Contents

Python Programming- Lecture 1

1 strings

```python
s='hello'
print(s)
type(s)
print(type(s))
```
    ⇒ **hello**
    ⇒ **<class 'str'>**

```python
# 1.2 strings
s='hello'
print(s)
type(s)
```
    ⇒ **hello**
    ⇒ **str**

More about 'type'.

```python
python                                    Copy code

x = 5
y = 'hello'
z = [1, 2, 3]

print(type(x))   # Output: <class 'int'>
print(type(y))   # Output: <class 'str'>
print(type(z))   # Output: <class 'list'>
```

There are situations where knowing the type explicitly is necessary, such as when dealing with certain operations or debugging.

#2. string manipulation

```python
s="hello"
print(s)
print(s[2])
```

```python
#Accessing individual characters by index. In Python, indexing
starts at 0.
print(s[0],s[1],s[2],s[3],s[4])
# Accessing and printing characters using multiple indices.
print(s[0],s[1],s[-1],s[-2],len(s))
# Accessing characters using both positive and negative indices. Negative indices
for last character start from -1. the length of the string is denoted by len(s).

print(s+" world")
# This concatenates the string "hello" with the string " world" and prints the result.
The output will be "hello world".
```

- ⇨ **Hello**
- ⇨ **L**
- ⇨ **h e l l o**
- ⇨ **h e o l 5**
- ⇨ **hello world**

3. string is immutable

```python
s='Hello'
print(s)
type(s)
s[1]='b'    # see this statement itself highlighted
```

- ⇨ **TypeError: 'str' object does not support item
  assignment**

Explanation: you're trying to assign a new value ('b') to the character at index 1 of the string **s**. However, this will result in an error because strings in Python are immutable.

If you need to modify a string, you can create a new string with the desired changes. For example.

```python
s = 'Hello'
s = s[:1] + 'b' + s[2:]
print(s)
```

- ⇨ **Hbllo**

Explanation:

s[:1]: This is a slicing operation that extracts the substring from the beginning of the string up to, *but not including,* the character at index 1. In this case, it takes the first character 'H'.

s[2:]: This is another slicing operation that extracts the substring starting from index 2 *(including)* to the end of the string. In this case, it takes the characters 'llo'.

4. strings and integer

```
s="hello"
num=10

print(num)
print(s+num)
```

> ⇨ **TypeError: can only concatenate str (not "int") to str**

In Python, you can't directly concatenate a string and an integer using the + operator without converting the integer to a string first.

Here's an updated version of your code:

```
s = "hello"
num = 10

print(num)

# Convert the integer to a string before concatenating
print(s + str(num))
```

> ⇨ **10**
> ⇨ **hello10**

5. string methods

```
s1="HelloTarif"
print(s1.lower())
```

```python
print(s1.upper())
#Converts all characters in the string to uppercase.
print(s1.strip())
#Removes leading and trailing whitespaces from the string.
print(s1.isalpha())
#Checks if all characters in the string are alphabetic.
print(s1.isdigit())
#Checks if all characters in the string are digits.
print(s1.isspace())
#Checks if all characters in the string are whitespaces.

print(s1.replace(s1,'old'))
```
#Replaces the entire string with the string 'old' (this might not be the intended behavior; it seems like you might want to replace a part of the string).
```python
print('a,b,c'.split(','))
```
*#**Splits the string** 'a,b,c' into a list ['a', 'b', 'c'] based on the comma delimiter.*
```python
print(':'.join(['a','b','c']))
```
*#**Joins the elements** of the list ['a', 'b', 'c'] into a string using the colon (':') as a delimiter.*

⇨ **hellotarif**
HELLOTARIF
HelloTarif
True
False
False
old
['a', 'b', 'c']
a:b:c

6. strings slice

```python
s="Hello"
print(s[1:3])
```
*#Prints characters from index 1 to index 2 (**excluding 3**). The result is "el".*
```python
print('\n')
```
*#The `'\n'` is used to print a **newline character**, which creates a blank line between the outputs for better readability.*
```python
print(s[1:])
print('\n')

print(s[-4:3])
```

⇨ `el`

⇨

⇨ `ello`

⇨

⇨ `el`

7. strings tuple of arguments (this is old style)

```
str1='hello %s, %d' % ('world',10)
print(str1)
```

⇨ `hello world, 10`

Explanation:

example of old-style string formatting in Python. In the old style, the `%` operator is used to format a string with values provided in a tuple on the right side of the `%` operator.

`'%s'` and `'%d'` are placeholders for string and integer values, respectively.

8. strings format (this is new style)

```
str2='Hello {}, {}'.format('world',10)
print(str2)
```
*#**Basic String Formatting.** This uses placeholders `{}` in the string and fills them in order with the values provided in the `format()` method.*

```
str3 = 'Hello {wd}, {num}'.format(num=10, wd='World')
print(str3)
```
*#**Named Placeholder Formatting.** Here, named placeholders `{wd}` and `{num}` are used, and their values are specified in the `format()` method using keyword arguments.*

```
str4 = 'Hello {wd:10}, {num:.3f}'.format(num=10.0,
wd='World')
print(str4)
```

*#**Formatting with Width and Precision.** This example shows how to format with width and precision. `{wd:10}` specifies a **minimum width** of 10 characters for the `wd` placeholder, and `{num:.3f}` specifies a floating-point number with **3 digits** after the decimal point for the `num` placeholder.*

```
⇨ Hello world, 10
Hello World, 10
Hello World    , 10.000
```

9. Formatted string literals

```python
w = 'World'
num = 10
print(f'Hello {w}, {num}')
print(f'Hello {w}, {num + num}')
```

```
⇨ Hello World, 10
  Hello World, 20
```

Explanation:

Your code is using formatted string literals, often referred to as f-strings. This feature was introduced in Python 3.6, providing a concise and readable way to format strings.

f'Hello {w}, {num}': This is an f-string where expressions inside curly braces **{}** are evaluated and replaced with their values. In this case, **{w}** is replaced with the value of the variable **w** ('World'), and **{num}** is replaced with the value of the variable **num** (10).

f'Hello {w}, {num + num}': Here, an expression **{num + num}** is evaluated, and the result (20) is inserted into the string.

