
plt.plot(nums,nums)
plt.figure()
plt.plot(nums, nums*nums)
plt.figure()
plt.plot(nums, nums**3)
```

```
[<matplotlib.lines.Line2D at 0x7f176271d450>]
```


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

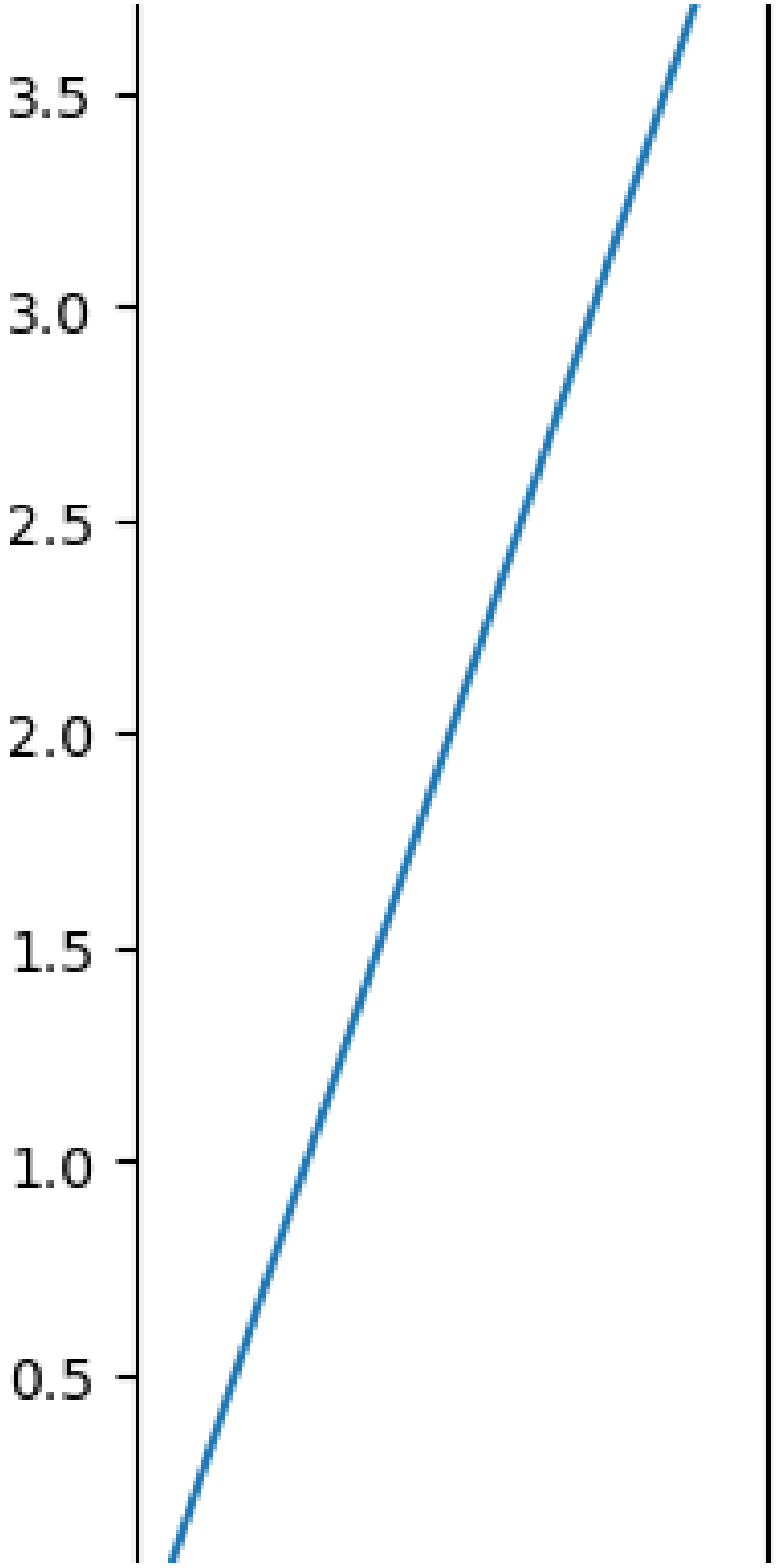

16
14
12
10
8
6
4
2
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0



figsize:

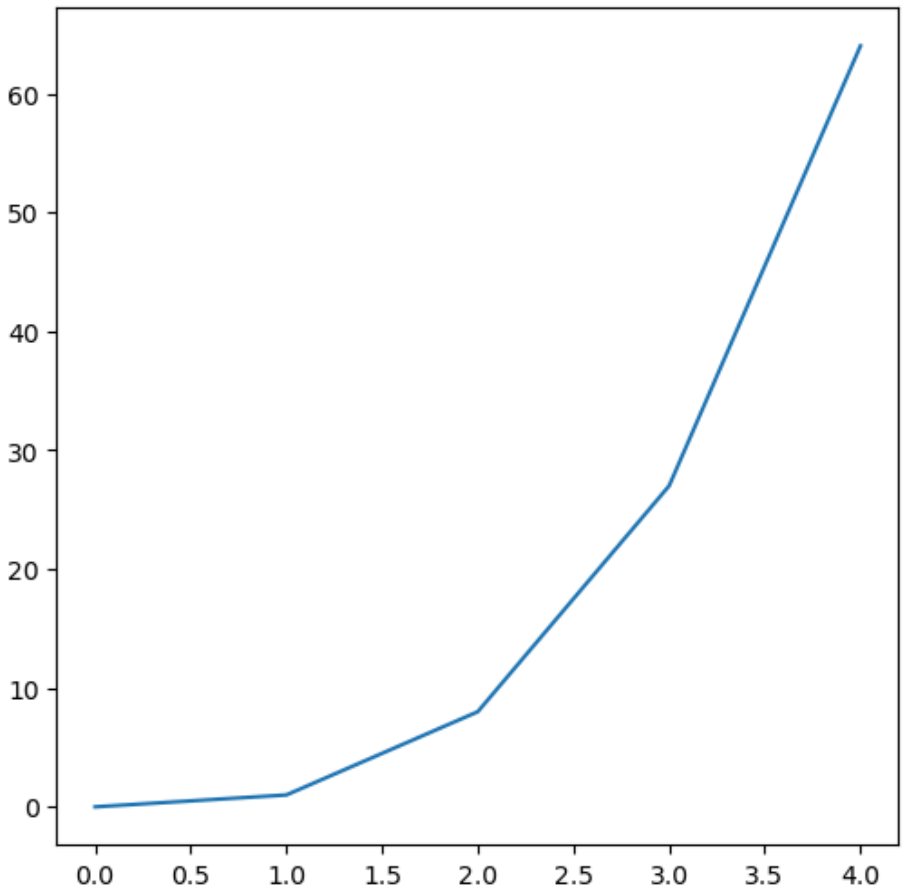
```
plt.figure(figsize=(2,6))
plt.plot(nums,nums)
```

```
[<matplotlib.lines.Line2D at 0x7f176262bd90>]
```


3.5
3.0
2.5
2.0
1.5
1.0
0.5

```
plt.figure(figsize=(6,6))
plt.plot(nums,nums**3)
```

```
[<matplotlib.lines.Line2D at 0x7f17624c1bd0>]
```



2.b.ii. *Plot styles:*

```
plt.style.available
```

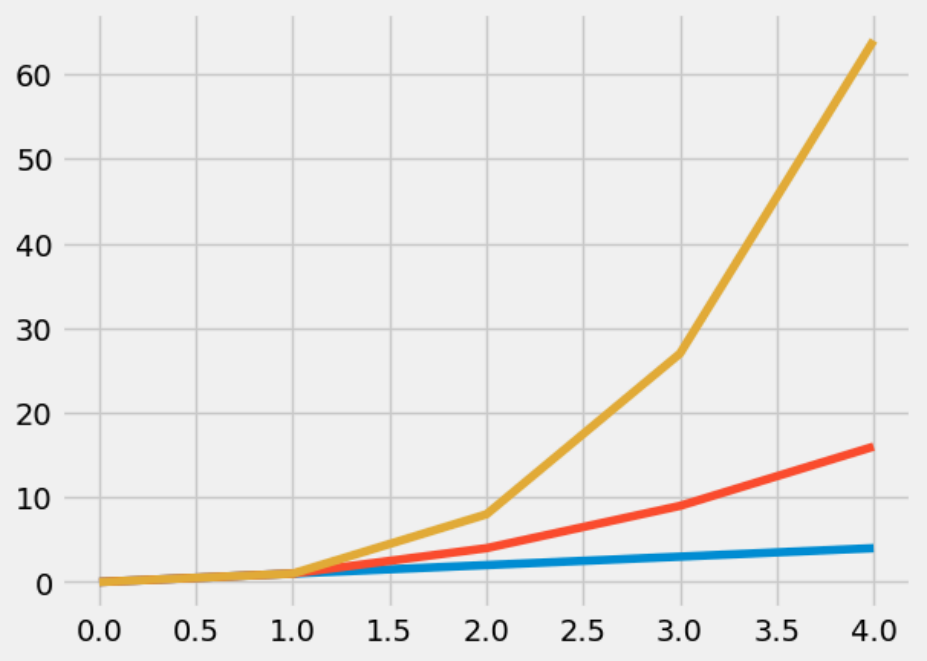
```
['Solarize_Light2',
 '_classic_test_patch',
 '_mpl-gallery',
 '_mpl-gallery-nogrid',
 'bmh',
 'classic',
 'dark_background',
 'fast',
 'fivethirtyeight',
 'ggplot',
 'grayscale',
 'petroff10',
 'seaborn-v0_8',
 'seaborn-v0_8-bright',
 'seaborn-v0_8-colorblind',
 'seaborn-v0_8-dark',
```

```
 'seaborn-v0_8-dark-palette',
 'seaborn-v0_8-darkgrid',
 'seaborn-v0_8-deep',
 'seaborn-v0_8-muted',
 'seaborn-v0_8-notebook',
 'seaborn-v0_8-paper',
 'seaborn-v0_8-pastel',
 'seaborn-v0_8-poster',
 'seaborn-v0_8-talk',
 'seaborn-v0_8-ticks',
 'seaborn-v0_8-white',
 'seaborn-v0_8-whitegrid',
 'tableau-colorblind10']
```

```
plt.style.use('fivethirtyeight')
```

```
plt.plot(nums,nums)
plt.plot(nums, nums*nums)
plt.plot(nums, nums**3)
```

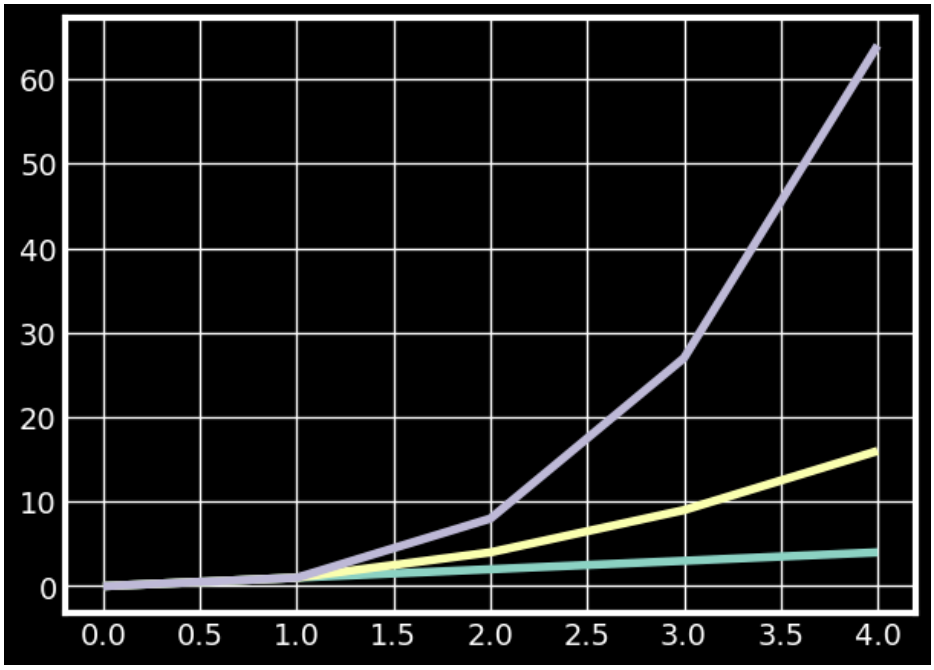
```
[<matplotlib.lines.Line2D at 0x7f176271e850>]
```




```
plt.style.use('dark_background')
```

```
plt.plot(nums,nums)
plt.plot(nums, nums*nums)
plt.plot(nums, nums**3)
```

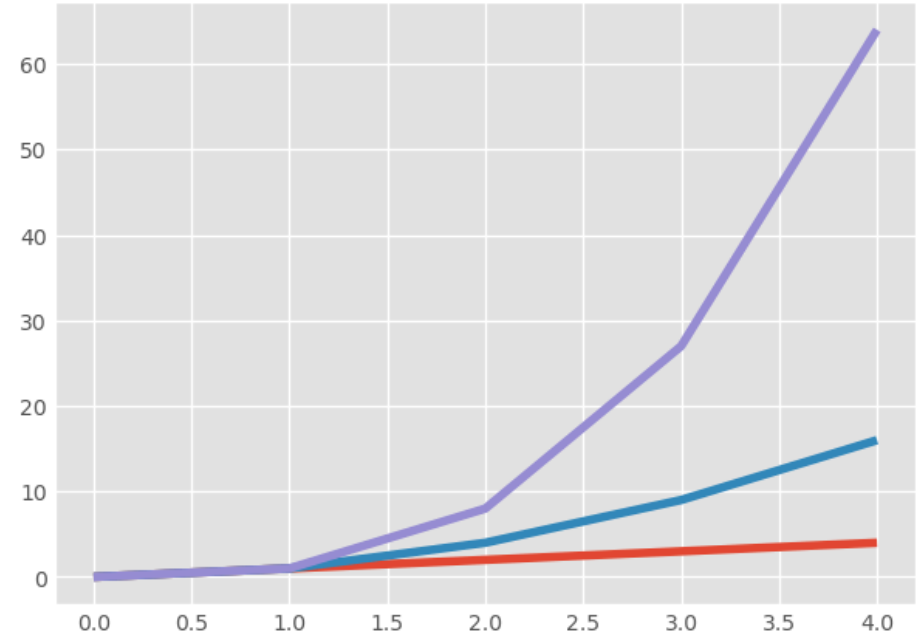
```
[<matplotlib.lines.Line2D at 0x7f1762a28910>]
```




```
plt.style.use('ggplot')
```

```
plt.plot(nums, nums)
plt.plot(nums, nums*nums)
plt.plot(nums, nums**3)
```

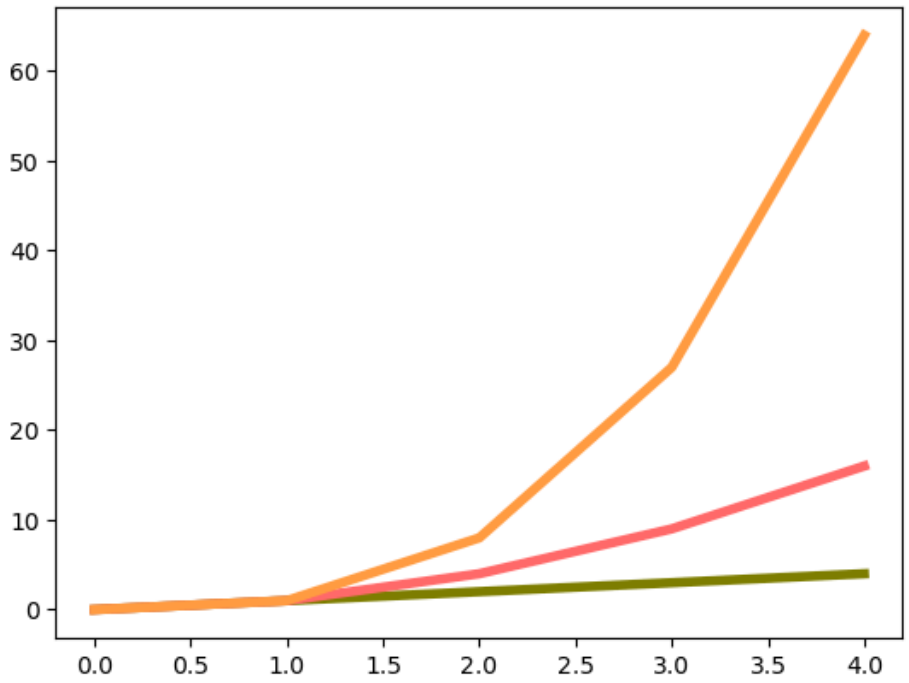
```
[<matplotlib.lines.Line2D at 0x7f1762873110>]
```




```
plt.style.use('default')
```

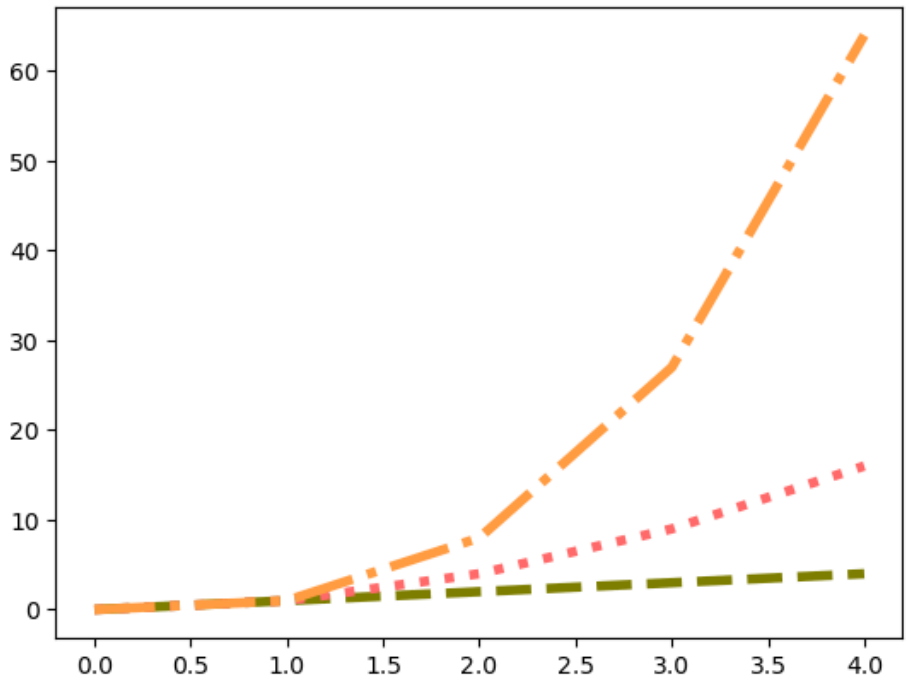
```
plt.plot(nums, nums, color="olive", linewidth=4)
plt.plot(nums, nums*nums, color="#ff6b6b", linewidth=4)
plt.plot(nums, nums**3, c="#ff9f43", linewidth=4)
```

```
[<matplotlib.lines.Line2D at 0x7f176253b610>]
```



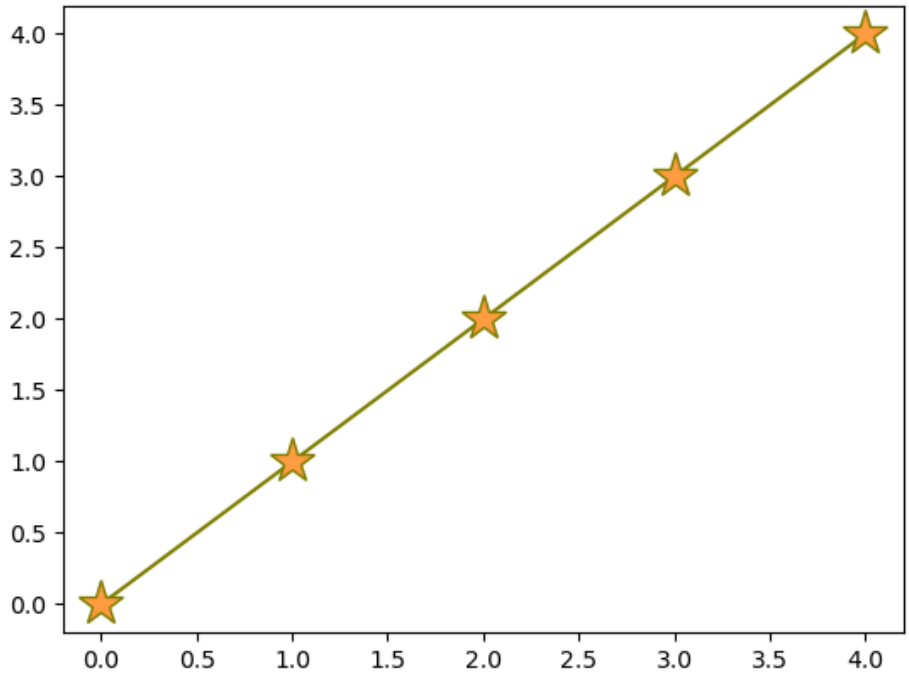

```
plt.plot(nums, nums, color="olive", linewidth=4, linestyle="dashed")
plt.plot(nums, nums*nums, color="#ff6b6b", linewidth=4,
linestyle="dotted")
plt.plot(nums, nums**3, c="#ff9f43", linewidth=4, linestyle="-.")
```

```
[<matplotlib.lines.Line2D at 0x7f17623bf390>]
```




```
plt.plot(nums, nums, color="olive", marker="*", markersize=20,
markerfacecolor="#ff9f43")
```

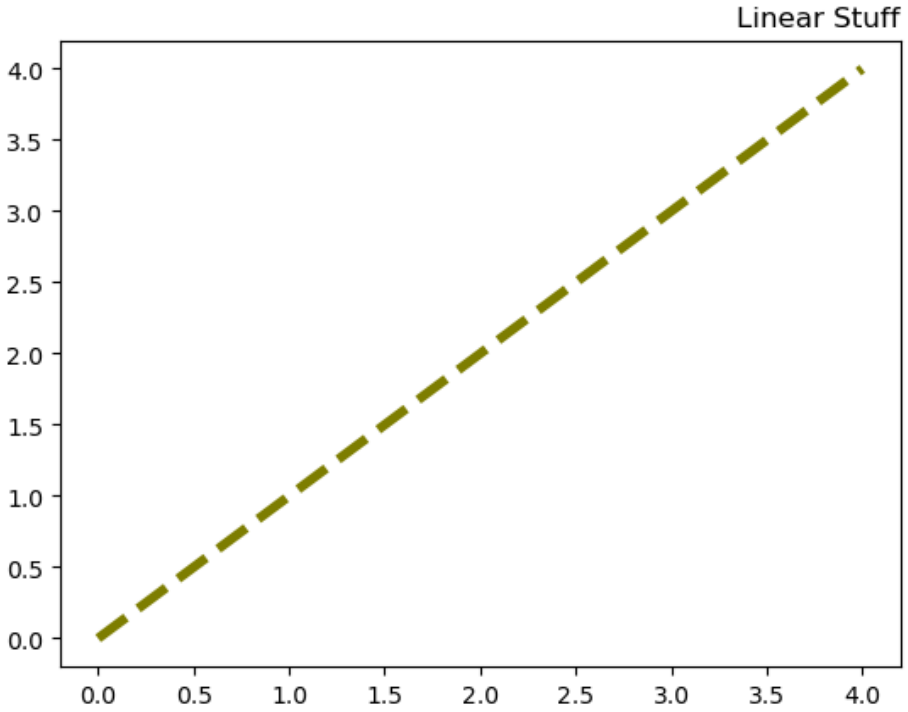
```
[<matplotlib.lines.Line2D at 0x7f1762442fd0>]
```




```
salaries=[55000,65000,72000,90000,115000,150000]
ages = [20,25,30,32,40,45]
plt.plot(ages, salaries)
```

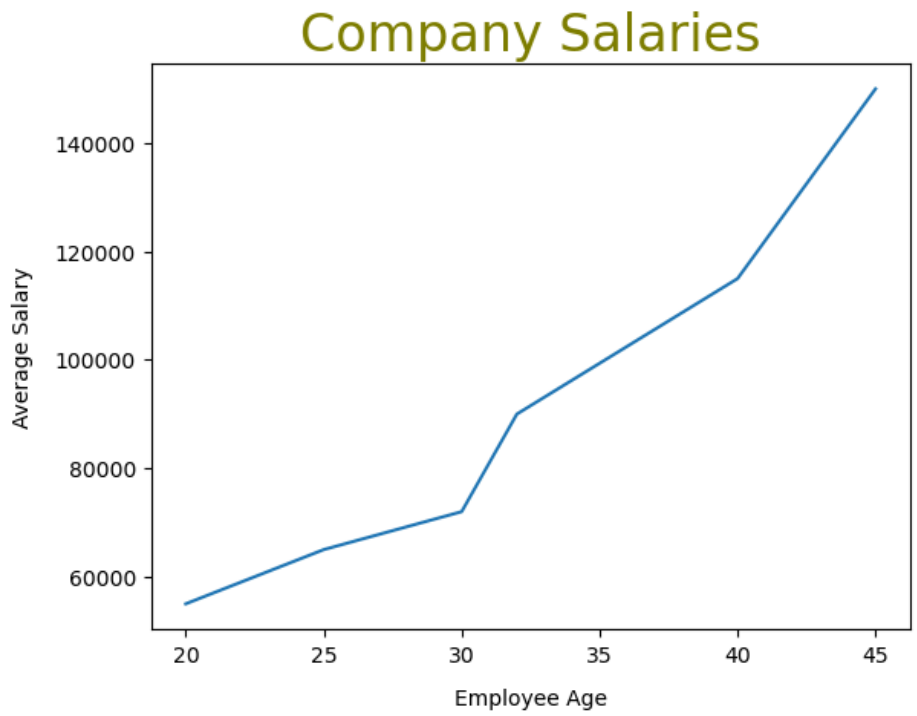
```
plt.title("Company Salaries")
plt.figure()
plt.plot(nums, nums, color="olive", linewidth=4, linestyle="dashed")
plt.title("Linear Stuff", loc="right")
```

```
Text(1.0, 1.0, 'Linear Stuff')
```



```
salaries=[55000,65000,72000,90000,115000,150000]
ages = [20,25,30,32,40,45]
plt.plot(ages, salaries)
plt.title("Company Salaries", fontsize=24, color="olive")
plt.xlabel("Employee Age", labelpad=10)
plt.ylabel("Average Salary", labelpad=10 )
```

```
Text(0, 0.5, 'Average Salary')
```

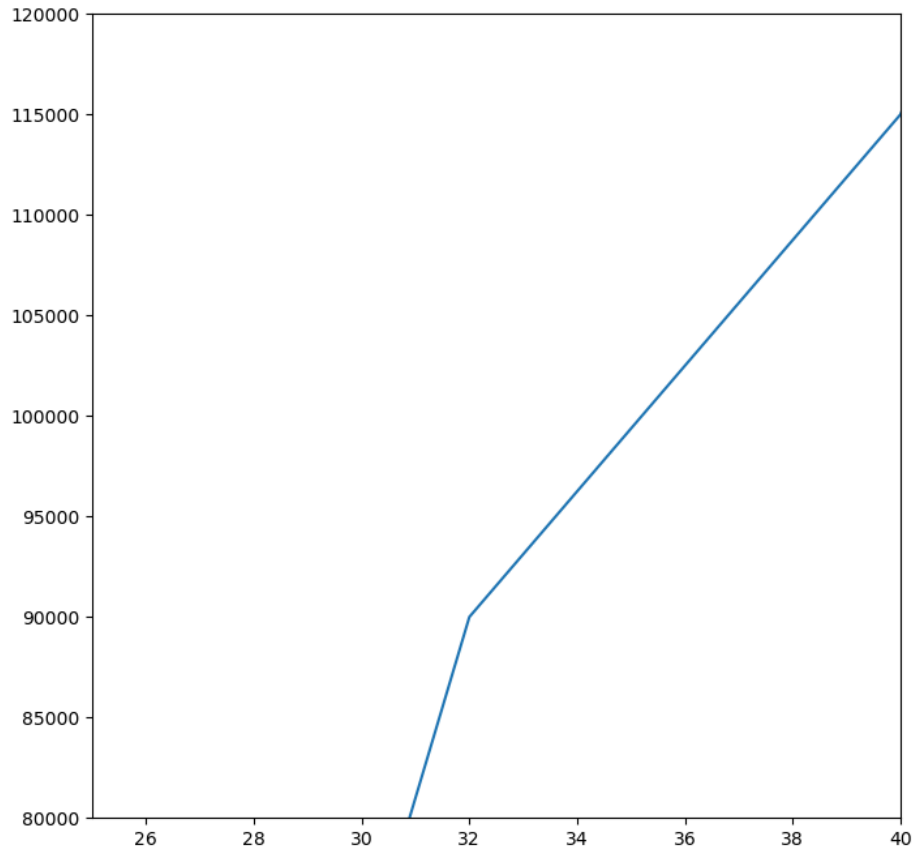



```
plt.figure(figsize=(8,8))
salaries=[55000,65000,72000,90000,115000,150000]
ages = [20,25,30,32,40,45]
plt.plot(ages, salaries)
plt.title("Company Salaries", fontsize=24, color="olive")
plt.xlabel("Employee Age", labelpad=10)
plt.ylabel("Average Salary", labelpad=10 )
plt.xticks([20,25,30,35,40,45])
plt.yticks([60000,80000,100000, 120000, 140000], labels=["60k", "80k",
"100k", "120k","140k"])
```

```
([<matplotlib.axis.YTick at 0x7f1762221950>,
  <matplotlib.axis.YTick at 0x7f1762261d10>,
  <matplotlib.axis.YTick at 0x7f1762262490>,
  <matplotlib.axis.YTick at 0x7f1762262c10>,
  <matplotlib.axis.YTick at 0x7f1762263390>],
 [Text(0, 60000, '60k'),
  Text(0, 80000, '80k'),
  Text(0, 100000, '100k'),
  Text(0, 120000, '120k'),
  Text(0, 140000, '140k')])
```

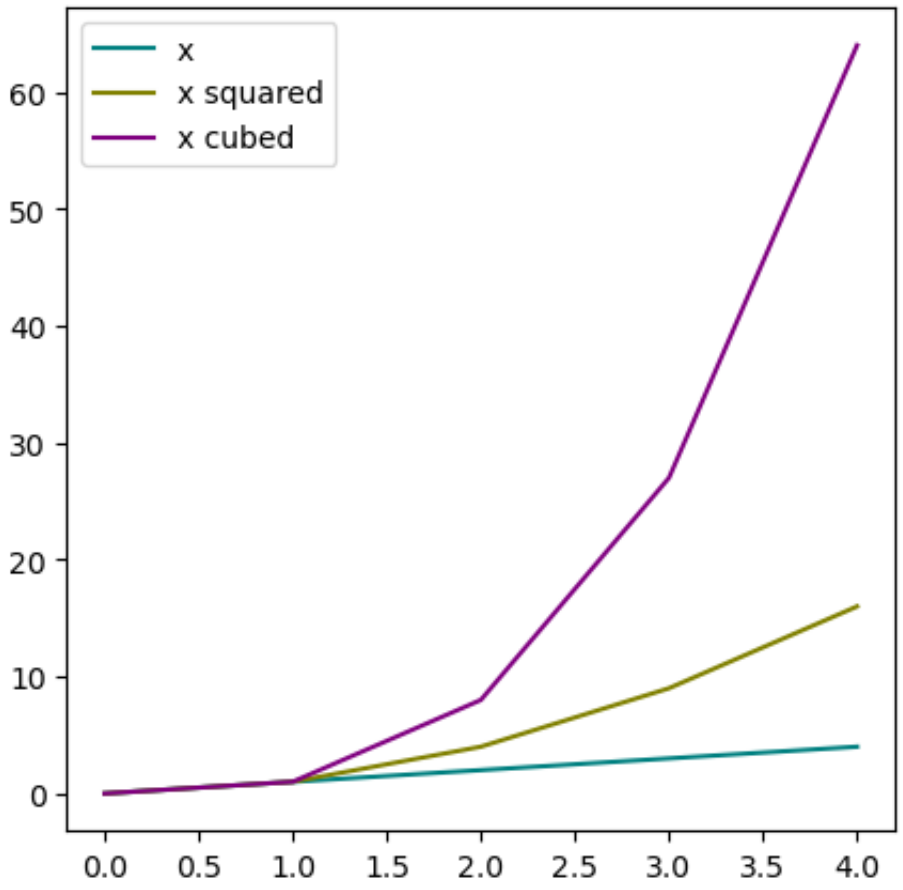

```
plt.figure(figsize=(8,8))
salaries=[55000,65000,72000,90000,115000,150000]
ages = [20,25,30,32,40,45]
plt.plot(ages, salaries)
plt.xlim(25,40)
plt.ylim(80000, 120000)
```

```
(80000.0, 120000.0)
```



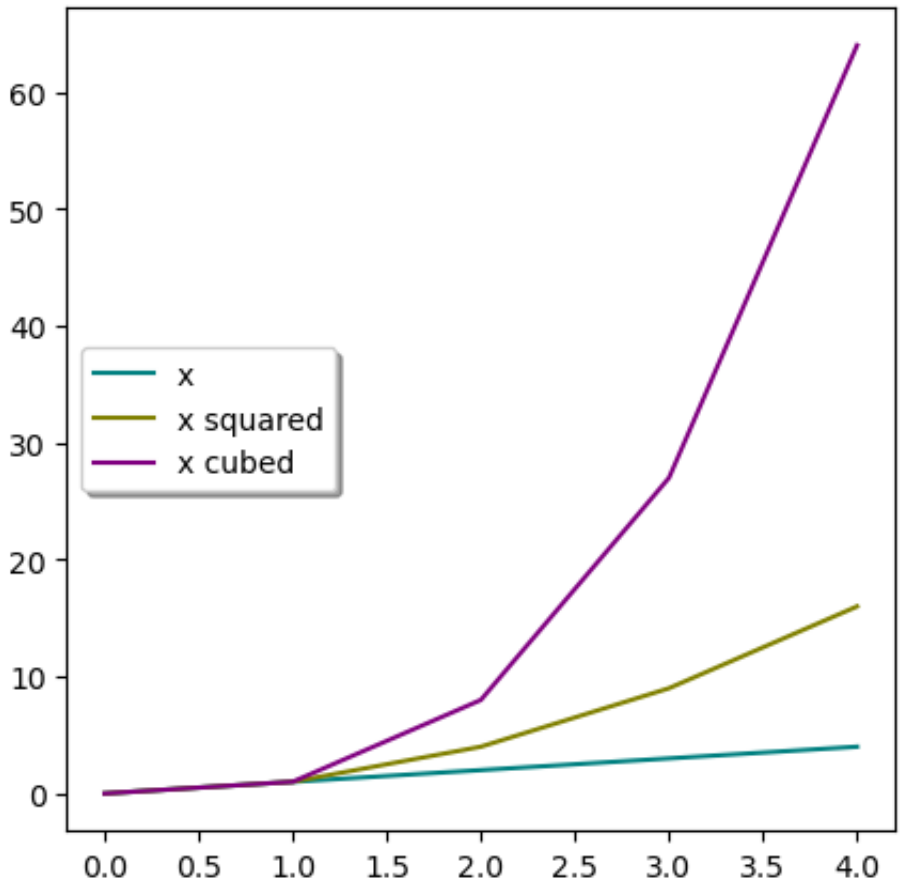

```
plt.figure(figsize=(5,5))
plt.plot(nums, color="teal", label="x")
plt.plot(nums**2, color="olive", label="x squared")
plt.plot(nums**3, color="purple",label="x cubed")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f1762bc9400>
```




```
plt.figure(figsize=(5,5))
plt.plot(nums, color="teal", label="x")
plt.plot(nums**2, color="olive", label="x squared")
plt.plot(nums**3, color="purple",label="x cubed")
plt.legend(loc="center left", shadow=True, frameon=True,
facecolor="white")
```

```
<matplotlib.legend.Legend at 0x7f176229f110>
```

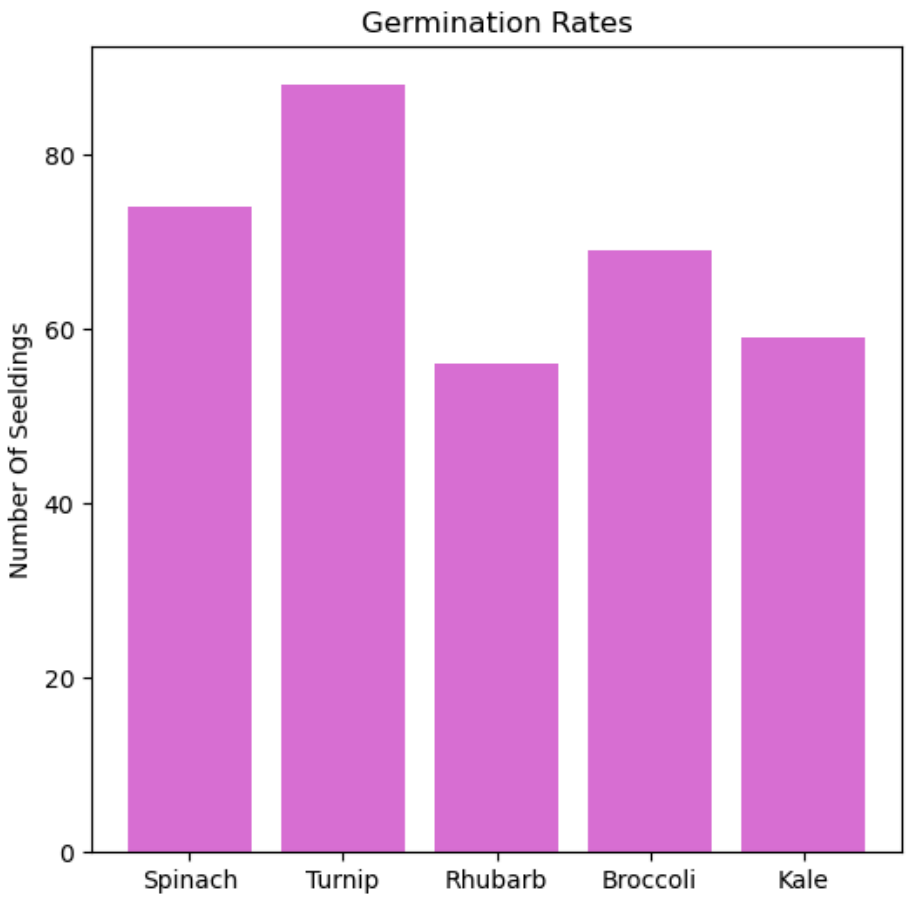


2.b.iii. *Bar Plots:*

```
plants = ['Spinach', 'Turnip', 'Rhubarb', 'Broccoli', 'Kale']
died = [10,25,5,30,21]
germinated = [74, 88, 56,69,59]
```

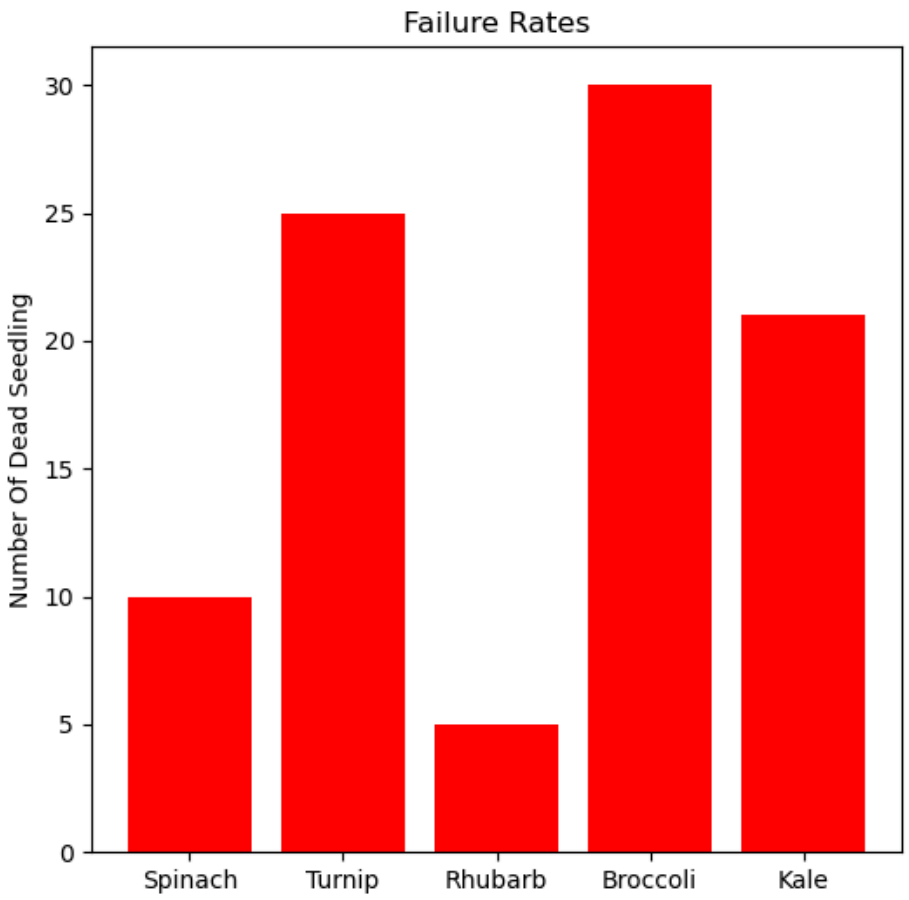
```
plt.figure(figsize=(6,6))
plt.bar(plants, germinated, color="orchid")
plt.title("Germination Rates")
plt.ylabel("Number Of Seeldings")
```

```
Text(0, 0.5, 'Number Of Seeldings')
```



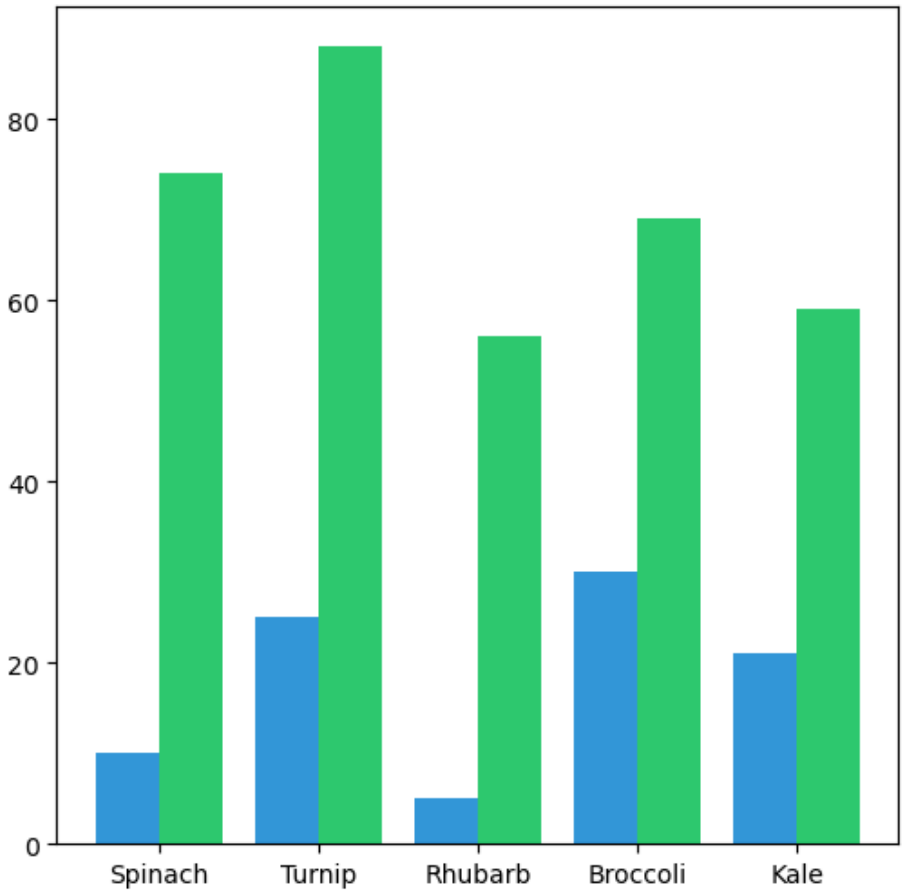