10. If statement

```python
num = 40
if num < 5:
  print('num is smaller than 5')


if num > 5:
  print('num is greater than 5')
```

```python
elif num == 5:
    print('num is 5')

else:
    print('num is smaller than 5')
```

⇨ **num is greater than 5**

11. Function Calls

```python
print(type(32))
#Prints the type of the number 32. The output will be <class 'int'>.
type(32)
print(int(32))
#Converts the number 32 to an integer using the int() function and prints it.

#int('Hello')
# If you want to convert a string to an integer, the string must contain a valid
integer representation. it will raise a ValueError.

int(-2.3)
print(int(-2.3))
#The output is -2 because the decimal part of -2.3 is truncated, and only the whole
number part is retained. It's important to note that the conversion doesn't round
the number; it simply removes the decimal part towards zero.

import math
# Imports the math module.
print(math.sin(3.1416))
print(math.cos(3.1416))
print(math.tan(3.1416))
print(math.acos(-1))
print(math.sqrt(2))

#help('math')   # try this too
```

⇨ **<class 'int'>**
    32
    -2
    -7.346410206643587e-06
    -0.9999999999730151
    7.346410206841829e-06

```
3.141592653589793
1.4142135623730951
```

12.def function_name(parameters): BLOCK

```python
def print_name():
  print("I'm Raja")
  print("I'm Blah Blah Blah")

print_name()
```
⇨ **I'm Raja**
   I'm Blah Blah Blah

Explanation:

`def print_name():`: This line defines a function named `print_name` with no parameters. It consists of two `print` statements that will display messages when the function is called. The colon `:` indicates the start of the function block.

13. Function parameters and arguments

```python
def print_twice(arg1):
  print(arg1)
  print(arg1)

print_twice('Spam' * 4)
```

⇨ **SpamSpamSpamSpam**
   SpamSpamSpamSpam

If you intend to use **math.pi (for printing pi twice using this function)**, you need to import the **math** module first. See below:

```python
# 13.1 Function parameters and arguments
import math
```

```python
def print_twice(arg1):
  print(arg1)
  print(arg1)

print_twice(math.pi)
```

```
=> 3.141592653589793
   3.141592653589793
```

14. Function without Returns

```python
def print_twice(arg1):
  print(arg1)
  print(arg1)

result = print_twice('Bing')
print(result)
```
```
⇨ Bing
  Bing
  None
```

In this case, **print_twice** doesn't explicitly return a value (implicitly returns **None**). The function can perform actions, such as printing, but it doesn't provide a result that can be assigned to a variable or used in expressions elsewhere.

14.1. With 'return'

```python
def print_twice(arg1):
    print(arg1)
    print(arg1)
    return arg1 * 2

result = print_twice('Bing')
print(result)
```
```
 => Bing
    Bing
    BingBing
```

Here, **print_twice** not only performs actions (printing) but also returns a value (**arg1 * 2**). This returned value can be assigned to a variable (**result** in this case) and used elsewhere in your code.

15. Function Returns tuple

```python
def print_twice(arg1,arg2):
```
#*Defines a function named* **print_twice** *that takes two arguments*

```python
    return arg1,arg2
```
#*The function returns a tuple containing the values of* **arg1** *and* **arg2**.

```python
res1,res2 = print_twice('Bing','soumys')
```
#*Calls the function with arguments* **'Bing'** *and* **'soumys'** *and unpacks the returned tuple into the variables* **res1** *and* **res2**.

```python
print(res1)
print(res2)
print(res1, res2)
```

=> **Bing**

   soumys

   Bing soumys

Explanation:

The function **print_twice** returns a tuple. A tuple is a data structure in Python that is similar to a list but is <mark>immutable</mark>, meaning its elements cannot be changed after it is created. In this case, the function returns a tuple containing two elements: **arg1** and **arg2**.

The line **return arg1, arg2** is equivalent to **return (arg1, arg2)**. The parentheses are optional when returning multiple values, and Python automatically creates a tuple.

The returned tuple **('Bing', 'soumys')** <mark>is automatically unpacked into the variables</mark> **res1** and **res2**. As a result, **res1** gets the value **'Bing'**, and **res2** gets the value **'soumys'**. Then, you print these values individually.

16. 'from' function

In Python, when you import a module using **import math**, you need to reference items from that module using the module name as a prefix,

like `math.pi`. If you want to directly import specific items from a module without needing to use the module name as a prefix, you can use the `from` keyword. E.g. `from math import pi, cos`

16.1 Importing without 'from'

```python
import math

print(math)
#output indicates that math is a built-in module in Python. It's part of the standard library, and <module 'math' (built-in)> is just a representation of the math module object.

print(math.pi)

#print(pi)
#This will not work here without math. Prefix. Shows "NameError: name 'pi' is not defined"
```

```
=> <module 'math' (built-in)>
   3.141592653589793
```

16.2 Importing with 'from' function

```python
from math import pi

print(pi)

from math import *
# In Python, the * character, when used in the context of importing with the from statement, is a wildcard that means "import everything." It is often referred to as a "star import." You are indeed importing all names from the math module into the current namespace using the wildcard *.
cos(pi)
```

```
=> 3.141592653589793
   -1.0
```

17. Function Checking type of arguments-recursive

```python
def factorial (n):
  if not isinstance(n, int):
    print('Factorial is only defined for integers.')
    return None
```
*# isinstance(n, int): Checks if **n** is an integer. If it's not, it prints an error message and returns **None**.*
```python
  elif n < 0:
   print('Factorial is not defined for negative integers.')
    return None
  elif n == 0:
    return 1
  else:
    return n * factorial(n-1)

factorial(5)
```
⇨ **120**

18. Default arguments/Function overloading

```python
def func1(arg1, arg2=None):
  if arg2 is None:
    print(arg1)
  else:
    print(arg1, arg2)

func1('hello')

func1('hello', 'world')
```

⇨ `hello`
  `hello world`

Explanation:

In the definition of **func1**, **arg2=None** makes **arg2** an optional parameter, and its default value is **None**. When you call the function without providing a value for **arg2** (as in **func1('hello')**), it uses the

default value, and the condition **if arg2 is None:** is true. In this case, it prints only **arg1**.