```
plt.figure(figsize=(6,6))
plt.bar(plants, died, color="red")
plt.title("Failure Rates")
plt.ylabel("Number Of Dead Seedling")
```

```
Text(0, 0.5, 'Number Of Dead Seedling')
```



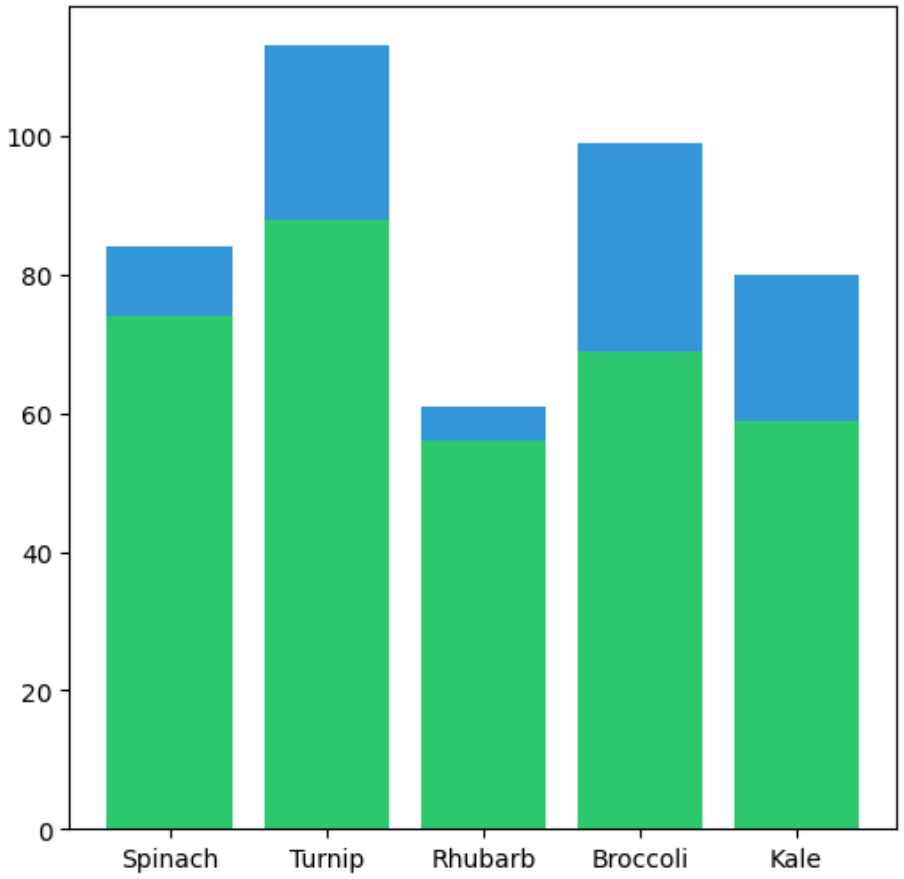

```
plt.figure(figsize=(6,6))
plt.bar(plants, died, color="#3498db")
plt.bar(plants, germinated, width=0.4, color="#2ecc71", align="edge")
```

```
<BarContainer object of 5 artists>
```

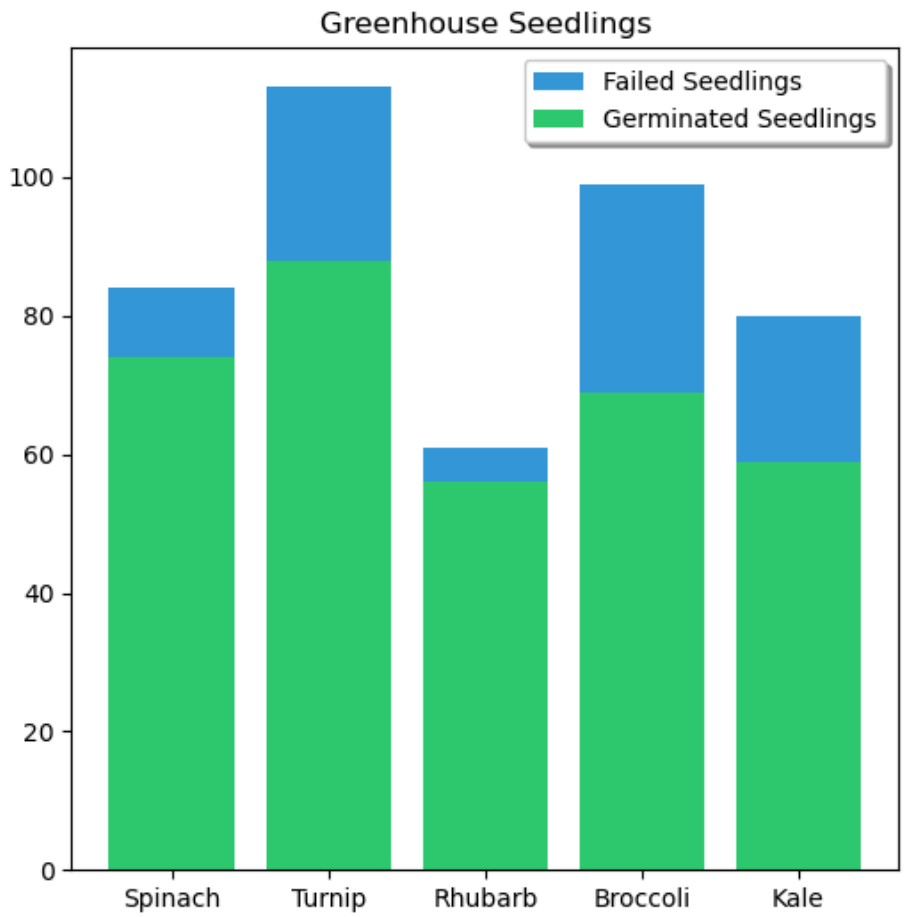



```
plt.figure(figsize=(6,6))
plt.bar(plants, died, color="#3498db", bottom=germinated )
plt.bar(plants, germinated, color="#2ecc71")
```

```
<BarContainer object of 5 artists>
```

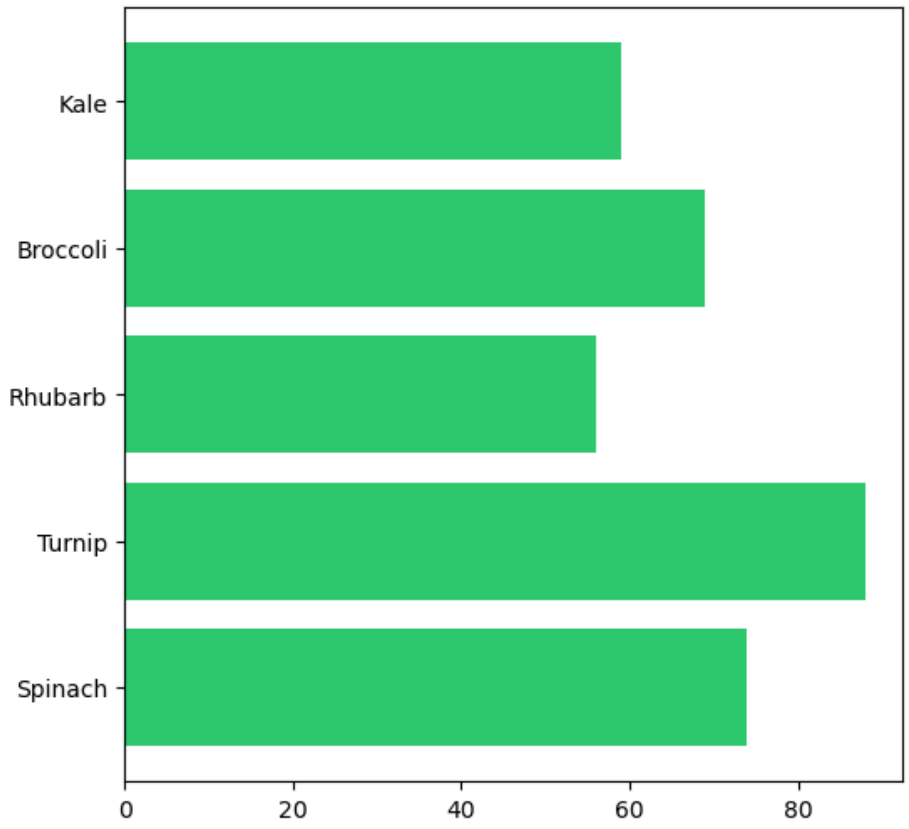



```
plt.figure(figsize=(6,6))
plt.bar(plants, died, color="#3498db", bottom=germinated,label="Failed
Seedlings")
plt.bar(plants, germinated, color="#2ecc71", label="Germinated
Seedlings")
plt.legend(shadow=True, frameon=True, facecolor="white")
plt.title("Greenhouse Seedlings")
plt.show()
```




```
plt.figure(figsize=(6,6))
plt.barh(plants, germinated, color="#2ecc71")
```

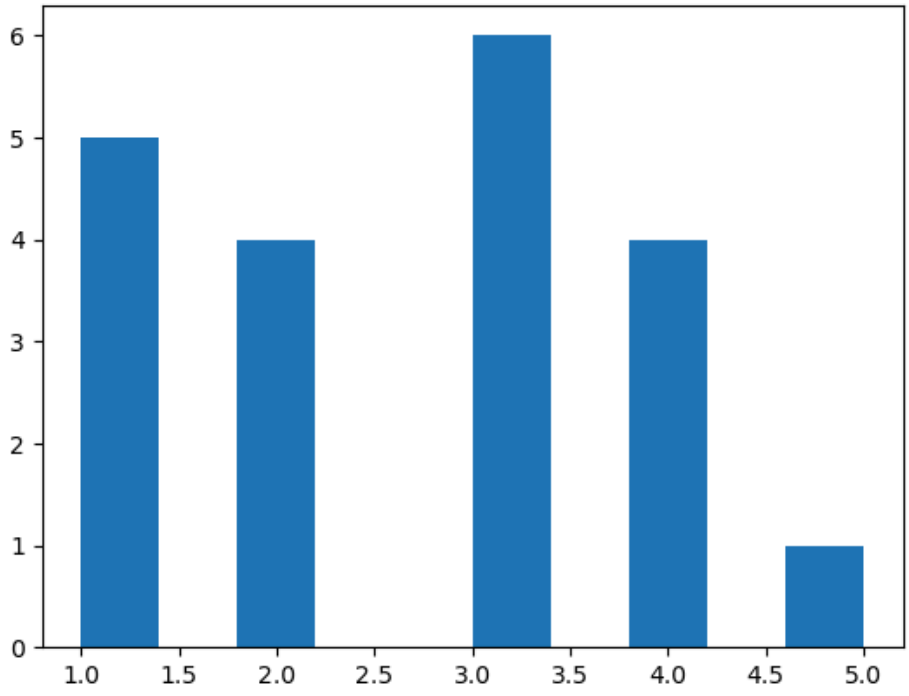
```
<BarContainer object of 5 artists>
```



2.b.iv. *Histograms:*

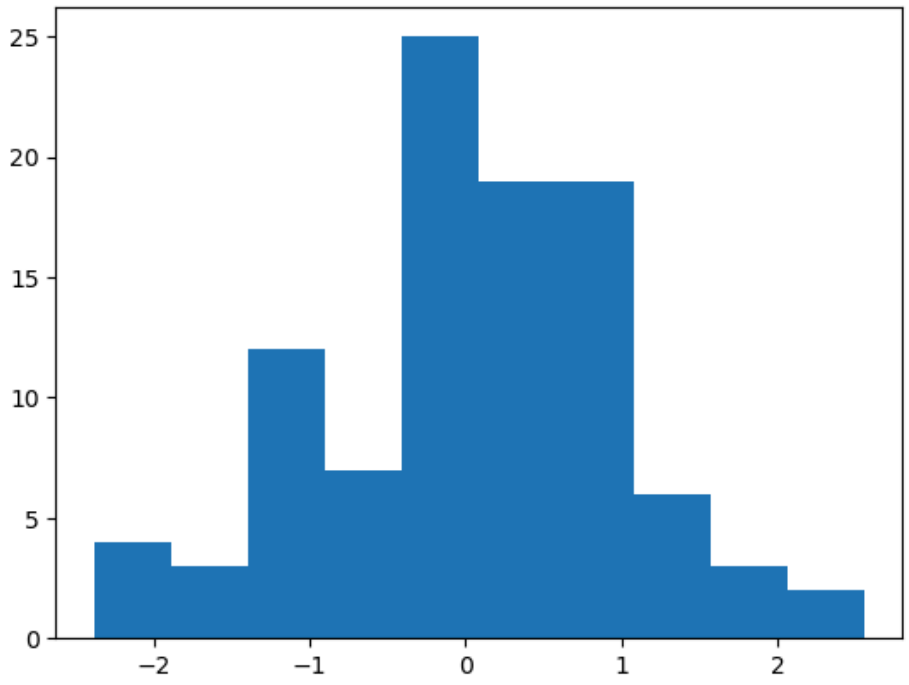
```
plt.hist([1,1,2,3,3,3,3,3,4,4,4,5,1,2,1,2,1,2,3,4])
```

```
(array([5., 0., 4., 0., 0., 6., 0., 4., 0., 1.]),
 array([1. , 1.4, 1.8, 2.2, 2.6, 3. , 3.4, 3.8, 4.2, 4.6, 5. ]),
 <BarContainer object of 10 artists>)
```



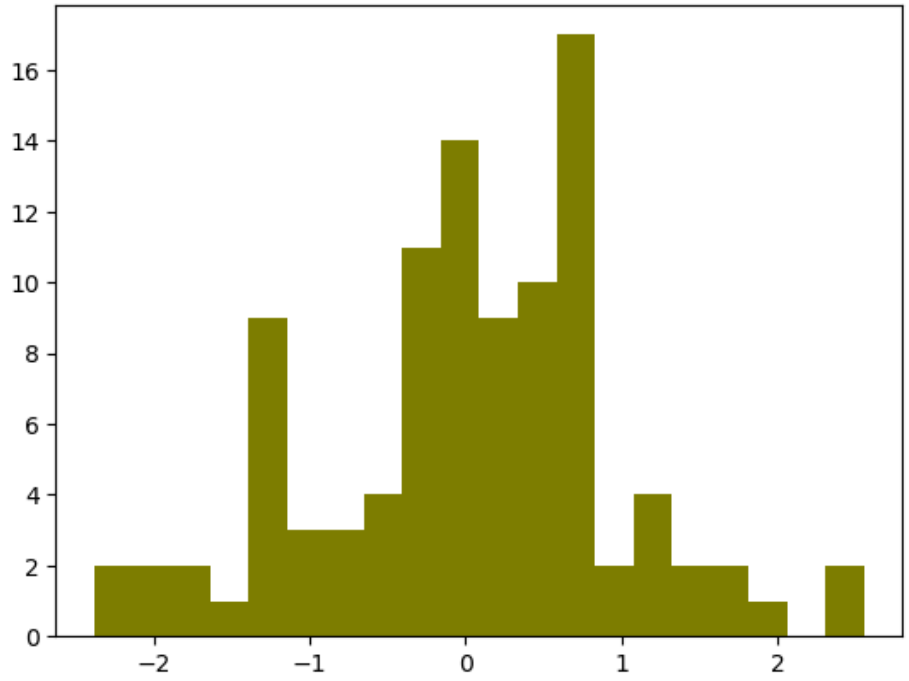

```
nums = np.random.randn(100)
plt.hist(nums)
```

```
(array([ 4.,  3., 12.,  7., 25., 19., 19.,  6.,  3.,  2.]),
 array([-2.38056732, -1.88677616, -1.392985  , -0.89919384, -0.40540268,
         0.08838849,  0.58217965,  1.07597081,  1.56976197,  2.06355313,
         2.55734429]),
 <BarContainer object of 10 artists>)
```



```
plt.hist(nums, bins=20, color="olive")
```

```
(array([ 2.,  2.,  2.,  1.,  9.,  3.,  3.,  4., 11., 14.,  9., 10., 17.,
         2.,  4.,  2.,  2.,  1.,  0.,  2.]),
 array([-2.38056732, -2.13367174, -1.88677616, -1.63988058, -1.392985  ,
        -1.14608942, -0.89919384, -0.65229826, -0.40540268, -0.15850709,
         0.08838849,  0.33528407,  0.58217965,  0.82907523,  1.07597081,
         1.32286639,  1.56976197,  1.81665755,  2.06355313,  2.31044871,
         2.55734429]),
 <BarContainer object of 20 artists>)
```

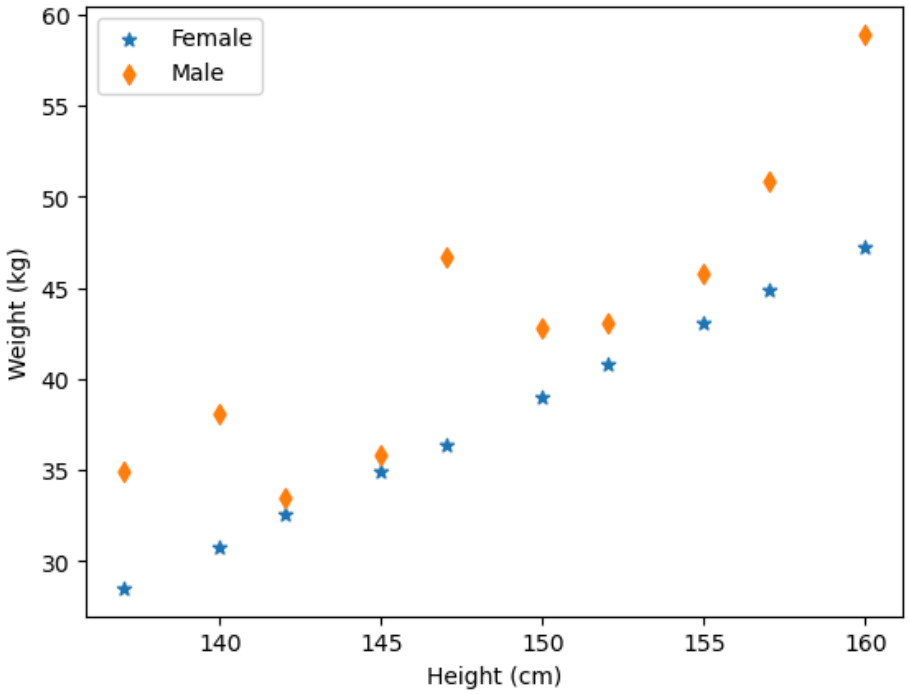


2.b.v. *Scatter Plots:*

```
heights = [137,140,142,145,147,150,152,155,157,160]
f_weights = [28.5,30.8,32.6,34.9,36.4,39,40.8,43.1,44.9,47.2]
m_weights = [34.9,38.1,33.5,35.8,46.7, 42.8,43.1,45.8,50.8,58.9]
```

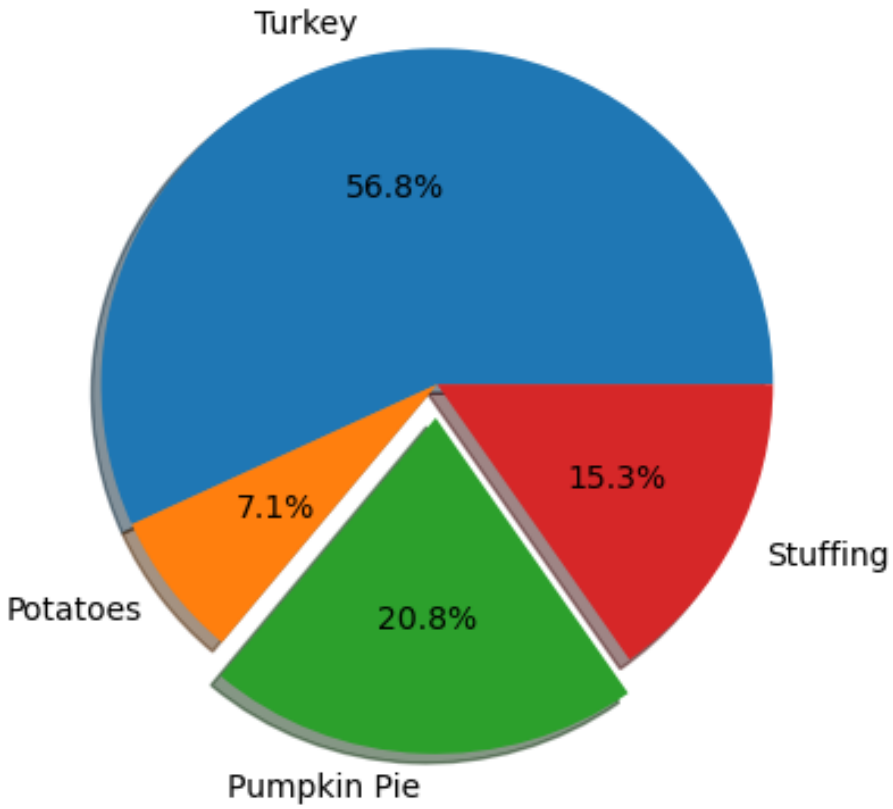
```
plt.scatter(heights, f_weights,  marker="*", label="Female")
plt.scatter(heights, m_weights,marker="d", label="Male")
plt.legend()
plt.xlabel("Height (cm)")
plt.ylabel("Weight (kg)")
```