When you call the function with both arguments (as in **func1('hello', 'world')**), it prints both **arg1** and **arg2**.

19. Global/Local variables

```
global_var = 1

def func1():
  global_var = 2  # This is not a global variable,
but a local variable.

func1()
global_var
```

⇨ **1**

Explanation:

You have a global variable global_var with a value of 1, and you define a function func1 that has a local variable with the same name global_var with a value of 2.

When you call func1(), it doesn't modify the global variable global_var. Instead, it creates a new local variable with the same name that exists only within the scope of the function.

After calling func1(), if you print global_var, it will still refer to the global variable outside the function. Therefore, the output will be 1.

If you want to modify the global variable within the function, you need to use the **global** keyword:

```
global_var = 1

def func1():
```

```
  global global_var
  global_var = 2

func1()
# Remember one thing. This ' func1()' is used to 'call' the function. It is necessary.
If you comment out this line, it will still show output as 1.
global_var
```

⇨ **2**

20. Some math functions:

```python
import math

a = 2
b = 5
c = (a**2 + b**2)**0.5
d = math.sqrt(c)

print('hypoteneous length =', c)
print('sq root d =', d)

e = math.log(d)
f = c * d
g = c / d
h = a * math.sin(b)

print('multiplication =', f)
print('division =', g)
print('sin func =', h)
```

⇨ **hypoteneous length = 5.385164807134504**
   sq root d = 2.320595787106084
   multiplication = 12.496790764308276
   division = 2.3205957871060834
   sin func = -1.917848549326277

20.1. Arguments of Trigonometric functions are in radian as default.

In Python's **math.sin** function, the argument is expected to be in radians. Therefore, if you want to calculate the sine of an angle given in degrees, you need to convert the angle from degrees to radians using the **math.radians** function.

So, in the line:

```
h = a * math.sin(b)
```

If **b** represents an angle in degrees, you should modify it like this:

```
h = a * math.sin(math.radians(b))
```

==This ensures that the **math.sin** function receives the angle in radians==, which is the expected format.

21. Modules in python

numpy:

This module provides a variety of mathematical functions, such as ==array manipulation, linear algebra, and Fourier transforms.==

Specifically, `numpy.linalg` module provides a variety of linear algebra functions, such as matrix multiplication, inversion, and eigenvalue decomposition.

```python
import numpy as np
import numpy.linalg as LA
# This imports NumPy and renames the numpy.linalg module to LA for convenience.

# Create a NumPy matrix and print that
A = np.array([[1, 2], [3, 4]])
print('A=',A)

# Calculate the inverse of the matrix
B = LA.inv(A)
```

```
# Print the inverse matrix
print('B=',B)
```

```
A= [[1 2]
 [3 4]]
B= [[-2.    1. ]
 [ 1.5 -0.5]]
```

os.MFD_HUGE_256MB:

This constant represents the size of a huge memory file descriptor, which is 256 MB. 'os' module provides functions for interacting with the operating system, such as creating and deleting files and directories, opening and closing files, and running other programs.

scipy:

This module provides a variety of scientific computing functions, such as statistical analysis, signal processing, and optimization.

```
import numpy as np
import scipy.stats as stats
# Imports the stats submodule from the SciPy library and renames it to stats.

# Create a NumPy array of random numbers
data = np.random.randn(1000)
# Generates an array of 1000 random numbers from a standard normal
distribution. Each number in this array is drawn independently from a Gaussian
distribution with a mean (average) of 0 and a standard deviation of 1.

# Calculate the mean and standard deviation of the
data
mean = stats.tmean(data)
std = stats.tstd(data)

# Print the results
print("Mean:", mean)
print("Standard deviation:", std)
```

⇨ **Mean: 0.01063247408029411**

```
Standard deviation: 0.9726377211855272
```

#22. Matrix operations

```python
import numpy as np
from scipy.linalg import inv
from numpy import array
```
 # You may use from numpy import *. But, we import the `array` *function from NumPy. Note that wildcard imports (*) are avoided to reduce the risk of naming conflicts.*

```python
M1 = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
M2 = array([[1, 4, 3], [4, 7, 6], [7, 12, 9]])

M3 = np.dot(M1, M2)
```
 # It's important to note that the result of **A * B** *is an element-wise multiplication, not the standard matrix multiplication. If you want to perform standard matrix multiplication, you should use* **np.dot(M1, M2)** *or the M1* **@ M2** *operator.*

```python
M4 = M1[1]
M5 = M1 * M4
```
 # This is not a proper matrix multiplication. Element-wise multiplication, but note that M4 is a row vector. So, this is of no-meaning.

```python
M6 = inv(M2)
```
 # Use the inv function from scipy.linalg for matrix inversion

```python
print('\n M1=', M1)
print('\n M2=', M2)
print('\n M3=', M3)
print('\n M4=', M4)
print('\n M5=', M5)
print('\n M6=', M6)
```

⇨ **M1= [[1 2 3]**
  [4 5 6]
  [7 8 9]]

  M2= [[1 4 3]
  [4 7 6]
  [7 12 9]]

```
M3= [[ 30  54   42]
 [ 66 123   96]
 [102 192 150]]

M4= [4 5 6]

M5= [[ 4 10 18]
 [16 25 36]
 [28 40 54]]

M6= [[-7.50000000e-01  2.96059473e-16
 2.50000000e-01]
 [ 5.00000000e-01 -1.00000000e+00  5.00000000e-01]
 [-8.33333333e-02  1.33333333e+00 -7.50000000e-01]]
```

23. Bassel functions- theory

Bessel functions are a family of mathematical functions that are solutions to Bessel's differential equation. The Bessel differential equation is given by:

$$x^2 y'' + x y' + (x^2 - n^2) y = 0$$

where y'' is the second derivative of y with respect to x, y' is the first derivative, n is a parameter, and x is the independent variable.

The solutions to the Bessel differential equation are called Bessel functions of the first kind, and they are denoted by $J_n u(z)$. There are also other types of Bessel functions, such as Bessel functions of the second kind, $Y_n u(z)$, and modified Bessel functions, $I_n u(z)$ and $K_n u(z)$.

Bessel functions can be expressed in a variety of ways, including infinite series, integrals, and continued fractions. They can also be computed numerically using a variety of algorithms.

Bessel Functions of the First Kind ($J_n(x)$):

For a given non-negative integer n, the Bessel function of the first kind $J_n(x)$ is defined by the series expansion:

$$J_n(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k!\,(n+k)!} \left(\frac{x}{2}\right)^{2k+n}$$

This series converges for all values of x, making $J_n(x)$ well-defined for all real x and non-negative integers n.

Bessel Functions of the Second Kind ($Y_n(x)$):

The Bessel function of the second kind $Y_n(x)$ is related to $J_n(x)$ and is defined as:

$$Y_n(x) = \frac{J_n(x)\cos(n\pi) - J_{-n}(x)}{\sin(n\pi)}$$

where $J_{-n}(x)$ is the Bessel function of the first kind with order $-n$.

Here are some mathematical properties of Bessel functions:

Bessel functions of the first kind are continuous and infinitely differentiable for all z and nu.

Bessel functions of the first kind are oscillatory functions, meaning that they have an infinite number of zeros.

The zeros of Bessel functions of the first kind are interlaced, meaning that the zeros of $J_{nu+1}(z)$ are always located between the zeros of $J_{nu}(z)$.

Bessel functions of the first kind satisfy the following recurrence relation:

$$J_{\nu+1}(z) - \frac{2\nu}{z} J_\nu(z) + J_{\nu-1}(z) = 0$$

This is a second-order linear homogeneous recurrence relation. The terms $J_{\nu+1}(z)$, $J_\nu(z)$, and $J_{\nu-1}(z)$ are successive Bessel functions of the first kind with orders $\nu + 1$, ν, and $\nu - 1$, respectively.

This recurrence relation is derived from the Bessel's differential equation and is a useful tool for computing Bessel functions numerically. It provides a way to compute higher-order Bessel functions based on lower-order ones.

Physical Interpretation:

In physical terms, these functions often arise in problems involving circular or cylindrical symmetry. For example:

Vibrations of Circular Membranes: In the context of a circular drumhead, $J_n(x)$ describes the radial displacement of the drumhead at position x for a given mode n.

Heat Conduction: In the context of heat conduction in a cylindrical object, $J_n(x)$ can describe the temperature distribution at a radial position x for a given mode n.

The Schrödinger equation for a hydrogen atom.

The Helmholtz equation in spherical coordinates.

Visualization:

For more detail, visit
https://mathworld.wolfram.com/BesselFunctionoftheFirs
tKind.html
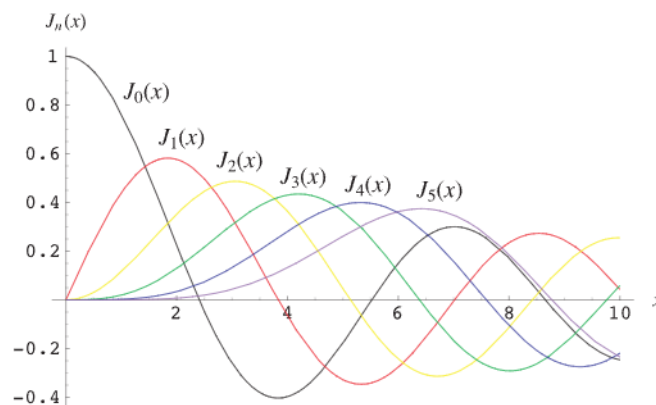
24. Plot Bessel function of 1ˢᵗ and 2ⁿᵈ type:

```python
import numpy as np

import matplotlib.pyplot as plt

from scipy.special import jn, yn
```
*#Import necessary libraries. **numpy** for numerical operations, **matplotlib**.**pyplot** for plotting, and **jn** and **yn** functions from **scipy**.**special** for Bessel functions of the first and second kinds.*

```python
# Define parameters
x = np.linspace(0, 20, 100)   # Values of x
n_values = [0, 1, 2, 3]   # Orders of Bessel functions

# Plot Bessel functions of the first kind (J_n(x))
plt.figure(figsize=(12, 6))
```
#creating a figure using Matplotlib in Python with a specified figure size of 12 inches in width and 6 inches in height.

```python
plt.subplot(1, 2, 1)
```
#Create the first subplot in a 1x2 grid (1 row, 2 columns), and set the current subplot to the first one.

```python
for n in n_values:
    plt.plot(x, jn(n, x), label=r"$J_{%d}(x)$" % n)
```
#Loop through the orders in `n_values` and plot Bessel functions of the first kind for each order. The `label` argument is used for legend labeling. `plt.plot()` function is typically used to plot a line graph.