```
Text(0, 0.5, 'Weight (kg)')
```



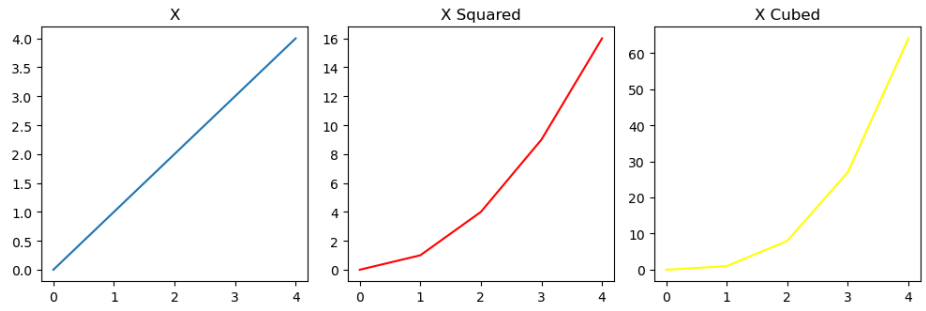
2.b.vi. *Pie Charts:*

```
labels = ["Turkey", "Potatoes", "Pumpkin Pie", "Stuffing"]
prices = [25.99,3.24,9.50, 6.99]

plt.pie(prices, labels=labels, autopct="%1.1f%%", shadow=True,
explode=(0,0,0.1,0))
plt.show()
```

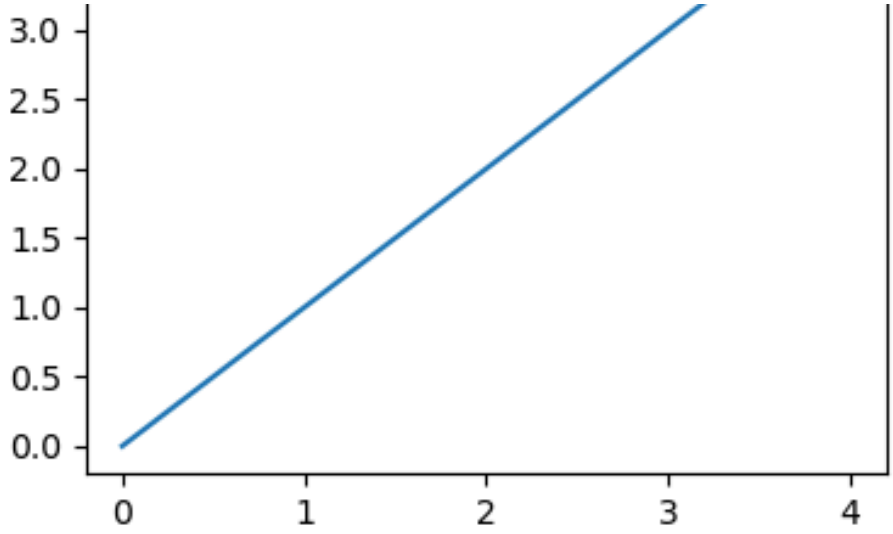


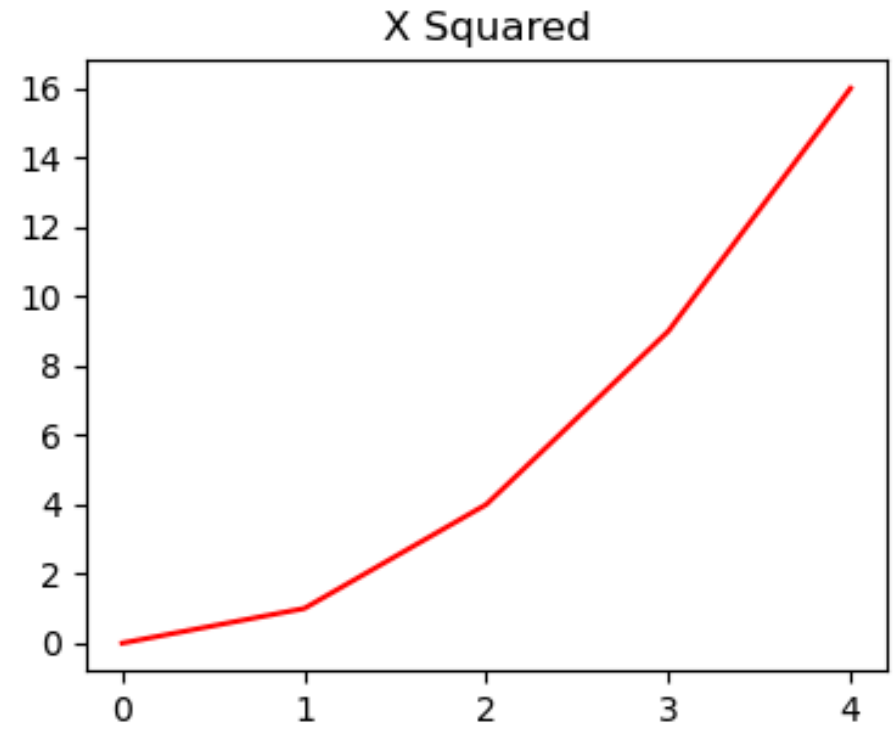
2.b.vii. *Subplots:*

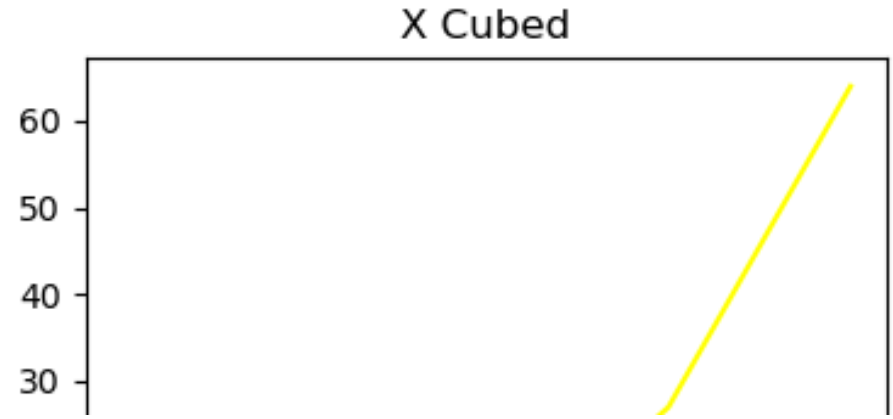
```
nums = np.arange(5)
plt.figure(figsize=(10,4))
plt.suptitle("Our First Subplot", fontsize=30)

plt.subplot(1,3,1)
plt.title("X")
plt.plot(nums, nums)

plt.subplot(1,3,2)
plt.plot(nums, nums**2, color="red")
plt.title("X Squared")


plt.subplot(1,3,3)
plt.plot(nums, nums**3, color="yellow")
plt.title("X Cubed")
plt.tight_layout()
plt.show()
```




```
nums = np.arange(5)
plt.figure(figsize=(4,10))
plt.suptitle("Our First Subplot", fontsize=30)

plt.subplot(3,1,1)
plt.title("X")
plt.plot(nums, nums)

plt.subplot(3,1,2)
plt.plot(nums, nums**2, color="red")
plt.title("X Squared")


plt.subplot(3,1,3)
plt.plot(nums, nums**3, color="yellow")
plt.title("X Cubed")
plt.tight_layout()
plt.show()
```

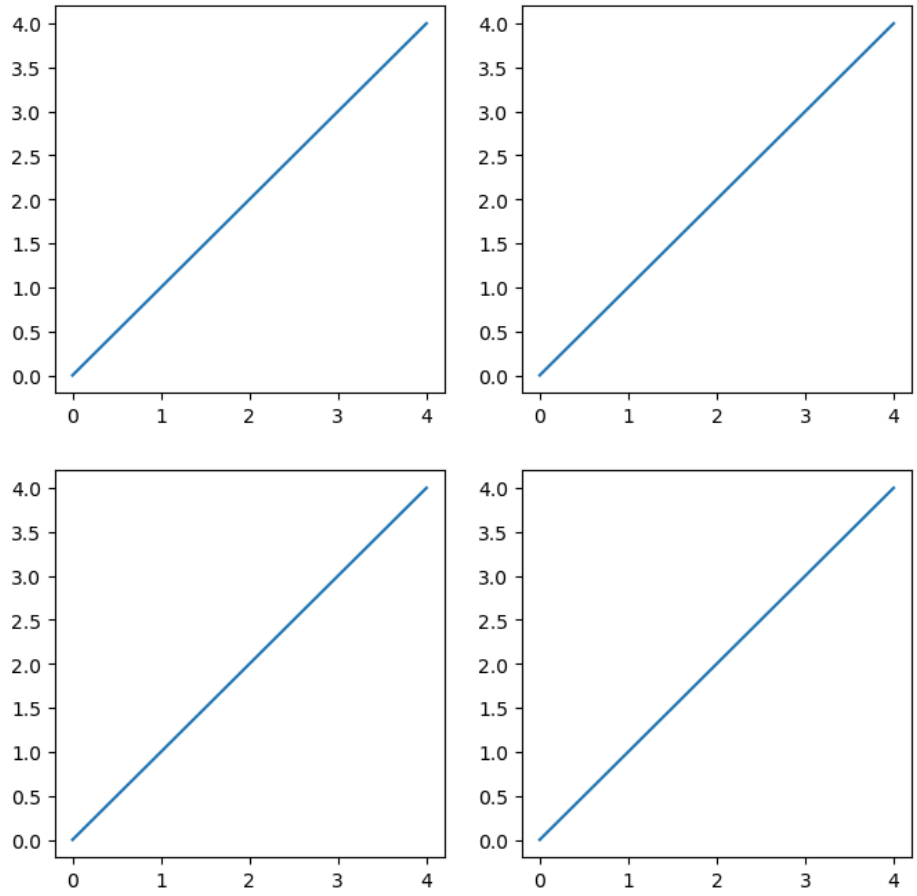

3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4


X Squared
16
14
12
10
8
6
4
2
0
0
1
2
3
4


X Cubed
60
50
40
30

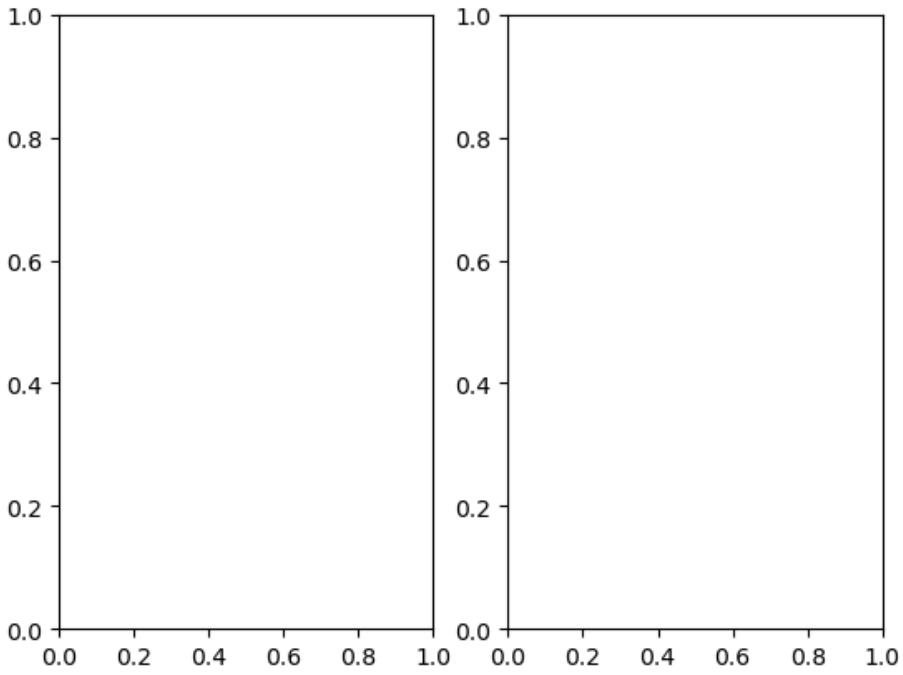
```
plt.figure(figsize=(8,8))
plt.subplot(2,2,1)
plt.plot(nums)

plt.subplot(2,2,2)
plt.plot(nums)


plt.subplot(2,2,3)
plt.plot(nums)


plt.subplot(2,2,4)
plt.plot(nums)
```

```
[<matplotlib.lines.Line2D at 0x7f1761846e90>]
```



2.b.viii. *The Object-Oriented Approach:*

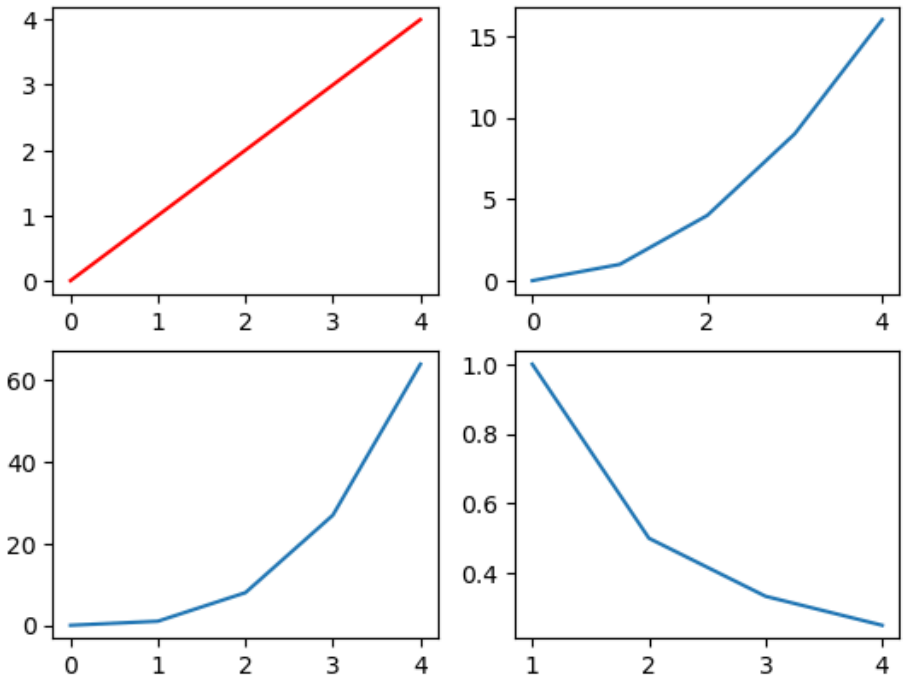
```
fig, axs = plt.subplots(1,2)
```




```
fig, axs = plt.subplots(2,2)
axs[0][0].plot(nums,color="red")
axs[0][1].plot(nums*nums)
axs[1][0].plot(nums**3)
axs[1][1].plot(1/nums)

axs[0][1].set_xticks([0,2,4])
```

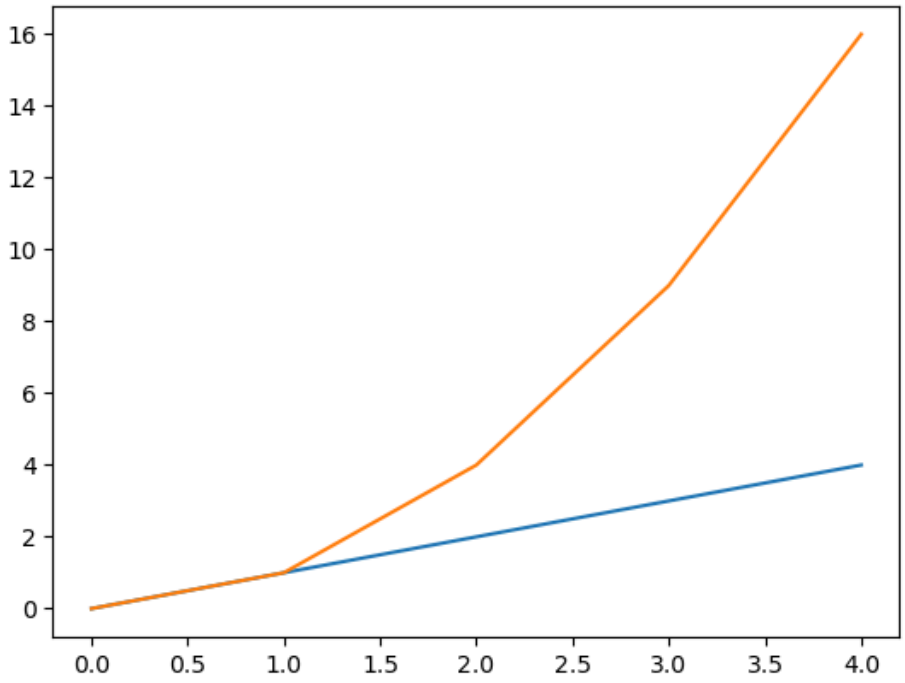
```
/tmp/ipykernel_11048/2433834221.py:5: RuntimeWarning: divide by zero
encountered in divide
  axs[1][1].plot(1/nums)
```

```
[<matplotlib.axis.XTick at 0x7f17613b8f50>,
 <matplotlib.axis.XTick at 0x7f176142d950>,
 <matplotlib.axis.XTick at 0x7f176142e0d0>]
```




```
fig, ax = plt.subplots()
ax.plot(nums)
ax.plot(nums*nums)
```

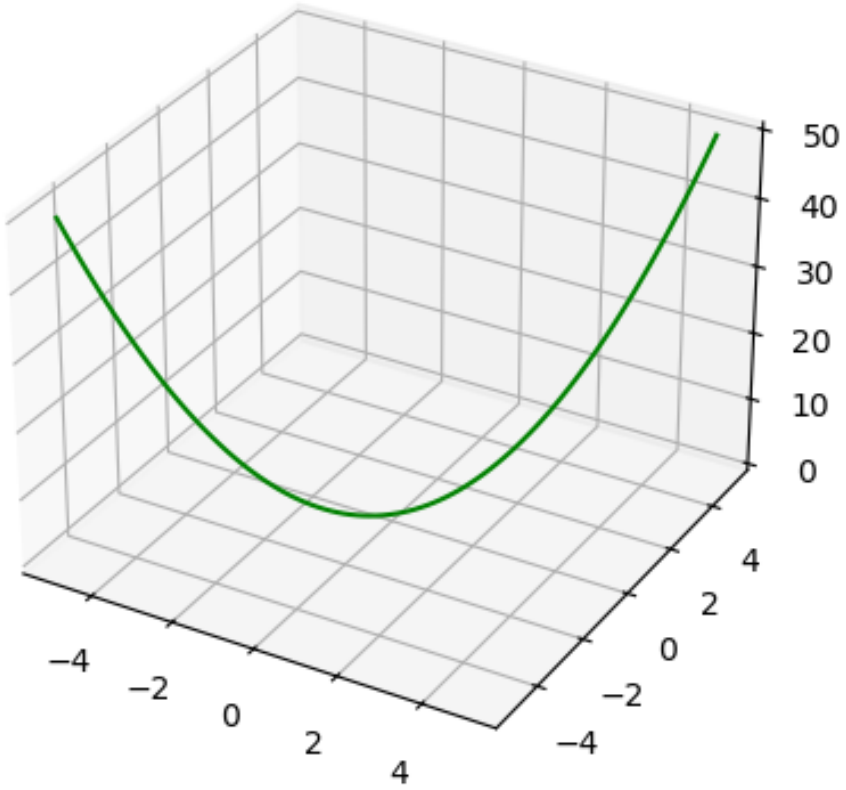
```
[<matplotlib.lines.Line2D at 0x7f17612f39d0>]
```



2.b.ix. *3D Plots:*

```
# For 3D plots, let's consider some equations
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
z = x**2 + y**2

ax = plt.figure().add_subplot(projection='3d')
ax.plot(x, y, z, 'green')
```

```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f17616c9090>]
```



2.c. Linear Equations

A linear equation is a mathematical statement that represents a straight line when graphed.

Then let's take input from user, in the following format,

```
Enter number of unknowns: 3
Enter row 1 in the format (ax + by + cz = d)
```

```
# User input (auto)
# N = int(input("Enter number of unknowns: "))

# arr = []
# for i in range(N):
#     arr.append(list(map(int, input().split())))

# User input (manual)
N = 3
arr = [[2, 1, -1, 8], [-3, -1, 2, -11], [-2, 1, 2, -3]]
```

```
# User input's output
print(N)

def print_arr(arr):
  for i in range(N):
    print(arr[i])

print_arr(arr)
```

```
3
[2, 1, -1, 8]
[-3, -1, 2, -11]
[-2, 1, 2, -3]
```

2.c.i. *Gaussian Elimination:*

```
import copy

def gaussian_elimination(N, arr):
  arr = copy.deepcopy(arr)
  for i in range(N):
    for j in range(N, -1, -1):
      arr[i][j] /= arr[i][0+i]

    for j in range(i+1, N):
      for k in range(N, -1, -1):
        arr[j][k] -= arr[i][k] * arr[j][0+i]

    print_arr(arr)
    print()

  solve = [0 for j in range(N)]
  for i in range(N-1, -1, -1):
    for j in range(N):
```

```
      solve[i] = arr[i][N]
      for k in range(i+1, N):
        solve[i] -= arr[i][k] * solve[k]
    solve[i] /= arr[i][i]

  print(solve)

gaussian_elimination(N, arr)
```

```
[1.0, 0.5, -0.5, 4.0]
[0.0, 0.5, 0.5, 1.0]
[0.0, 2.0, 1.0, 5.0]

[1.0, 0.5, -0.5, 4.0]
[0.0, 1.0, 1.0, 2.0]
[0.0, 0.0, -1.0, 1.0]

[1.0, 0.5, -0.5, 4.0]
[0.0, 1.0, 1.0, 2.0]
[0.0, 0.0, 1.0, -1.0]

[2.0, 3.0, -1.0]
```

2.c.ii. *Gauss Jordan:*

```
import copy
arr = copy.deepcopy(arr)

def gauss_jordan(arr, N):
    for i in range(N):
        for j in range(N):
            if i != j:
                p = arr[j][i] / arr[i][i]
                for k in range(N+1):
                    arr[j][k] -= arr[i][k] * p
        print_arr(arr)
        print()

    for i in range(N):
        print(arr[i][3] / arr[i][i])

gauss_jordan(arr, N)
```

```
[2, 1, -1, 8]
[0.0, 0.5, 0.5, 1.0]
[0.0, 2.0, 1.0, 5.0]

[2.0, 0.0, -2.0, 6.0]
[0.0, 0.5, 0.5, 1.0]
[0.0, 0.0, -1.0, 1.0]

[2.0, 0.0, 0.0, 4.0]
[0.0, 0.5, 0.0, 1.5]
[0.0, 0.0, -1.0, 1.0]
```

```
2.0
3.0
-1.0
```

2.c.iii. *Cramers' rule:*

Another one easiest way is to use cramer's rule. It's not so dynamic but it works for small systems of equations. The idea is to express the solution in terms of determinants.