 # `jn(n, x)`: This is the Bessel function $J_n(x)$ with order n evaluated at the given x-values.

#`label=r"$J_{%d}(x)$" % n:` This is a raw string (indicated by the `r` prefix) that uses LaTeX-style formatting. The `%d` is a placeholder for an integer, which will be replaced by the value of `n`.

```python
plt.title("Bessel Functions of the First Kind ($J_n(x)$)")
plt.xlabel("x")
plt.ylabel("$J_n(x)$")
plt.legend()
```

```
plt.grid(True)
```
#Add title, xlabel, ylabel, legend, and grid to the first subplot.

```
# Plot Bessel functions of the second kind (Y_n(x))
plt.subplot(1, 2, 2)
for n in n_values:
    plt.plot(x, yn(n, x), label=r"$Y_{%d}(x)$" % n)

plt.title("Bessel Functions of the Second Kind
($Y_n(x)$)")
plt.xlabel("x")
plt.ylabel("$Y_n(x)$")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```
#Without `tight_layout()`*, subplots might overlap or not be evenly spaced, especially when you have multiple subplots.*
#The `show()` *function is used to display the plot. When you run a Matplotlib script in interactive mode, you don't always need to call* `show()` *explicitly because the plot will be displayed automatically. However, in certain environments (like scripts or non-interactive modes), you may need to call* `show()` *to display the plot.*
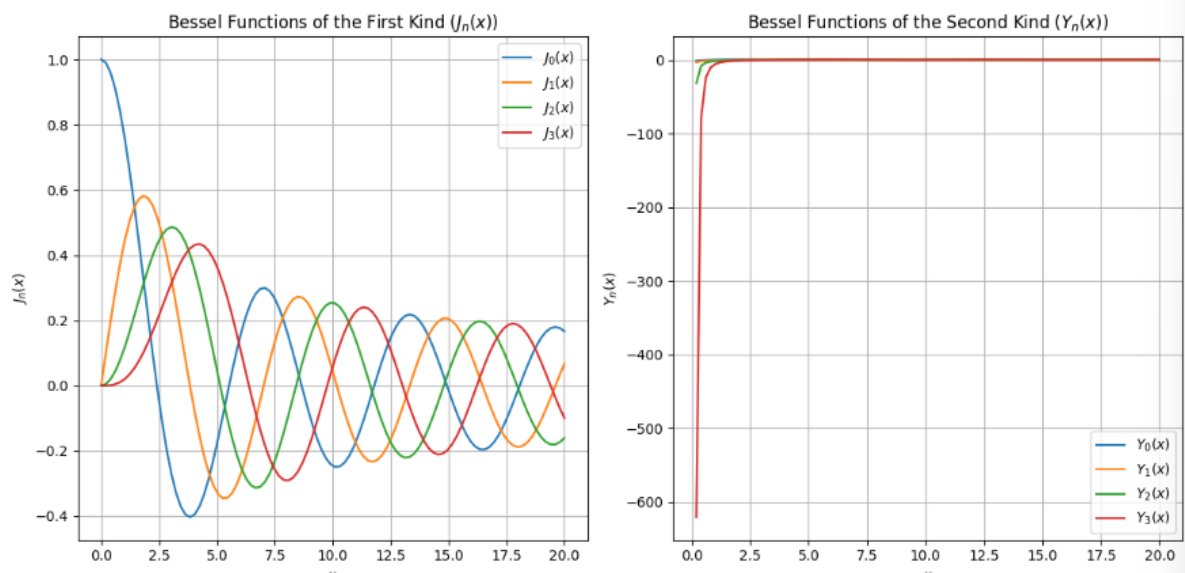
25. Integrate f(x) between limits, ∫baf(x)dx

```
 ##Integrate f(x) between limits, $\displaystyle
\int_a^b f(x) dx$
```

```python
import numpy as np
```

```python
import matplotlib.pyplot as plt
from scipy.integrate import quad
```
*#`quad` function from `scipy.integrate` for **numerical** integration. See 25.2 for more.*

```python
def f(x):
    a = 1
    return a * x**2
```
#Define a function `f(x)` that represents the function $f(x)=x^2$.(though, local variable a is not necessary to include, as its value=1).

```python
# Define integration bounds
x_lower = 0
x_upper = 1
```

```python
# Perform the integration
val, abserr = quad(f, x_lower, x_upper)
print("Integral =", val, ", Absolute error =",
abserr)
```
#Use the `quad` function to perform numerical integration of the function $f(x)=x^2$ over the specified bounds (`x_lower` to `x_upper`). The result (`val`) is the definite integral, and `abserr` is the estimated absolute error.

```python
# Plot the function
n = 256
X = np.linspace(x_lower, x_upper, n, endpoint=True)
Y = f(X)
```
#This line uses NumPy's `linspace` function to create an array of `n` equally spaced points between `x_lower` and `x_upper`, inclusive. The `endpoint=True` argument specifies that the endpoint (i.e., `x_upper`) should be included in the array.

```python
# Plotting
fig, ax = plt.subplots()  #See below
```

```python
ax.plot(X, Y, label='f(x) = x^2')
```
 # The `label` parameter is set to 'f(x) = x^2' for use in the legend.

```python
ax.fill_between(X, 0, Y, alpha=0.2, color='blue')
```

```
ax.legend()
ax.set_title('Plot of f(x) = x^2')
ax.set_xlabel('x')
ax.set_ylabel('f(x)')

# Additional plotting commands
plt.plot(X, Y, '-', color='r', linewidth=6)
```
 # `'-'` represents a solid line. Other line styles include `'--'` for dashed lines, `':'` for dotted lines, and `'-.'` for dash-dot lines.
 # `'r'` represents red. You can use various color representations such as `'b'` for blue, `'g'` for green, `'k'` for black, `'c'` for cyan, and so on. Additionally, you can specify colors using RGB values like `(0.8, 0.2, 0.1)`.

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

Integral = 0.33333333333333337 , Absolute error = 3.700743415417189e-15

Plot of f(x) = x^2
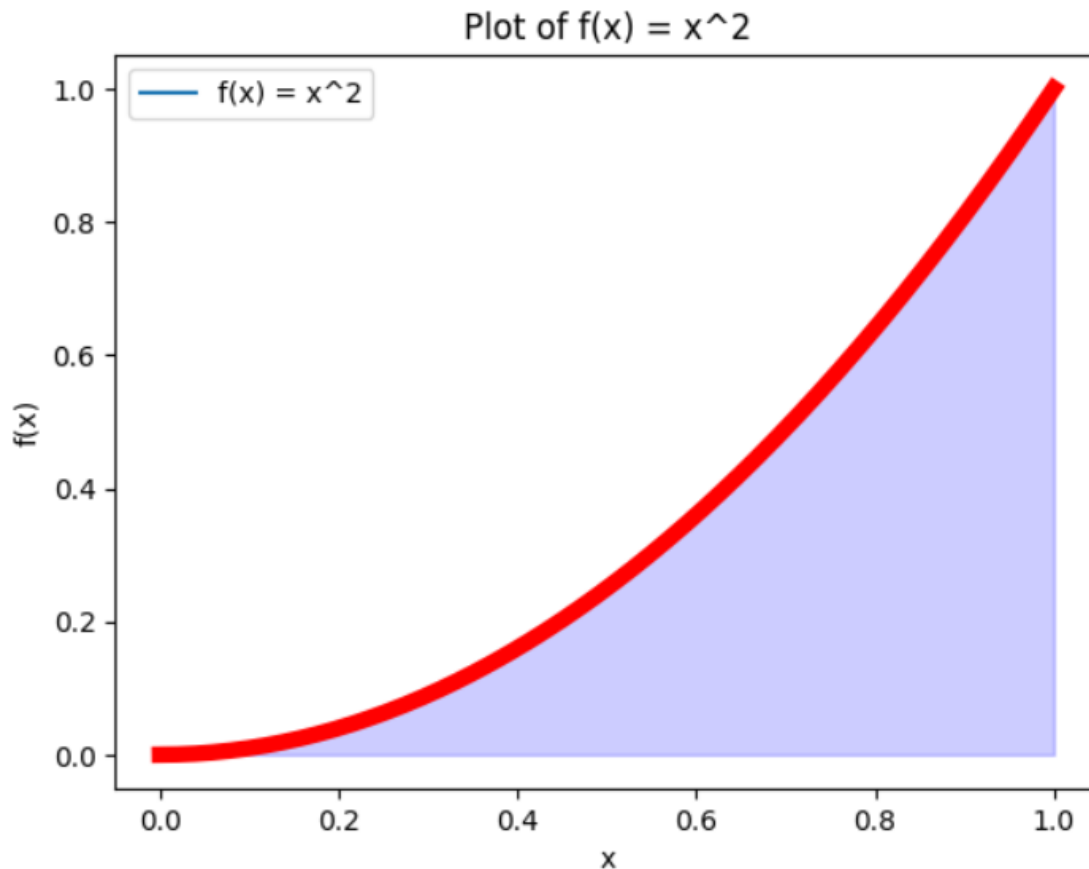
25.1 fig, ax = plt.subplots()

The line `fig, ax = plt.subplots()` creates a new figure (fig) and a single set of subplots (ax). In other words, it creates a new window and a single plot within that window. This line is using Python's tuple unpacking to assign the two returned objects (**fig** and **ax**) to the variables **fig** and **ax**.

The `plt.subplots()` function takes two optional arguments: `nrows` and `ncols`, which specify the number of rows and columns of the subplot grid. If only one argument is provided, then a single subplot is created. If two arguments are provided, then a grid of subplots is created.

In the example you provided, the `plt.subplots()` function is called with no arguments. This means that a single subplot will be created. The function returns two objects: a figure object (fig) and an axes object (ax).

The figure object represents the entire window, while the axes object represents the single plot within the window.

Once the figure and axes objects have been created, you can use the **ax** object to plot data, set labels, titles, and customize the appearance of your plot.

25.2 dblquad, tplquad meaning

In Python, particularly in the context of scientific computing and numerical integration, **dblquad** and **tplquad** refer to functions provided by the SciPy library for double and triple integration, respectively.

1. **dblquad**:

The **dblquad** function is used for double integration. It has the following syntax:

```
scipy.integrate.dblquad(func, a, b, gfun, hfun)
```

- **func**: The integrand function of two variables.

- **a**, **b**: The lower and upper limits of integration for the variable x.

- **gfun**, **hfun**: These are functions defining the lower and upper limits of integration for the variable y as functions of x.

Here's an example:

```python
from scipy import integrate

def integrand(x, y):
    return x * y

result, error = integrate.dblquad(integrand, 0, 1,
lambda x: 0, lambda x: 2)
print(result)
    ⇨ 0.9999999999999999
```

This example computes the double integral of xy over the region where $0 \leq x \leq 1$ and $0 \leq y \leq 2$.

2. **tplquad**:

The **tplquad** function is used for triple integration. Its syntax is as follows:

```
scipy.integrate.tplquad(func, a, b, gfun, hfun, qfun,
rfun)
```

- **func**: The integrand function of three variables.

- **a**, **b**: The lower and upper limits of integration for the variable *x*.

- **gfun**, **hfun**: Functions defining the lower and upper limits of integration for the variable *y* as functions of *x*.

- **qfun**, **rfun**: Functions defining the lower and upper limits of integration for the variable *z* as functions of *x* and *y*.

Here's an example:

```
from scipy import integrate

def integrand(x, y, z):
    return x * y * z

result, error = integrate.tplquad(integrand, 0, 1,
lambda x: 0, lambda x: 2, lambda x, y: 0, lambda x,
y: 3)
# 0, 1: These are the lower and upper limits of integration for the variable x.
lambda x: 0, lambda x: 2: These are the functions defining the lower and upper
limits of integration for the variable y. The limits are expressed as functions of x.
lambda x, y: 0, lambda x, y: 3: These are the functions defining the lower
and upper limits of integration for the variable z. The limits are expressed as
functions of both x and y.

print(result)
```
   ⇨ **4.5**

This example computes the triple integral of *xyz* over the region where $0 \leq x \leq 1$, $0 \leq y \leq 2$, and $0 \leq z \leq 3$.

These functions are useful when you need to numerically integrate functions of multiple variables over specified regions.

26. Integration of Bassel Function.

```python
import numpy as np
from scipy.integrate import quad
from scipy.special import jn
```
 # Importing necessary libraries for numerical operations, numerical integration and Bessel functions.

```python
# Define the Bessel function to integrate
def integrand(x):
    n = 0    # Replace with the desired order of the Bessel function. Here, I
```
took 0th order.
```python
    return jn(n, x)

# Define integration bounds
x_lower = 0
x_upper = 10 # Adjust the upper limit as needed

# Perform the integration
result, error = quad(integrand, x_lower, x_upper)
```
 # The `quad` function from `scipy.integrate` is used to perform numerical integration. It takes the integrand function (`integrand`) and the integration bounds (`x_lower` and `x_upper`). The result of the integration is stored in the variables `result` and `error`. `result` contains the estimated value of the integral, and `error` contains an estimate of the absolute error.