```
import numpy as np

main_array = np.array(arr)
D = np.array(main_array[:,:-1])

last_col  = np.array(main_array[:,-1])

D1 = np.array([last_col, main_array[:,1], main_array[:,2]])
D2 = np.array([main_array[:,0], last_col, main_array[:,2]])
D3 = np.array([main_array[:,0], main_array[:,1], last_col])

# We can also define our own det function for n specific variable easily
like,
# def det(arr):
#     a =   arr[0][0] * ( arr[1][1] * arr[2][2] - arr[1][2] * arr[2]
[1] )
#     b = - arr[1][0] * ( arr[0][1] * arr[2][2] - arr[0][2] * arr[2]
[1] )
#     c =   arr[2][0] * ( arr[0][1] * arr[1][2] - arr[0][2] * arr[1]
[1] )
#     return a + b + c

D_det = np.linalg.det(D)
D1_det = np.linalg.det(D1)
D2_det = np.linalg.det(D2)
D3_det = np.linalg.det(D3)

print("x = ", D1_det / D_det)
print("y = ", D2_det / D_det)
print("z = ", D3_det / D_det)
```

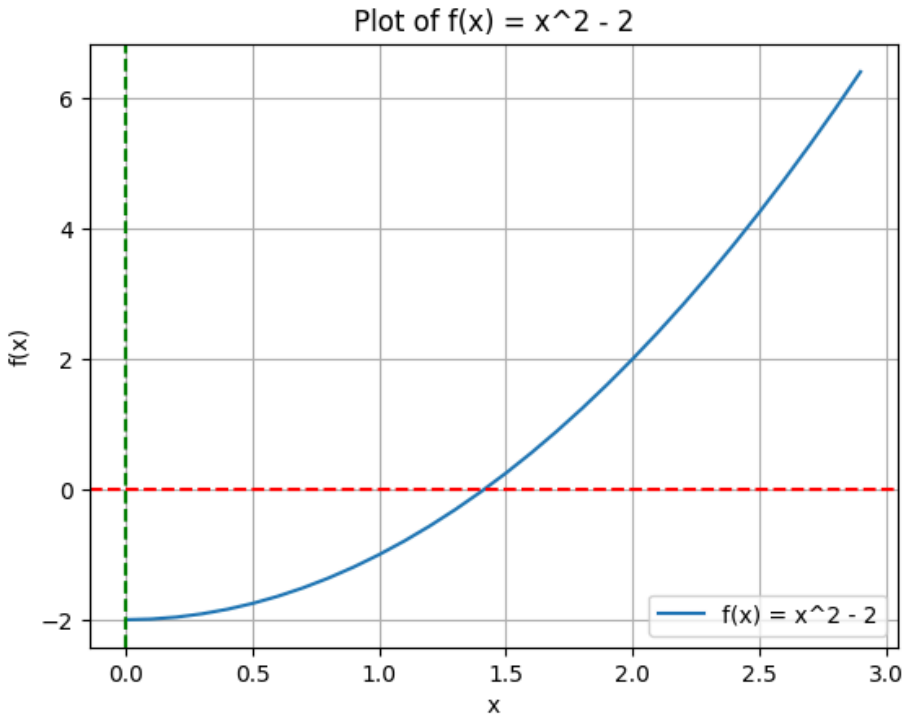
```
x =  2.0
y =  2.9999999999999996
z =  -0.9999999999999998
```

2.d. Root Finding

Root finding refers to the process of finding solutions to equations of the form f(x) = 0. This is a fundamental problem in numerical analysis and has various applications in science and engineering.

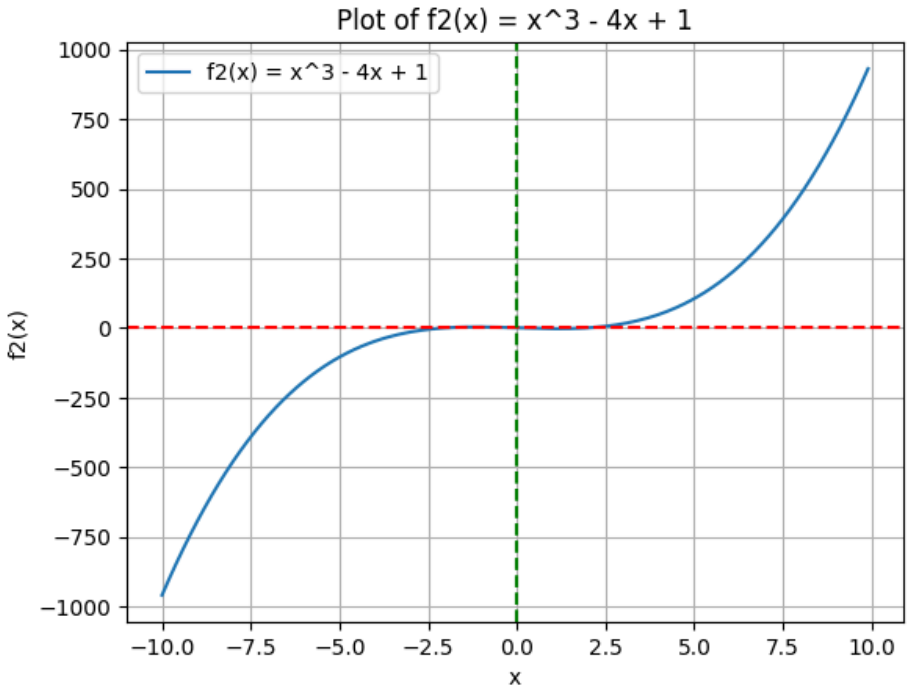
```
# First let's import necessary libs
import matplotlib.pyplot as plt
import numpy as np
import math

# Our first function
def f(x):
  return x**2 - 2

# Let's plot the function
x = np.arange(0, 3, 0.1)
plt.plot(x, f(x), label='f(x) = x^2 - 2')

plt.axhline(0, color='red', linestyle='--')
plt.axvline(0, color='green', linestyle='--')

plt.xlabel('x')
plt.ylabel('f(x)')

plt.title('Plot of f(x) = x^2 - 2')
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef91f83b60>
```



```
# Our second function
def f2(x):
  return x**3 - 4 * x + 1

# Let's plot the function
x = np.arange(-10, 10, 0.1)
plt.plot(x, f2(x), label='f2(x) = x^3 - 4x + 1')

plt.axhline(0, color='red', linestyle='--')
plt.axvline(0, color='green', linestyle='--')

plt.xlabel('x')
plt.ylabel('f2(x)')

plt.title('Plot of f2(x) = x^3 - 4x + 1')
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8fc59090>
```



2.d.i. *Bisection Method:*

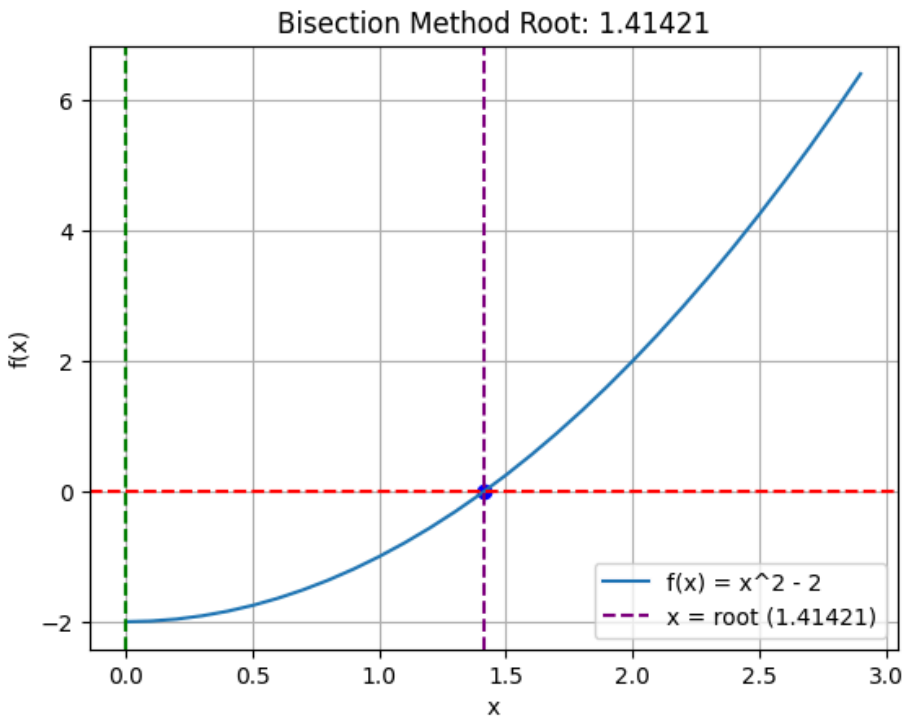
Bisection method finds the root of a function f in the interval [a, b].

```
Parameters:
- f : function
  The function for which we want to find the root.
- a : float
  The start of the interval.
- b : float
  The end of the interval.
```

```
- tol : float
  The tolerance for convergence.

Returns:
- float
  The approximate root of the function.
```

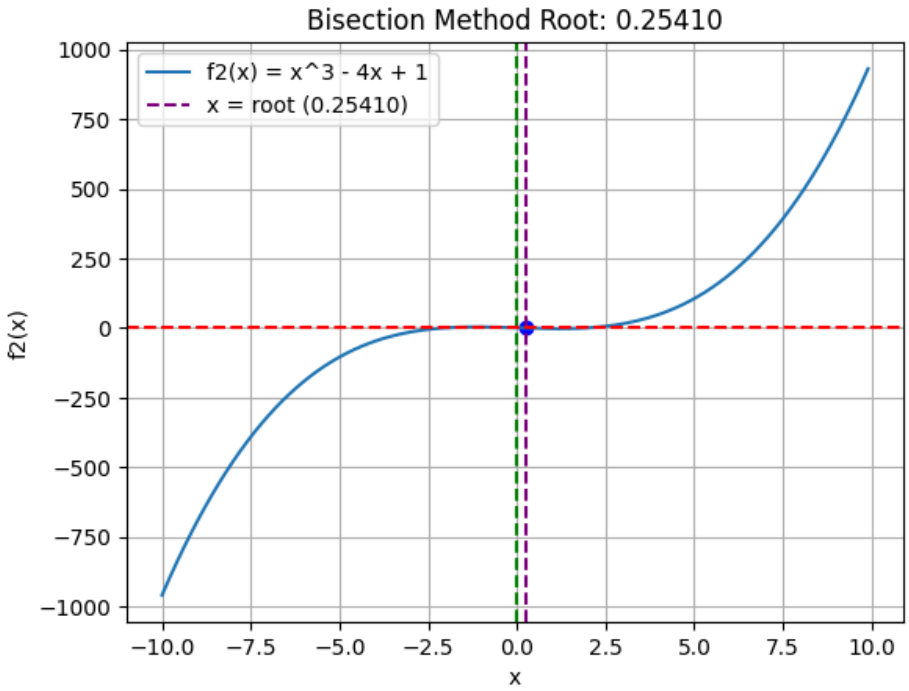
```
def bisection_method(f, a, b, tol=1e-5):
    if f(a) * f(b) >= 0:
        raise ValueError("f(a) and f(b) must have opposite signs.")

    mid = (a + b) / 2.0

    if abs(f(mid)) < tol:
        return mid
    elif f(a) * f(mid) < 0:
        return bisection_method(f, a, mid, tol)
    else:
        return bisection_method(f, mid, b, tol)

root = bisection_method(f, 0, 10)

# Let's plot the result
x = np.arange(0, 3, 0.1)

plt.plot(x, f(x), label='f(x) = x^2 - 2')
plt.scatter(root, f(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')

plt.axvline(0, color='green', linestyle='--')
plt.axhline(0, color='red', linestyle='--')

plt.xlabel('x')
plt.ylabel('f(x)')
plt.title(f"Bisection Method Root: {root:.5f}")
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8fd27d90>
```




```
root = bisection_method(f2, -1, 1)

# Let's plot the result
x = np.arange(-10, 10, 0.1)

plt.plot(x, f2(x), label='f2(x) = x^3 - 4x + 1')
plt.scatter(root, f2(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')

plt.axvline(0, color='green', linestyle='--')
plt.axhline(0, color='red', linestyle='--')

plt.xlabel('x')
plt.ylabel('f2(x)')
plt.title(f"Bisection Method Root: {root:.5f}")
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8fdb3750>
```

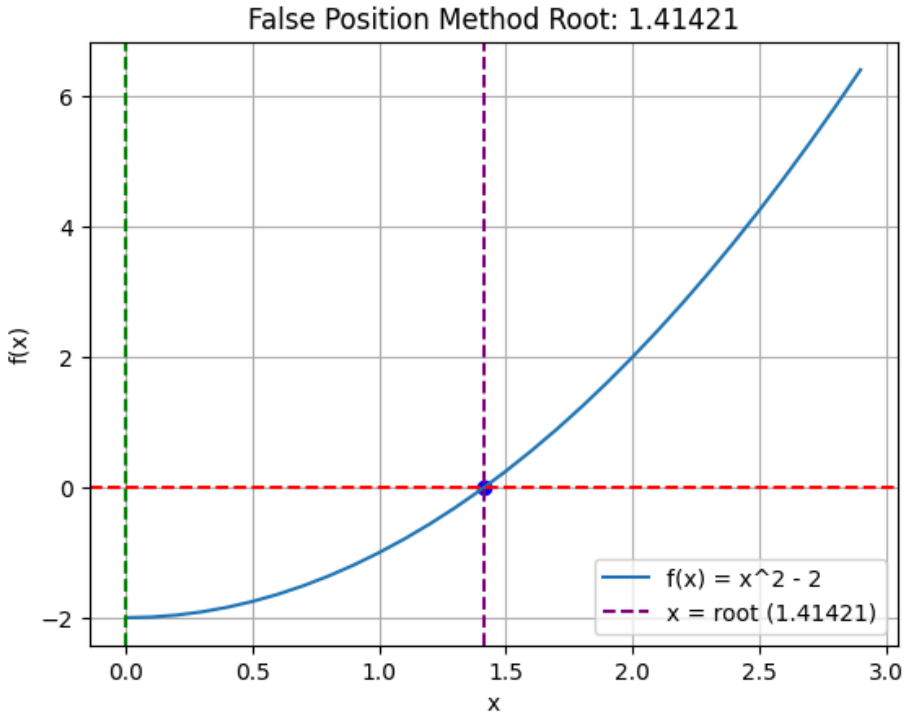


2.d.ii. *False Position Method:*

```
def false_position_method(f, a, b, tol=1e-5):
    if f(a) * f(b) >= 0:
        raise ValueError("f(a) and f(b) must have opposite signs.")

    c = a - (f(a) * (b - a)) / (f(b) - f(a))

    if abs(f(c)) < tol:
        return c
    elif f(a) * f(c) < 0:
        return false_position_method(f, a, c, tol)
    else:
        return false_position_method(f, c, b, tol)
root = false_position_method(f, 0, 10)

# Let's plot the result
x = np.arange(0, 3, 0.1)

plt.plot(x, f(x), label='f(x) = x^2 - 2')
plt.scatter(root, f(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')

plt.axvline(0, color='green', linestyle='--')
plt.axhline(0, color='red', linestyle='--')

plt.xlabel('x')
plt.ylabel('f(x)')
```

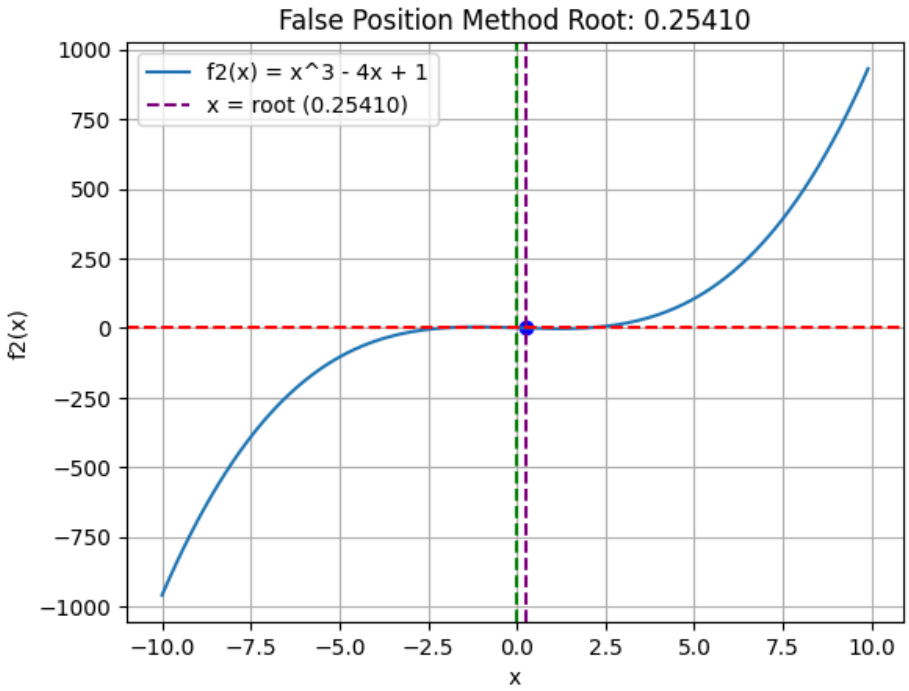
```
plt.title(f"False Position Method Root: {root:.5f}")
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8fa49d10>
```




```
root = false_position_method(f2, -1, 1)

# Let's plot the result
x = np.arange(-10, 10, 0.1)

plt.plot(x, f2(x), label='f2(x) = x^3 - 4x + 1')
plt.scatter(root, f2(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')

plt.axvline(0, color='green', linestyle='--')
plt.axhline(0, color='red', linestyle='--')

plt.xlabel('x')
plt.ylabel('f2(x)')
plt.title(f"False Position Method Root: {root:.5f}")
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8f8dd590>
```



2.d.iii. *Iteration Method:*

The iteration method is a common approach to find roots of equations. It involves rearranging the equation into the form x = g(x) and then iteratively applying g to an initial guess until convergence.

Let's consider the equation $x^3 - 4x + 1 = 0$. We can rearrange it to $x = \frac{x^3+1}{4}$.

```
def f2(x):
    return x**3 - 4*x + 1

def g2(x):
    return (x**3 + 1) / 4
```

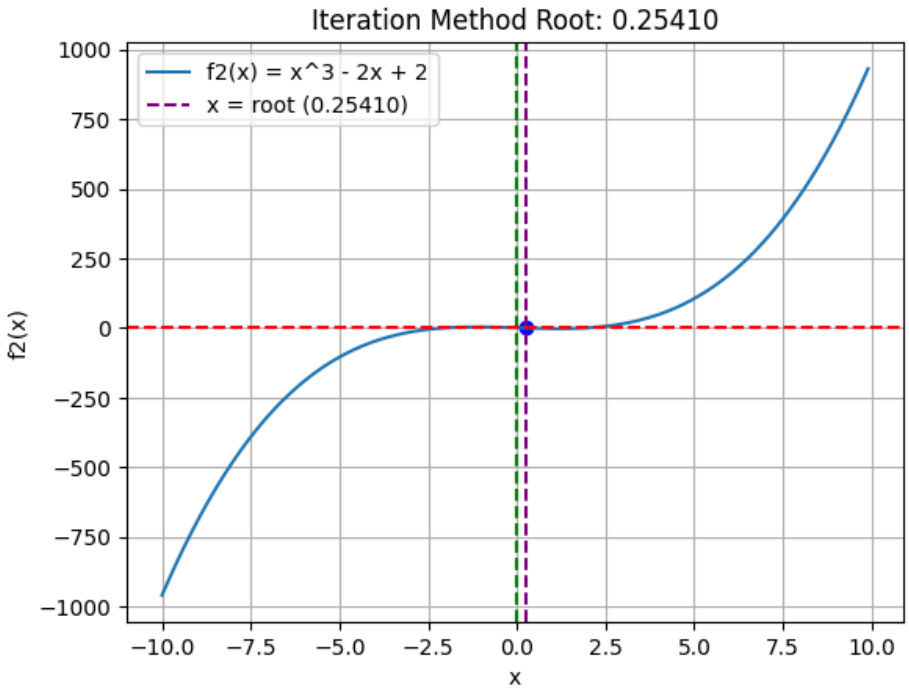
Let's implement the iteration methdod for this case,

```
def iteration_method(g, x, tol=1e-5, max_iter=100):
    for i in range(max_iter):
        x_new = g(x)
        if abs(x_new - x) < tol:
            return x_new
        x = x_new
    raise ValueError(
        "Iteration did not converge within the maximum number of
iterations."
    )
```

And finally plotting time,

```
root = iteration_method(g2, 1)

# Let's plot the result
x = np.arange(-10, 10, 0.1)

plt.plot(x, f2(x), label='f2(x) = x^3 - 2x + 2')
plt.scatter(root, f2(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')

plt.axvline(0, color='green', linestyle='--')
plt.axhline(0, color='red', linestyle='--')

plt.xlabel('x')
plt.ylabel('f2(x)')
plt.title(f"Iteration Method Root: {root:.5f}")
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef8f957b10>
```



2.d.iv. *Newton-Raphson Method:*

The Newton-Raphson method is an efficient root-finding algorithm that uses the derivative of a function to find its roots. The method starts with an initial guess and iteratively refines it using the formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1}$$

where:

- x_n - is the current guess,
- $f(x_n)$ - is the value of the function at x_n,
- $f'(x_n)$ - is the value of the derivative of the function at x_n.

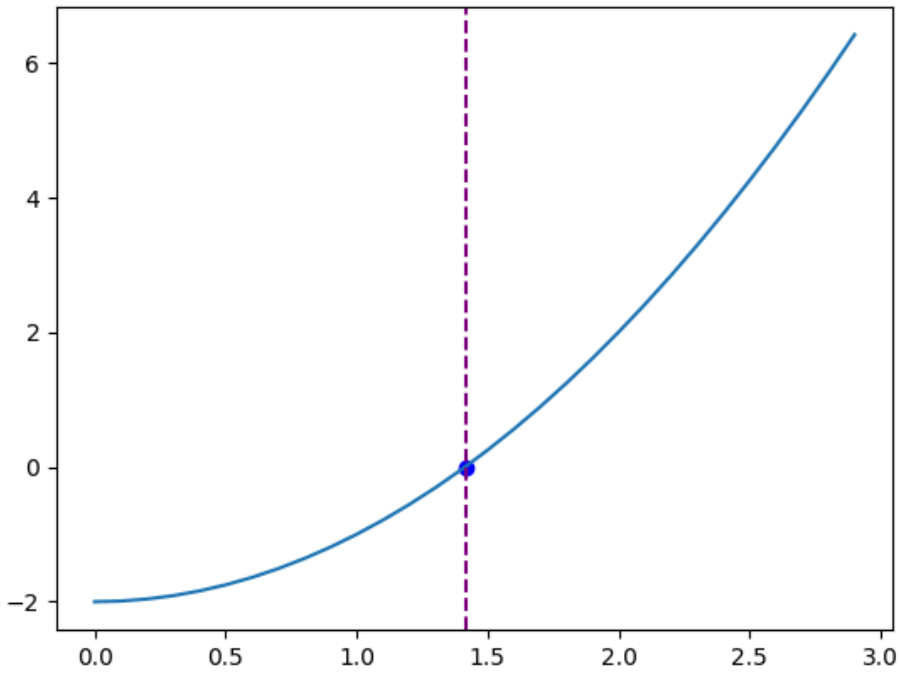
```
# Our function
def f(x):
    return x**2 - 2

def df(x):
    return 2*x

# Newton-Raphson Method
def newton_raphson_method(f, df, x0, tol=1e-5, max_iter=100):
    x = x0
    for i in range(max_iter):
        x_new = x - f(x) / df(x)
        if abs(x_new - x) < tol:
            return x_new
        x = x_new
    raise ValueError(
        "Newton-Raphson method did not converge within the maximum
number of iterations."
    )

root = newton_raphson_method(f, df, 1)
print(root)
```

```
1.4142135623746899
```

Let's plot it...