```python
# Print the result
print(f"The result of the integration is: {result}")
print(f"Estimated error: {error}")

# Plot the Bessel function and the area under the
curve (for explanation, see 25)
x_values = np.linspace(x_lower, x_upper, 1000)
y_values = integrand(x_values)

fig, ax = plt.subplots()
ax.plot(x_values, y_values, label=f'$J_{n}(x)$ for
n={n}')
ax.fill_between(x_values, 0, y_values, alpha=0.2,
label='Area under the curve')
ax.legend()
ax.set_title('Integration of Bessel Function')
ax.set_xlabel('x')
```

```
ax.set_ylabel(f'$J_{n}(x)$')

plt.show()
```

```
The result of the integration is: 1.0670113039567362
Estimated error: 7.434789460651883e-14
```
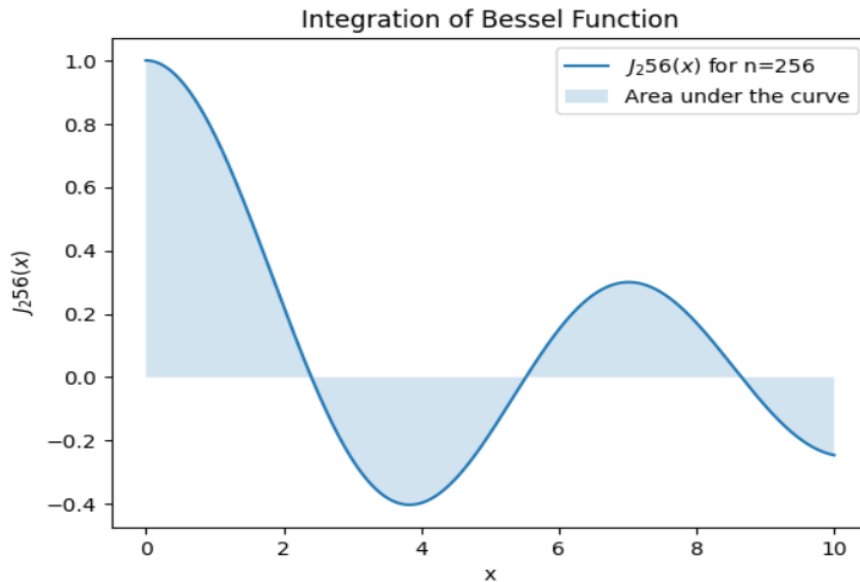

27. Double Integration

Evaluate the expression: $\iint_0^{10} {}_0^{10} e^{(-x^2-y^2)}dx\ dy$

```python
from math import *
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import Image
from scipy import *
import scipy.linalg as la
from scipy.integrate import quad, dblquad, tplquad
```
#Above this, is the block of libraries mostly used in scientific computing as told by professor. Though everything is not necessary for this code.

```python
def integrand(x,y):
  return exp(-x**2-y**2)

x_lower=0; x_upper=10
y_lower=0; y_upper=10

val, abserr = dblquad(integrand, x_lower, x_upper,
lambda x:y_lower, lambda x:y_upper)
```

#Here, the limits of integration for y are defined using lambda functions. Lambda functions are a way to create small, unnamed functions in Python. The general syntax for a lambda function is: **`lambda arguments: expression`**

```
print('value of Integration=',val,'absolute
error=',abserr)
```

⇨ **`Value of Integration= 0.7853981633974476`**
`Absolute Error= 1.3753098510194181e-08`

27.1 what is 'pandas'?

pandas is a popular open-source data manipulation and analysis library for Python. It provides data structures like Series and DataFrame, *designed for efficient and intuitive handling and analysis of structured data*. The library is widely used in data science, machine learning, and analytics.

Here are some key features and capabilities of Pandas:

1. **DataFrame**: A two-dimensional, tabular data structure with labeled axes (rows and columns). It is similar to a spreadsheet or SQL table.
2. **Series**: A one-dimensional labeled array, similar to a column in a DataFrame or a single variable in statistics.
3. **Data Cleaning and Preparation**: Pandas provides a wide range of functions for cleaning and preparing data, including handling missing values, reshaping data, and transforming data.
4. **Data Analysis and Exploration**: It supports various operations for analyzing and exploring data, such as filtering, grouping, aggregation, merging, and sorting.
5. **Handling Time Series Data**: Pandas has functionality for working with time series data, making it suitable for applications in finance, economics, and other domains dealing with time-ordered data.

6. **Input/Output**: Pandas supports reading and writing data in various formats, including CSV, Excel, SQL databases, and more.

Here's a simple example of using Pandas to create a DataFrame:

```python
import pandas as pd

# Creating a DataFrame
data = {'Name': ['John', 'Alice', 'Bob'],
        'Age': [28, 24, 22],
        'City': ['New York', 'Los Angeles',
'Chicago']}

df = pd.DataFrame(data)

# Displaying the DataFrame
print(df)
```
⇨
```
    Name  Age         City
0   John   28     New York
1  Alice   24  Los Angeles
2    Bob   22      Chicago
```

If you print without pandas this will look bad.

```python
data = {'Name': ['John', 'Alice', 'Bob'],
        'Age': [28, 24, 22],
        'City': ['New York', 'Los Angeles',
'Chicago']}

print(data)
```
⇨ {'Name': ['John', 'Alice', 'Bob'], 'Age': [28, 24, 22], 'City': ['New York', 'Los Angeles', 'Chicago']}

27.2 from IPython.display import Image

IPython: IPython is an interactive command-line shell for Python. It provides features such as interactive computing, easy-to-use syntax,

and enhanced introspection capabilities. IPython is often used in scientific computing, data analysis, and machine learning.

IPython.display module: This module within IPython provides tools for displaying various types of content in the IPython environment. It includes classes and functions for rendering ==multimedia content like images, audio, video, HTML,== etc.

Image class: The **Image** class is part of the **IPython.display** module, and it is used to display images in the IPython environment. You can create an instance of the **Image** class and then display it using IPython's rich display capabilities.