```
root = newton_raphson_method(f, df, 10)

# Let's plot the result
x = np.arange(0, 3, 0.1)
plt.plot(x, f(x), label='f(x) = x^2 - 2')
plt.scatter(root, f(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')
```

```
<matplotlib.lines.Line2D at 0x7fef8f83dd10>
```



2.d.v. *Secant Method:*

The secant method is a root-finding algorithm that uses two initial approximations to find the root of a function. Unlike the Newton-Raphson method, it does not require the computation of the derivative. Instead, it approximates the derivative using the two initial points.

The secant method uses the following formula to iteratively refine the root approximation:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \tag{2}$$

where:

- x_n is the current approximation,
- x_{n-1} is the previous approximation,
- $f(x_n)$ is the value of the function at x_n,
- $f(x_{n-1})$ is the value of the function at x_{n-1}.

```
def secant_method(f, x0, x1, tol=1e-7, max_iter=100):
    for i in range(max_iter):
        if abs(f(x1)) < tol:
            return x1
        if f(x1) == f(x0):  # Prevent division by zero
            print("Division by zero encountered in secant method.")
            return None
        # Secant method formula
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        x0, x1 = x1, x2
```

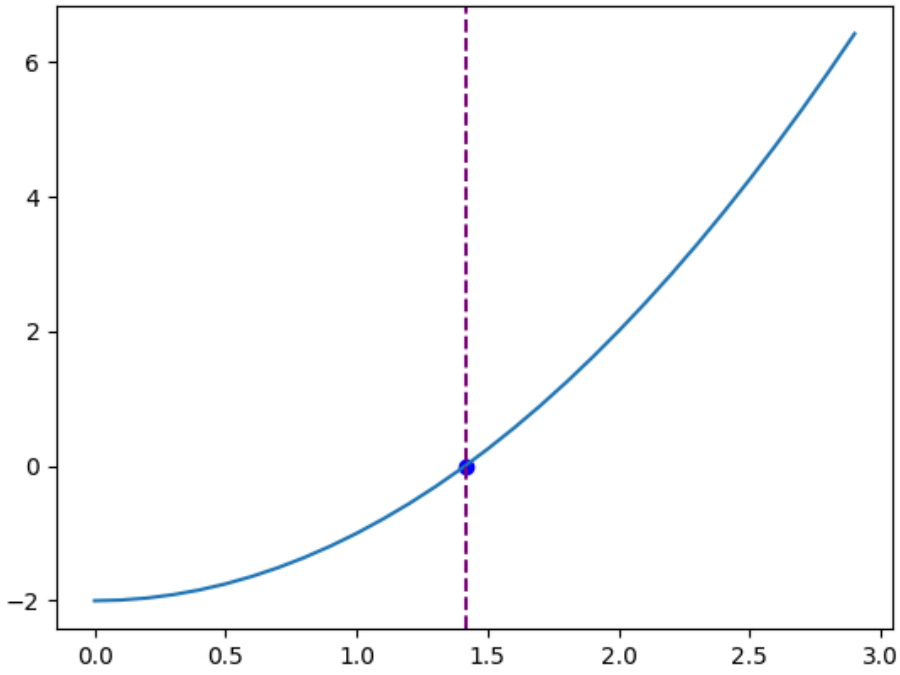
```
    print("Maximum iterations reached without convergence.")
    return None

print(secant_method(f, 1, 2))
```

```
1.4142135620573204
```

Let's plot it...

```
root = secant_method(f, 1, 2)

# Let's plot the result
x = np.arange(0, 3, 0.1)
plt.plot(x, f(x), label='f(x) = x^2 - 2')
plt.scatter(root, f(root), color='blue')  # Mark the root on the plot
plt.axvline(root, color='purple', linestyle='--', label=f'x = root
({root:.5f})')
```

```
<matplotlib.lines.Line2D at 0x7fef8f89bed0>
```



2.e. Linear Regression

In linear regression, we model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to the observed data. The goal is to find the best-fitting line (or hyperplane in higher dimensions) that minimizes the difference between the predicted values and the actual values.

Let's import the necessary libraries and create some sample data for linear regression.

```
import numpy as np
from sklearn.linear_model import LinearRegression
from matplotlib import pyplot as plt
```

Let's create some sample data for linear regression. Let's say we want to predict y from x using a linear relationship!

```
np.random.seed(0)
x = 2 * np.random.rand(100, 1)
y = 4 + 3 * x + np.random.randn(100, 1)
```

Create a linear regression model

```
model = LinearRegression()
model.fit(x, y)
```

```
LinearRegression()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org. [x]LinearRegression ?Documentation for LinearRegressioniFitted

Make predictions

```
x_new = np.array([[0], [2]])
y_predict = model.predict(x_new)
```

Print the coefficients

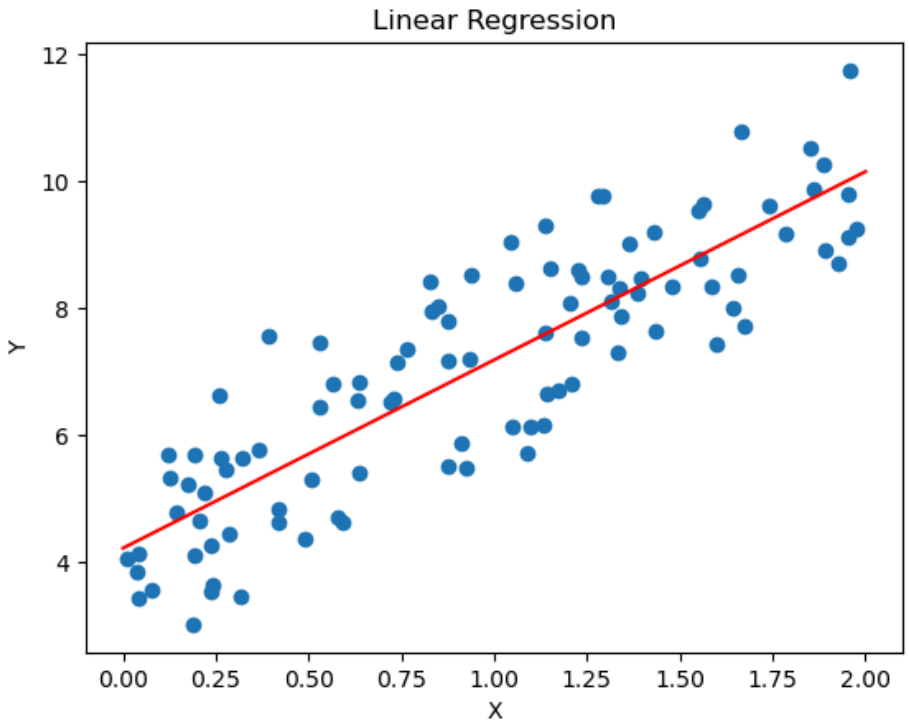
```
print("Intercept:", model.intercept_)
print("Coefficient:", model.coef_)
```

```
Intercept: [4.22215108]
Coefficient: [[2.96846751]]
```

Plot the results

```
plt.scatter(x, y)
plt.plot(x_new, y_predict, color='red')

plt.xlabel('X')
plt.ylabel('Y')
```

```
plt.title('Linear Regression')
plt.show()
```



2.e.i. *Gradient Descent:*

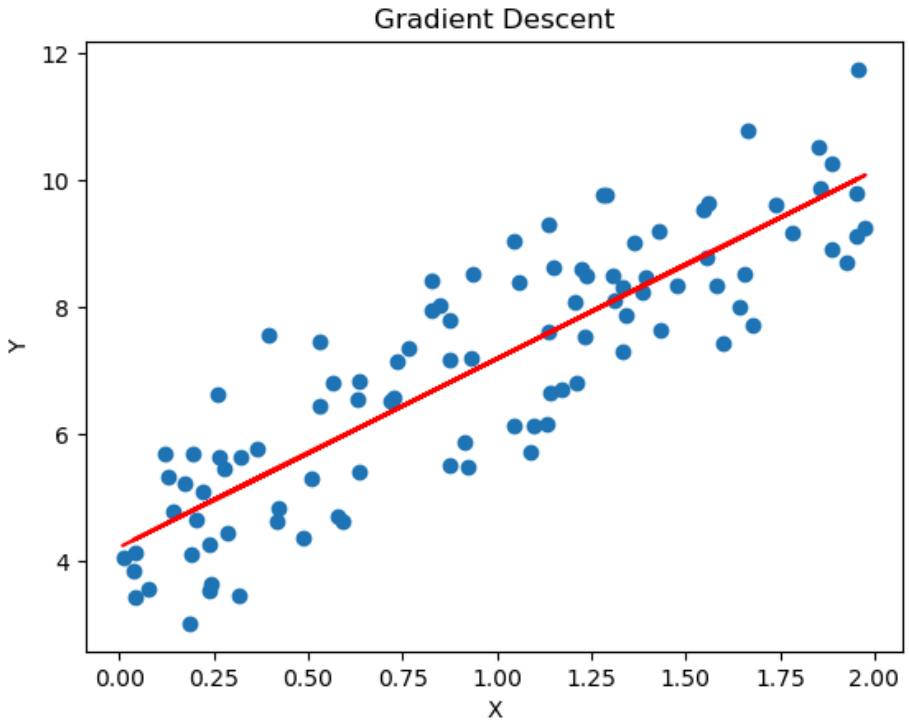
Gradient descent is an optimization algorithm used to minimize a function by iteratively moving towards the steepest descent, which is determined by the negative of the gradient. In the context of linear regression, gradient descent is used to find the optimal coefficients that minimize the cost function (usually the mean squared error).

```
def gradient_descent(x, y, m = 0, b = 0, learning_rate = 0.01, epochs =
10000):
    n = len(y)
    for _ in range(epochs):
        y_pred = m * x + b
        dm = (-2/n) * sum(x * (y - y_pred))
        db = (-2/n) * sum(y - y_pred)
        m -= learning_rate * dm
        b -= learning_rate * db
    return m, b
```

Let's plot...

```
plt.scatter(x, y)

m, b = gradient_descent(x, y)
plt.plot(x, m*x + b, color='red')

plt.xlabel('X')
```

```
plt.ylabel('Y')

plt.title('Gradient Descent')
plt.show()
```



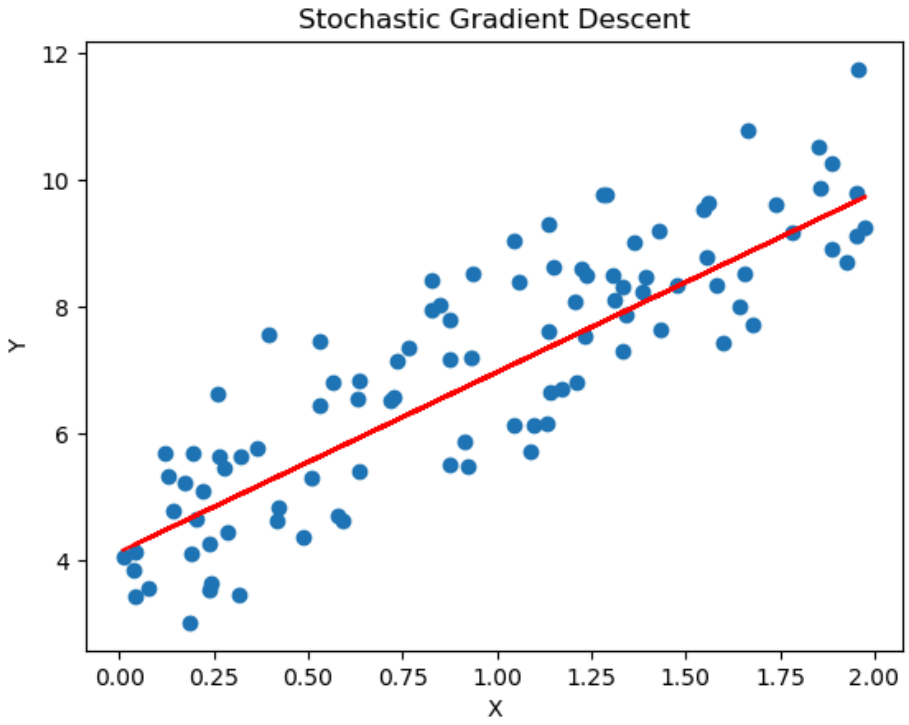
2.e.ii. *Stochastic Gradient Descent:*

Stochastic Gradient Descent (SGD) is a variant of the gradient descent algorithm that updates the model parameters using only a single or a few training examples at each iteration, rather than the entire dataset. This makes SGD more efficient for large datasets and can help the model converge faster.

```
def stochastic_gradient_descent(x, y, m=0, b=0, learning_rate=0.01,
epochs=10000):
    n = len(y)
    for _ in range(epochs):
        for i in range(n):
            xi = x[i:i+1]
            yi = y[i:i+1]
            y_pred = m * xi + b
            dm = -2 * xi * (yi - y_pred)
            db = -2 * (yi - y_pred)
            m -= learning_rate * dm
            b -= learning_rate * db
    return m, b
```

Let's plot it...

```
plt.scatter(x, y)

m, b = stochastic_gradient_descent(x, y)
plt.plot(x, m*x + b, color='red')

plt.xlabel('X')
plt.ylabel('Y')

plt.title('Stochastic Gradient Descent')
plt.show()
```



2.e.iii. *Logistic Regression (Bonus):*

Logistic regression is a statistical method used for binary classification problems, where the goal is to predict the probability of an instance belonging to one of two classes. Unlike linear regression, which predicts continuous values, logistic regression predicts probabilities that are then mapped to discrete classes (0 or 1).

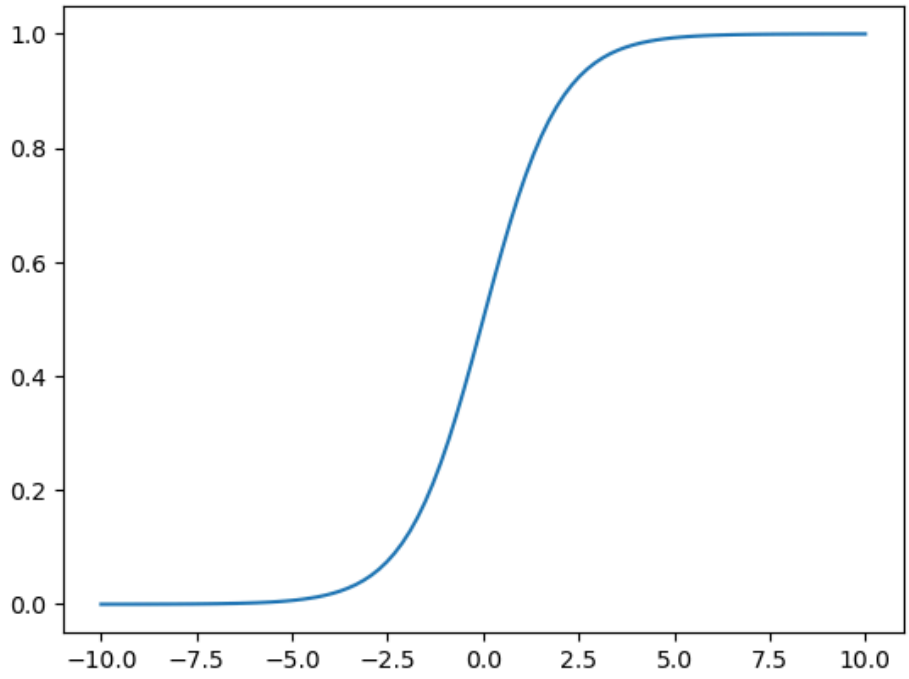
```
def logistic_regression(x):
    return 1 / (1 + np.exp(-x))

def predict_logistic(x, m, b):
    linear_combination = m * x + b
    return logistic_regression(linear_combination)

# An example usage
predict_logistic(0, 1, 0), predict_logistic(1, 1, 0),
predict_logistic(-1, 1, 0)
```

```
(np.float64(0.5),
 np.float64(0.7310585786300049),
 np.float64(0.2689414213699951))
```

Let's plot the curve...

```
x = np.linspace(-10, 10, 100)
y = logistic_regression(x)
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7feae2157ed0>]
```



2.f. Polynomial Regression

2.f.i. *Sample funtion:*

First, let's import `matplotlib.pyplot` and `numpy`, one for plotting and the other for numerical operations.

```
import matplotlib.pyplot as plt
import numpy as np
```

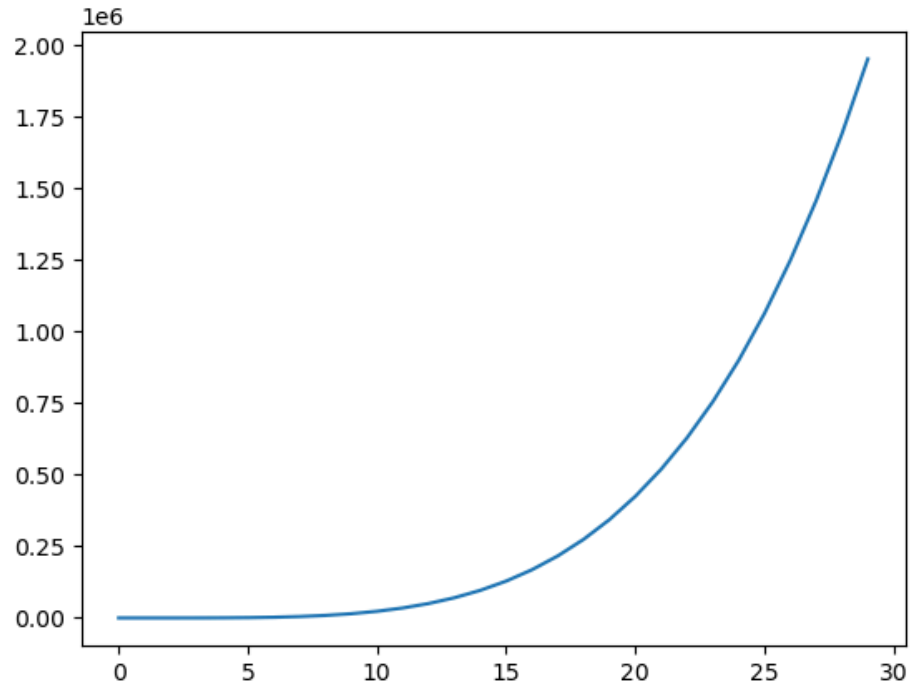
A basic function,

```
def fun(x):
    return 3 * x**4 - 7 * x**3 + 2 * x**2 + 11
```

Let's plot it!

```
x = np.arange(0, 30)
plt.plot(x, fun(x))
```

```
[<matplotlib.lines.Line2D at 0x7f51737a6710>]
```



2.f.ii. *Polynomial Regression:*

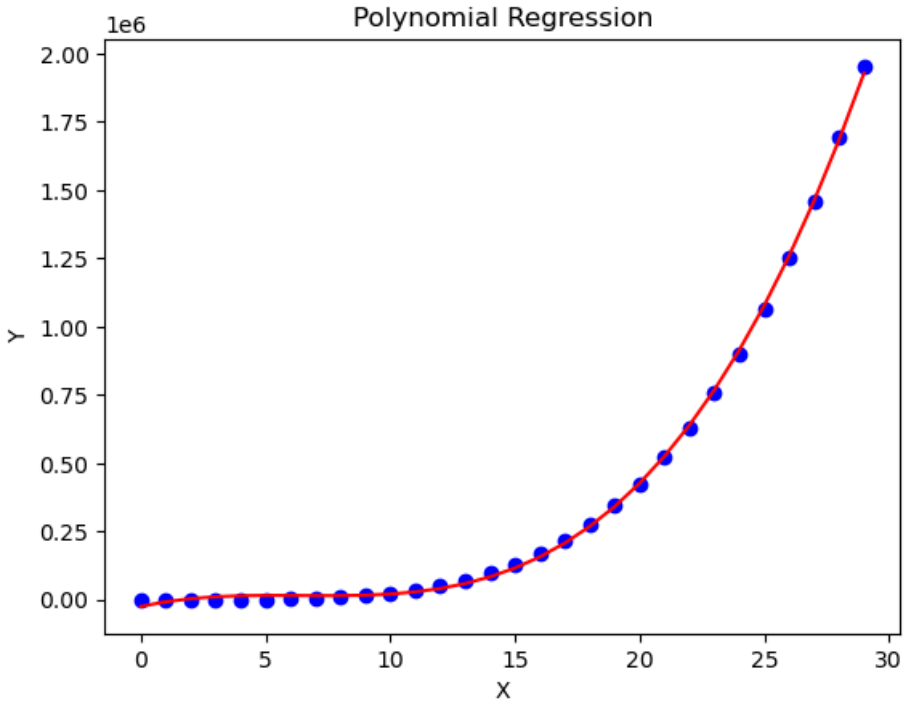
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

def polynomial_regression(x, y, degree=3):
  poly = PolynomialFeatures(degree)
  x_poly = poly.fit_transform(x.reshape(-1, 1))

  model = LinearRegression()
```

```
  model.fit(x_poly, y)
  return model.predict(x_poly)
```

Let's visualize our result,

```
y_pred = polynomial_regression(x, fun(x))
plt.title('Polynomial Regression')
plt.plot(x, y_pred, color='red')
plt.scatter(x, fun(x), color='blue')
plt.xlabel('X')
plt.ylabel('Y')
```

```
Text(0, 0.5, 'Y')
```



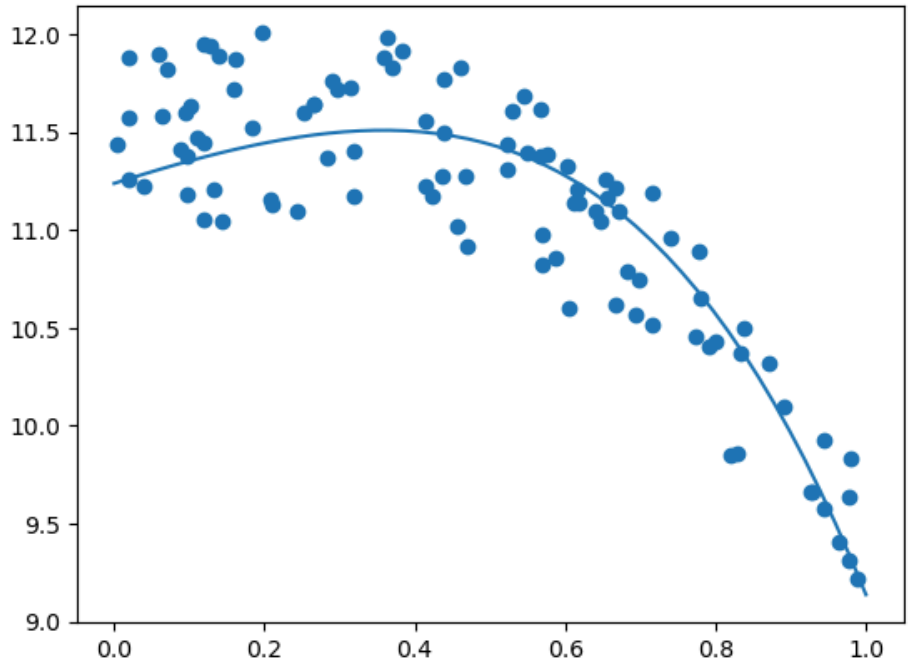
2.f.iii. *Gradient Descent:*

We can implement gradient descent to minimize the cost function for polynomial regression.

```
np.random.seed(0)
x = np.random.rand(100,1)
y = 3 * x**4 - 7 * x**3 + 2 * x**2 + 11 + np.random.rand(100,1)

# m0 x4 + m1 x3 + m2 x2 + m3 x + m4
def gd(x, y, m0=0, m1=0, m2=0, m3=0, m4=0, epoch=10000, learn=0.001):
    n = len(x)
    for i in range(epoch):
        y_n = m0 * x**4 + m1 * x**3 + m2 * x**2 + m3 * x + m4
        m4_l = - 2 / n * np.sum(y - y_n)
        m3_l = - 2 / n * np.sum((y - y_n) * x)
```

```
        m2_l = - 2 / n * np.sum((y - y_n) * x**2)
        m1_l = - 2 / n * np.sum((y - y_n) * x**3)
        m0_l = - 2 / n * np.sum((y - y_n) * x**4)

        m4 = m4 - learn * m4_l
        m3 = m3 - learn * m3_l
        m2 = m2 - learn * m2_l
        m1 = m1 - learn * m1_l
        m0 = m0 - learn * m0_l
    return m0, m1, m2, m3, m4


m0, m1, m2, m3, m4 = gd(x, y)
print(gd(x, y))
x_p = np.linspace(0, 1)

plt.plot(x_p, m0 * x_p**4 + m1 * x_p**3 + m2 * x_p**2 + m3 * x_p + m4)
plt.scatter(x,y)
plt.show()
```

```
(np.float64(-1.4008617749474424), np.float64(-1.237676294880945),
np.float64(-0.689765797956417), np.float64(1.228140471950126),
np.float64(11.237160966215388))
```



2.g. Logistic Regression

Logistic regression is used for binary classification tasks. It models the probability that a given input belongs to a particular class using the logistic function.

2.g.i. *From Linear:*

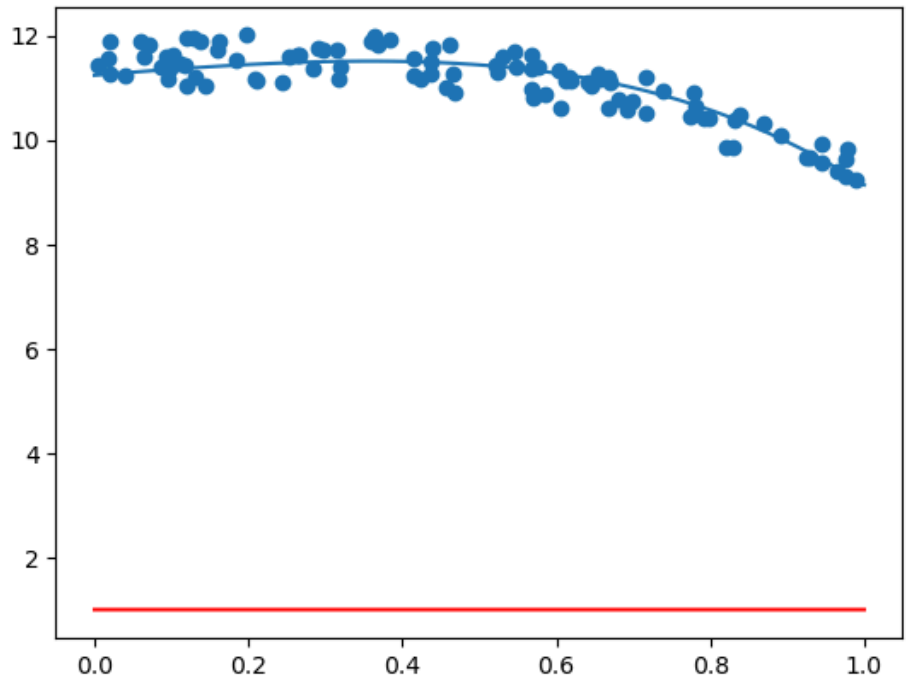
Check the ***Linear Regression*** section's last part.

2.g.ii. *From Polynomial:*

Let's take the function from our previous polynomial regression example and convert it into a binary classification problem. We'll classify points as belonging to class 1 if the output is greater than a certain threshold, and class 0 otherwise.

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
x = np.random.rand(100,1)
y = 3 * x**4 - 7 * x**3 + 2 * x**2 + 11 + np.random.rand(100,1)

# m0 x4 + m1 x3 + m2 x2 + m3 x + m4
def gd(x, y, m0=0, m1=0, m2=0, m3=0, m4=0, epoch=10000, learn=0.001):
    n = len(x)
    for i in range(epoch):
        y_n = m0 * x**4 + m1 * x**3 + m2 * x**2 + m3 * x + m4
        m4_l = - 2 / n * np.sum(y - y_n)
        m3_l = - 2 / n * np.sum((y - y_n) * x)
        m2_l = - 2 / n * np.sum((y - y_n) * x**2)
        m1_l = - 2 / n * np.sum((y - y_n) * x**3)
        m0_l = - 2 / n * np.sum((y - y_n) * x**4)

        m4 = m4 - learn * m4_l
        m3 = m3 - learn * m3_l
        m2 = m2 - learn * m2_l
        m1 = m1 - learn * m1_l
        m0 = m0 - learn * m0_l
    return m0, m1, m2, m3, m4

m0, m1, m2, m3, m4 = gd(x, y)
print(gd(x, y))
x_p = np.linspace(0, 1)

plt.plot(x_p, m0 * x_p**4 + m1 * x_p**3 + m2 * x_p**2 + m3 * x_p + m4)
plt.scatter(x,y)
```

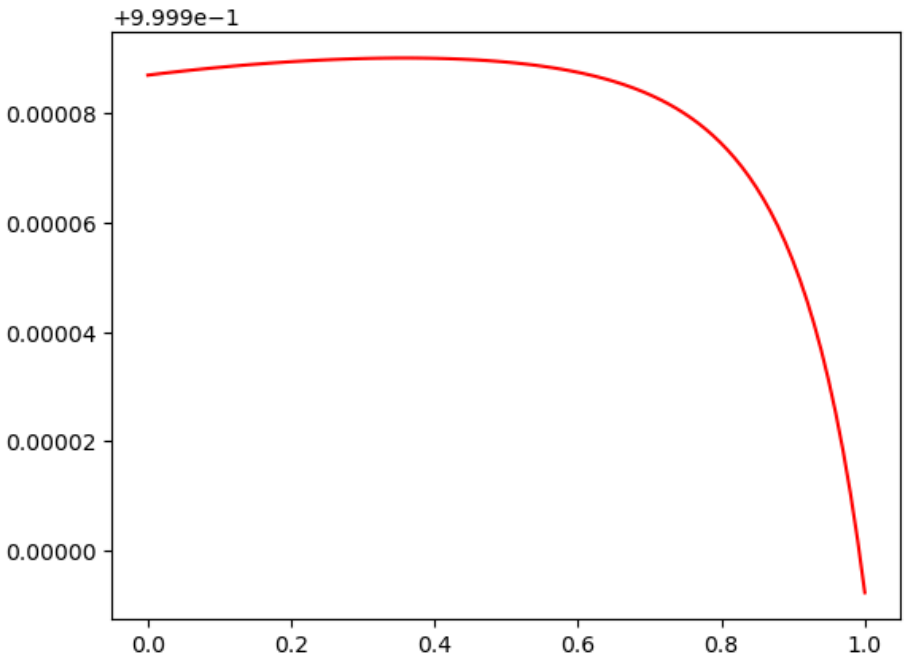
```
(np.float64(-1.4008617749474424), np.float64(-1.237676294880945),
np.float64(-0.689765797956417), np.float64(1.228140471950126),
np.float64(11.237160966215388))
```



Let's implement logistic based on the above function.

```
def logistic(z):
    return 1 / (1 + np.exp(-z))

def predict(x, m0, m1, m2, m3, m4, threshold=0.5):
    z = m0 * x**4 + m1 * x**3 + m2 * x**2 + m3 * x + m4
    return logistic(z)

x = np.linspace(0, 1, 100).reshape(-1, 1)
plt.plot(x, predict(x, m0, m1, m2, m3, m4), color='red')
```

```
[<matplotlib.lines.Line2D at 0x7fc171cdd6d0>]
```


+9.999e−1
0.00008
0.00006
0.00004
0.00002
0.00000
0.0
0.2
0.4
0.6
0.8
1.0

2.h. Polynomial Interpolation

Polynomial interpolation is a method of estimating values between known data points using polynomials. Given a set of data points, the goal is to find a polynomial that passes through all of these points.

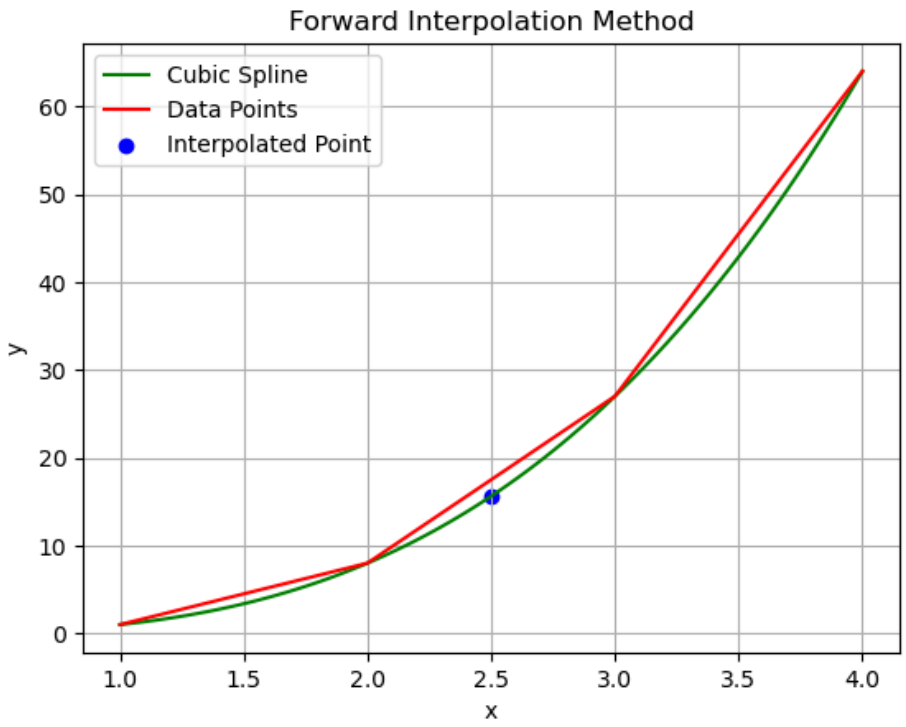
```
# some libs
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
import numpy as np
```

2.h.i. *Newton's Forward Interpolation:*

Let's define for the first 3 differences,

```
def forward(x, y, x_n):
    arr = [y]
    for i in range(3):
        temp = []
        for j in range(0, len(arr[-1])-1):
            temp.append(arr[-1][j+1] - arr[-1][j])
        arr.append(temp)
    print(arr)

    h = (x_n - x[0]) / (x[1] - x[0])
    a = arr[0][0]
    b = h * arr[1][0]
    c = h * (h-1) / 2 * arr[2][0]
    d = h * (h-1) * (h-2) / 6 * arr[3][0]

    return a + b + c + d
x = [1, 2,  3,  4]
y = [1, 8, 27, 64]
x_n = 2.5
y_n = forward(x, y, x_n)

x, y = np.array(x), np.array(y)
fn = interp1d(x, y, kind='cubic')
x_new = np.linspace(x.min(), x.max(), 100)
y_new = fn(x_new)

plt.plot(x_new, y_new, label='Cubic Spline', color='green')
plt.plot(x, y, color='red', label='Data Points')
plt.scatter(x_n, y_n, color='blue', label='Interpolated Point')
plt.title('Forward Interpolation Method')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid()
plt.show()
```

```
[[1, 8, 27, 64], [7, 19, 37], [12, 18], [6]]
```



2.h.ii. *Newton's Backward Interpolation:*

Same as before, but using backward differences. **Try it yourself!**

2.h.iii. *Lagrange Interpolation:*

Lagrange interpolation is another method for polynomial interpolation that constructs a polynomial that passes through a given set of points.

```
def lagrange(x, y, x_n):
    n = len(x)
    res = 0
    for i in range(n):
        temp = y[i]
        for j in range(n):
            if i != j:
                temp *= x_n - x[j]
                temp /= x[i] - x[j]
        res += temp
    return res

y_n_lagrange = lagrange(x, y, x_n)
print(f"Lagrange Interpolated value at x = {x_n} is y = {y_n_lagrange}")
```

```
Lagrange Interpolated value at x = 2.5 is y = 15.625
```

2.i. Differential Equations

A differential equation is a mathematical equation that relates a function with its derivatives. In simpler terms, it describes how a quantity changes in relation to another quantity. Differential equations are fundamental in various fields such as physics, engineering, and economics, as they model dynamic systems and processes.

2.i.i. *Euler's Method:*

Euler's method is a simple and widely used numerical technique for solving ordinary differential equations (ODEs) with a given initial value. It is particularly useful for approximating solutions to first-order ODEs of the form:

$$\frac{dy}{dx} = f(x, y) \tag{3}$$

with an initial condition $y(x_0) = y_0$.

Let's consider an example,

$$\frac{dy}{dx} = x + y \tag{4}$$

$$y(0) = 1 \tag{5}$$

Let's import libraries first,

```
import numpy as np
import matplotlib.pyplot as plt
```

Our differential equation,

$$\frac{dy}{dx} = x + y \tag{6}$$

```
def f(x, y):
    return x + y
```

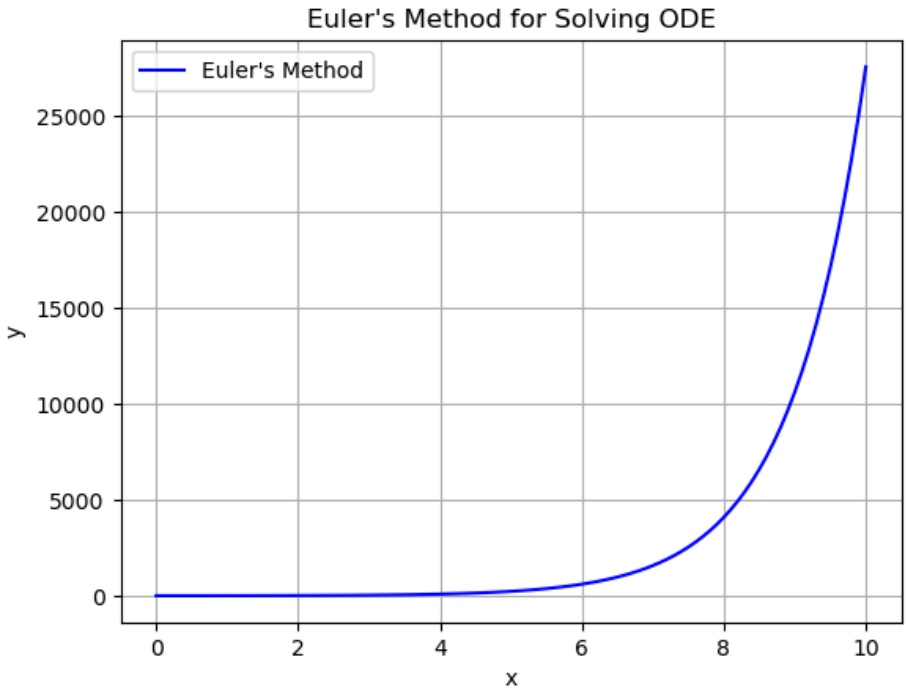
And our euler function,

```
def euler_method(f, x0, y0, h=0.1, n=100):
    x_values = [x0]
    y_values = [y0]

    for i in range(n):
        y0 = y0 + h * f(x0, y0)
        x0 = x0 + h
        x_values.append(x0)
        y_values.append(y0)

    return np.array(x_values), np.array(y_values)
```

Let's plot and visualize the results,

```
# Initial conditions and parameters
x0 = 0
```

```
y0 = 1
h = 0.1  # Step size
n = 100  # Number of steps
x_values, y_values = euler_method(f, x0, y0, h, n)

# Plotting the results
plt.plot(x_values, y_values, label="Euler's Method", color='blue')
plt.title("Euler's Method for Solving ODE")
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid()
```



2.i.ii. *Runge-Kutta Method:*

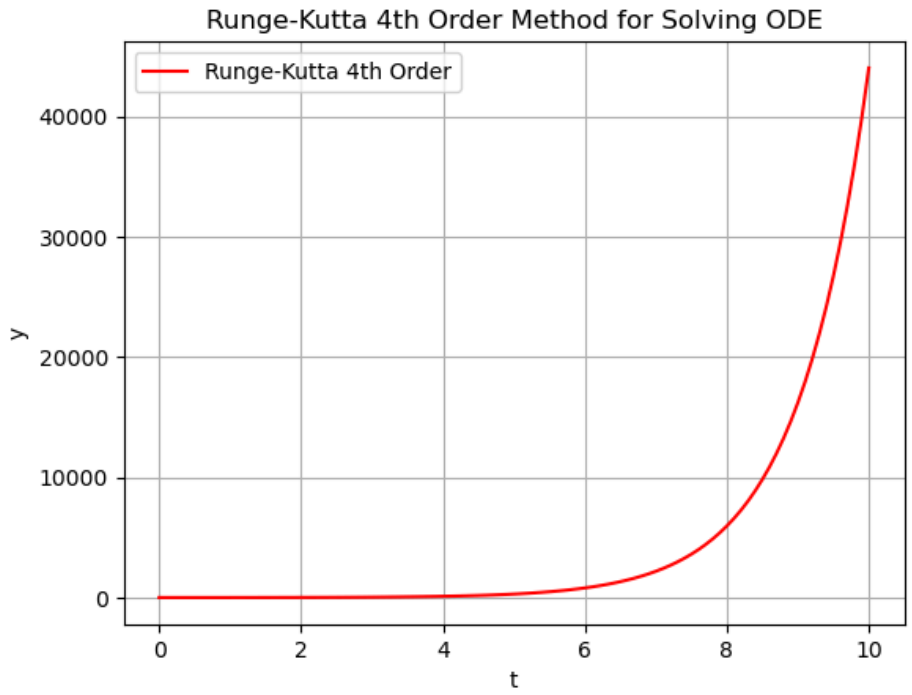
The Runge-Kutta methods are a family of iterative methods used to solve ordinary differential equations (ODEs). The most commonly used version is the fourth-order Runge-Kutta method (RK4), which provides a good balance between accuracy and computational efficiency.

```
def runge_kutta_4th_order(f, y0, x0, x1, h):
    """
    y0: Initial value of y at x0
    x0: Initial value of x
    x1: Final value of x
    h: Step size
    f: Function that returns dy/dt given t and y
    """
    n = int((x1 - x0) / h)
```

```
    x_values = np.linspace(x0, x1, n + 1)
    y_values = np.zeros(n + 1)
    y_values[0] = y0

    for i in range(n):
        x = x_values[i]
        y = y_values[i]

        k1 = h * f(x, y)
        k2 = h * f(x + h / 2, y + k1 / 2)
        k3 = h * f(x + h / 2, y + k2 / 2)
        k4 = h * f(x + h, y + k3)

        y_values[i + 1] = y + (k1 + 2 * k2 + 2 * k3 + k4) / 6

    return x_values, y_values
```

Let's plot it,

```
# Initial conditions and parameters for RK4
y0 = 1
x0 = 0
x1 = 10
h = 0.1
x_values, y_values_rk4 = runge_kutta_4th_order(f, y0, x0, x1, h)

# Plotting the results of RK4
plt.plot(x_values, y_values_rk4, label="Runge-Kutta 4th Order",
color='red')
plt.title("Runge-Kutta 4th Order Method for Solving ODE")
plt.xlabel('t')
plt.ylabel('y')
plt.legend()
plt.grid()
```



2.i.iii. *Picard's Method:*

Picard's method is an iterative technique used to approximate the solutions of ordinary differential equations (ODEs). It is particularly useful for solving initial value problems of the form:

$$\frac{dy}{dx} = f(x, y) \tag{7}$$

with an initial condition $y(x_0) = y_0$.

Mathematically, Picard's method constructs a sequence of functions that converge to the solution of the differential equation. The iterative formula is given by:

$$y_{n+1}(x) = y_0 + \int_{x_0}^{x} f(t, y_n(t))\, dt \tag{8}$$

where $y_n(x)$ is the nth approximation of the solution.

So, first apporximation is,

$$y_1(x) = y_0 + \int_{x_0}^{x} f(t, y_0)\, dt \tag{9}$$

The second approximation is,

$$y_2(x) = y_0 + \int_{x_0}^{x} f(t, y_1(t))\, dt \tag{10}$$

And so on...

For example, consider the initial value problem:

$$\frac{dy}{dx} = 1 + xy \tag{11}$$

$$y(0) = 1 \tag{12}$$

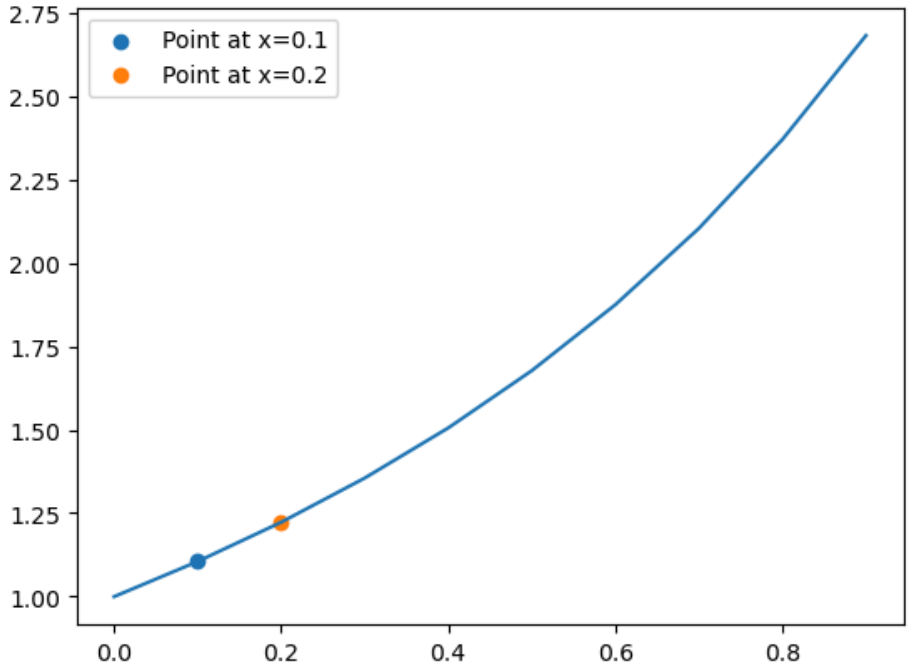
```
# Apporoximations
def Y1(x):
    return 1 + (x) + pow(x, 2) / 2


def Y2(x):
    return 1 + (x) + pow(x, 2) / 2 + pow(x, 3) / 3 + pow(x, 4) / 8


def Y3(x):
    return (
        1
        + (x)
        + pow(x, 2) / 2
        + pow(x, 3) / 3
        + pow(x, 4) / 8
        + pow(x, 5) / 15
        + pow(x, 6) / 48
    )


def picard_method(f, x0, h=0.1, n=10, iterations=3):
    x_values = np.arange(x0, x0 + n * h, h)
    y_values = np.array([f(i) for i in x_values])

    return x_values, y_values

picard_method(f=Y3, x0=0, h=0.1)
```

```
(array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]),
 array([1.        , 1.10534652, 1.22288933, 1.35518969, 1.50530133,
        1.67688802, 1.874356  , 2.10300152, 2.36917333, 2.68045019]))
```

Let's plot it...

```
x_values, y_values = picard_method(f=Y3, x0=0, h=0.1)

plt.plot(x_values, y_values)
plt.scatter(x_values[1], y_values[1], label=f'Point at
x={x_values[1]:.1f}')
plt.scatter(x_values[2], y_values[2], label=f'Point at
x={x_values[2]:.1f}')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f7e20086350>
```



2.i.iv. *Milne's Predictor-Corrector Method:*

Milne's predictor-corrector method is a numerical technique used to solve ordinary differential equations (ODEs). It is a multi-step method that combines both prediction and correction steps to improve the accuracy of the solution.

The formula is,

$$y_{n+1,p} = y_{n-3} + \frac{4h}{3}(2f_n - f_{n-1} + 2f_{n-2}) \tag{13}$$

$$y_{n+1} = y_{n-1} + \frac{h}{3}(f_{n+1,p} + 4f_n + f_{n-1}) \tag{14}$$

Where $y_{n+1,p}$ is the predicted value and y_{n+1} is the corrected value.

If the value of n = 3, then our formula becomes,

$$y_{4,p} = y_0 + \frac{4h}{3}(2f_3 - f_2 + 2f_1) \tag{15}$$

$$y_4 = y_2 + \frac{h}{3}(f_{4,p} + 4f_3 + f_2) \tag{16}$$

```
# Let's get x0, y0, x1, y1, x2, y2, x3, y3 from the picard method
x0, y0 = x_values[0], y_values[0]
x1, y1 = x_values[1], y_values[1]
x2, y2 = x_values[2], y_values[2]
x3, y3 = x_values[3], y_values[3]

print(f"x0: {x0}, y0: {y0}")
print(f"x1: {x1}, y1: {y1}")
```

```
print(f"x2: {x2}, y2: {y2}")
print(f"x3: {x3}, y3: {y3}")

# Let's get y4 from milne's method
def milne_method(x0, y0, x1, y1, x2, y2, x3, y3, x4, h=x1-x0):
    # Predictor
    y4_pred = y0 + (4 * h / 3) * (2 * f(x3, y3) - f(x2, y2) + 2 * f(x1,
y1))
    # Corrector
    y4_corr = y2 + (h / 3) * (f(x2, y2) + 4 * f(x3, y3) + f(x4,
y4_pred))
    return y4_corr

y4_milne = milne_method(x0, y0, x1, y1, x2, y2, x3, y3, x4=x3 + (x1 -
x0))
print(f"y4 from Milne's method: {y4_milne}")
```

```
x0: 0.0, y0: 1.0
x1: 0.1, y1: 1.1053465208333333
x2: 0.2, y2: 1.2228893333333333
x3: 0.30000000000000004, y3: 1.3551896875
y4 from Milne's method: 1.5567806387037035
```

2.j. Integrals

Let's approximate the integral of a function using numerical methods.

2.j.i. *Trapezoidal Rule:*

At first our necessary libs,

```
import numpy as np
import matplotlib.pyplot as plt
```

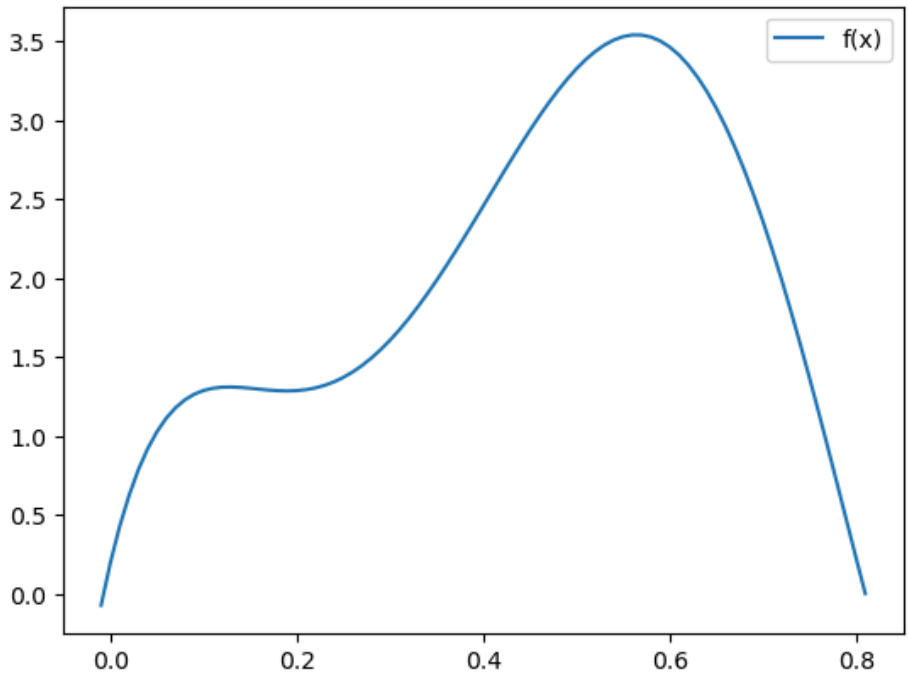
Let's define a sample function to integrate.

```
def fun(x):
  return 0.2 + 25 * x - 200 * x**2 + 675 * x**3 - 900 * x**4 + 400 *
x**5
```

Let's plot it!

```
array = np.arange(-0.01, 0.82, 0.01)
plt.plot(array, fun(array), label='f(x)')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fd35fb6ee90>
```



Let's define our Trapezoidal Rule!

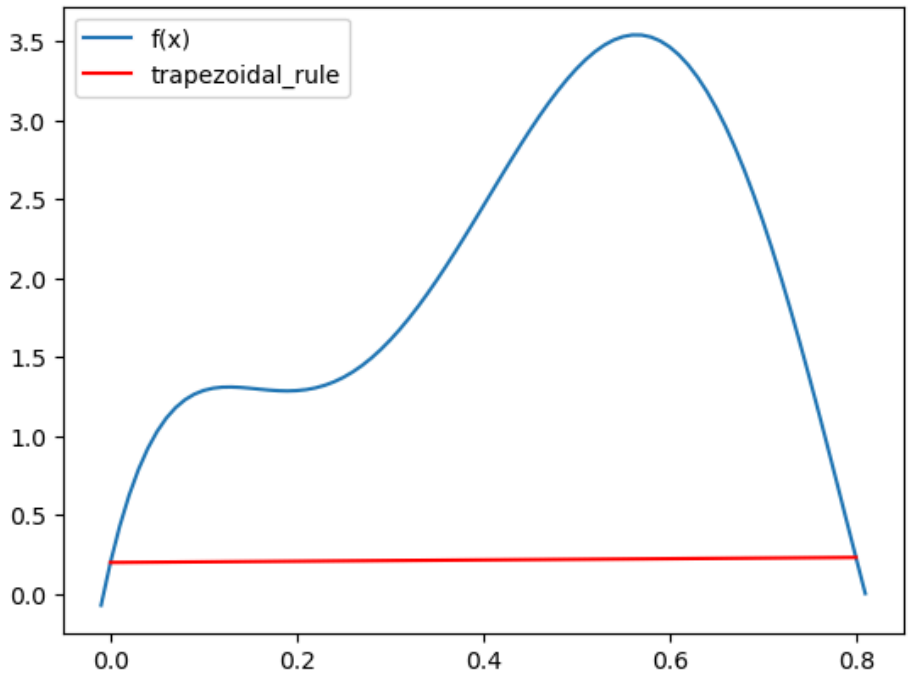
```
def trapezoidal_rule(fun, a, b):
  return (b - a) * (fun(a) + fun(b)) / 2
```

Our trapezoidal rule function is ready to use!

```
print(trapezoidal_rule(fun, 0, 0.8))
plt.plot(array, fun(array), label='f(x)')
plt.plot([0, 0.8], [fun(0), fun(0.8)], color='red',
label='trapezoidal_rule')
plt.legend()
```

```
0.1728000000000225
```

```
<matplotlib.legend.Legend at 0x7fd35fbde5d0>
```



2.j.ii. *Simspon's 1/3 Rule:*

Let's implement it,

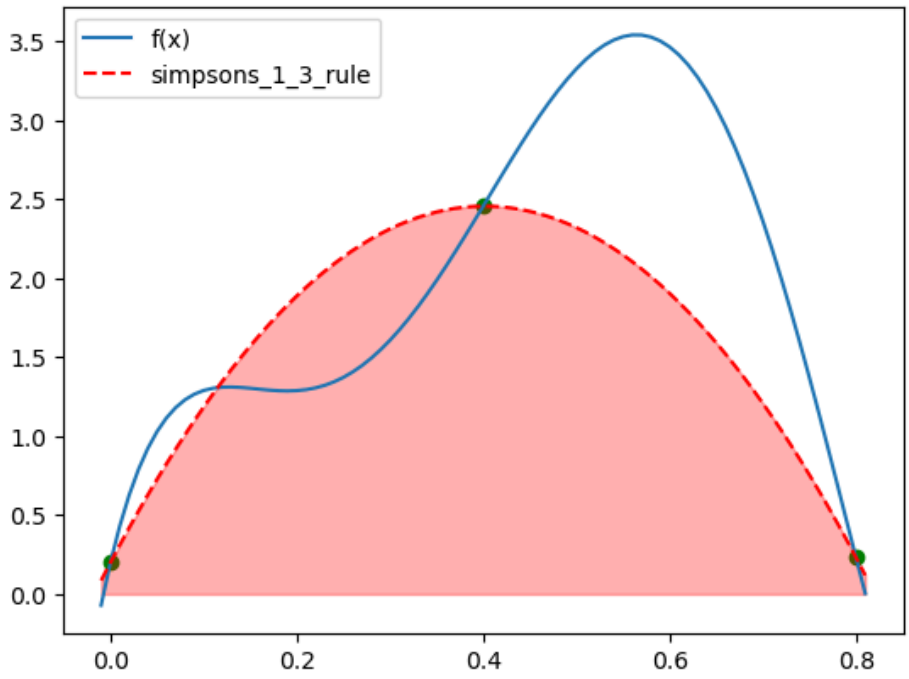
```
def simpsons_1_3_rule(fun, a, b):
    return (b - a) / 6 * (fun(a) + 4 * fun((a + b) / 2) + fun(b))
```

Let's plot it,

```
from scipy.interpolate import CubicSpline

print(simpsons_1_3_rule(fun, 0, 0.8))

plt.plot(array, fun(array), label="f(x)")
plt.scatter([0, 0.4, 0.8], [fun(0), fun(0.4), fun(0.8)], color='green')
plt.plot(array, CubicSpline([0, 0.4, 0.8], [fun(0), fun(0.4), fun(0.8)])
(array), '--', color='red', label='simpsons_1_3_rule')
plt.fill_between(array, CubicSpline([0, 0.4, 0.8], [fun(0), fun(0.4),
fun(0.8)])(array), color='red', alpha=0.3)
plt.legend()
```

```
1.3674666666666742
```

```
<matplotlib.legend.Legend at 0x7fd36173c7d0>
```



2.j.iii. *Simpson's 3/8 Rule:*

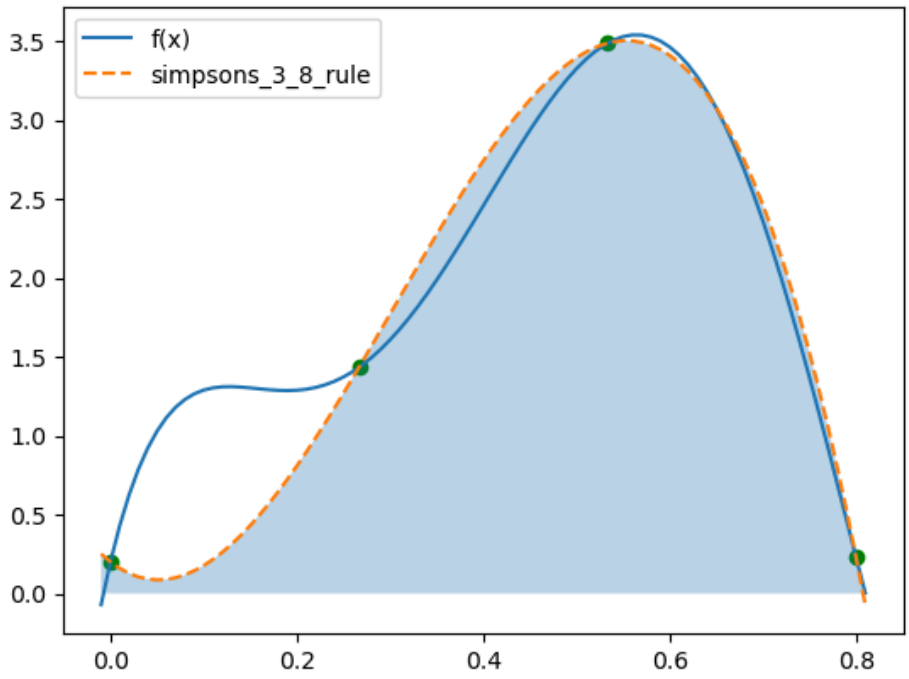
```
def simpsons_3_8_rule(fun, a, b):
    a, b = min(a, b), max(a, b)
    h = (b - a) / 3
    return (3 * h / 8) * (fun(a) + 3 * fun(a + h) + 3 * fun(a + 2 * h) +
fun(b))
```

Let's plot it,

```
from scipy.interpolate import CubicSpline

a, b, c, d = 0, 0.8/3, 0.8/3*2, 0.8
print(simpsons_3_8_rule(fun, a, d))

plt.plot(array, fun(array), label="f(x)")
plt.plot(array, CubicSpline([a, b, c, d], [fun(a), fun(b), fun(c),
fun(d)])(array), '--', label="simpsons_3_8_rule")
plt.scatter([a, b, c, d], [fun(a), fun(b), fun(c), fun(d)],
color='green')
plt.fill_between(array, CubicSpline([a, b, c, d], [fun(a), fun(b),
fun(c), fun(d)])(array), alpha=0.3)

plt.legend()
```

```
1.519170370370378
```

```
<matplotlib.legend.Legend at 0x7fd3615d9450>
```



Let's plot all of them altogether,

```
# Main function
plt.plot(array, fun(array), label="f(x)")

# Trapezoidal rule
print("Trapezoidal rule   -> ", trapezoidal_rule(fun, 0, 0.8))
plt.plot([0, 0.8], [fun(0), fun(0.8)], '--', color='green',
label='trapezoidal_rule')
plt.fill_between([0, 0.8], [fun(0), fun(0.8)], color='green', alpha=0.3)

# Simpson's 1/3 rule
print("Simpson's 1/3 rule -> ", simpsons_1_3_rule(fun, 0, 0.8))

plt.scatter([0, 0.4, 0.8], [fun(0), fun(0.4), fun(0.8)], color='green')
plt.plot(array, CubicSpline([0, 0.4, 0.8], [fun(0), fun(0.4), fun(0.8)])
(array), color='red', label='simpsons_1_3_rule')
plt.fill_between(array, CubicSpline([0, 0.4, 0.8], [fun(0), fun(0.4),
fun(0.8)])(array), color='red', alpha=0.3)

a, b, c, d = 0, 0.8/3, 0.8/3*2, 0.8

# Simpson's 3/8 rule
print("Simpson's 3/8 rule -> ", simpsons_3_8_rule(fun, a, d))

plt.scatter([a, b, c, d], [fun(a), fun(b), fun(c), fun(d)],
color='green')

plt.plot(array, CubicSpline([a, b, c, d], [fun(a), fun(b), fun(c),
fun(d)])(array), '--', color='blue', label="simpsons_3_8_rule")
```

```
plt.fill_between(array, CubicSpline([a, b, c, d], [fun(a), fun(b),
fun(c), fun(d)])(array), alpha=0.3)

plt.legend()
```

```
Trapezoidal rule   ->  0.1728000000000225
Simpson's 1/3 rule ->  1.3674666666666742
Simpson's 3/8 rule ->  1.519170370370378
```

```
<matplotlib.legend.Legend at 0x7fd35fc4f9d0>
```