```python
from IPython.display import Image

# Get the path to the image file
image_path = '/home/sayak/DEsktop/flower.jpeg'

# Display the image
Image(image_path)
```

Not working, some problem happened???!! Recheck!

27.3 Compare NumPy, SciPy and math

NumPy:

1. **Core Functionality**:
   - Provides support for large, multi-dimensional arrays and matrices (**numpy.ndarray**).
   - Essential for numerical operations and efficient array manipulation.
   - Fundamental building block for numerical computing in Python.
2. **Array Operations**:

- Element-wise operations on arrays (addition, subtraction, multiplication, etc.).
- Broadcasting: Operations between arrays of different shapes and sizes.

3. **Linear Algebra**:
   - Basic linear algebra operations (e.g., matrix multiplication, eigenvalue decomposition).
   - **numpy.linalg** module provides linear algebra functions.

4. **Random Number Generation**:
   - **numpy.random** module for generating random numbers and distributions.

SciPy:

1. **Extended Functionality**:
   - Extends NumPy with additional modules for scientific computing.
   - Covers a broader range of scientific and numerical algorithms.

2. **Integration and Differentiation**:
   - Numerical integration tools **scipy.integrate** (which contains **quad**, **dblquad**, **tplquad**).
   - Symbolic and numerical differentiation (**scipy.diff**).

3. **Optimization**:
   - Optimization algorithms for finding minima or roots of functions (**scipy.optimize**).

4. **Signal and Image Processing**:
   - Signal processing tools (**scipy.signal**).
   - Image processing functions (**scipy.ndimage**).

5. **Statistical Functions**:
   - Additional statistical functions beyond what NumPy provides (**scipy.stats**).

6. **Interpolation**:
   - Interpolation algorithms (**scipy.interpolate**).

7. **Special Functions**:
   - Mathematical special functions (**scipy.special**).

8. **Sparse Matrices**:

- Sparse matrix support (**scipy.sparse**).

math Library (Built-in):

1. **Basic Mathematical Functions**:
   - Provides basic mathematical functions such as trigonometry, logarithms, exponentials, etc.
2. **Single-Value Operations**:
   - Operates on single values, not arrays or matrices.
3. **Pure Python Implementation**:
   - Part of the Python Standard Library and implemented in pure Python.
4. **Limited to Scalars**:
   - Designed for scalar operations and not optimized for array or matrix operations.

Common Use Cases:

- **NumPy**:
  - Essential for numerical and array-based computing.
  - Used in applications involving linear algebra, statistical analysis, and machine learning.
- **SciPy**:
  - Applied in scientific research and engineering for a wide range of numerical and computational tasks.
  - Used for solving complex mathematical problems in physics, chemistry, biology, and more.
- **math Library**:
  - Simple mathematical operations in a scalar context.
  - Suitable for basic math computations in non-array contexts.

Relationship:

- **Interdependence**:
  - SciPy builds on NumPy and often relies on NumPy arrays as its underlying data structure.
  - Both libraries are typically used together in scientific computing applications.

- **math library** is a lightweight built-in library suitable for basic scalar mathematical operations.

In summary, NumPy provides fundamental array functionality and core numerical operations, while SciPy extends this functionality to cover a broader range of scientific computing tasks, including optimization, integration, signal processing, and more. The two libraries are designed to be used together to provide a comprehensive set of tools for numerical and scientific computing in Python.

28. Some more matrix operations

```python
from scipy import *
import numpy as np
from math import *
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import Image
# Importing various libraries for numerical computing (numpy and scipy), basic
# mathematical operations (math), data manipulation (pandas), plotting
# (matplotlib.pyplot), and displaying images in IPython.

V1=np.array([1,2,4,5])
print('\n V1=',V1)


M1=np.array([[1,complex(2,0),3],[3,4,5],[0,9,2]])
print('\n M1=',M1)
# array M1 with a mix of integers and complex numbers.

dd=M1[:,1]
print('\n dd=', dd)
#Extracting data: Slicing the second column of array M1 and printing it. In
# Python and many programming languages, including NumPy, indexing typically
# starts from 0.

A=np.array([[6,1,1],[4,-2,5],[2,8,7]])
print('inverse=',np.linalg.inv(A)) #inverse

M1=np.random.random((3,3))
```

```
# you've created a 3x3 NumPy array named M1 filled with random values between
0 and 1 using the np.random.random function. Each element in the array will be a
random float in the half-open interval [0.0, 1.0).
ee=linalg.eigvals(M1)
print('\n eigenvalue=',ee)    #eigenvalue of random
matrix


#from scipy.linalg import svd
#(this line is not necessary, because already everything imported from scipy)
M1=np.random.random((5,5))
U, s, VT = svd(M1)
print('U=',U)
```

⇨
```
 V1= [1 2 4 5]

 M1= [[1.+0.j 2.+0.j 3.+0.j]
 [3.+0.j 4.+0.j 5.+0.j]
 [0.+0.j 9.+0.j 2.+0.j]]

 dd= [2.+0.j 4.+0.j 9.+0.j]
 inverse= [[ 0.17647059 -0.00326797 -0.02287582]
 [ 0.05882353 -0.13071895  0.08496732]
 [-0.11764706  0.1503268   0.05228758]]

 eigenvalue= [ 1.34895175+0.j -0.52664035+0.j -0.11203187+0.j]
 U= [[-0.27000824  0.02194517 -0.4610434   0.19566695 -0.82205072]
 [-0.51609845 -0.83111595 -0.02787021 -0.16347384  0.12404928]
 [-0.42279365  0.43326529 -0.47398082 -0.57517509  0.27936089]
 [-0.42655014  0.24485939  0.74879153 -0.28472659 -0.34108777]
 [-0.54775765  0.24716342  0.03627195  0.72325316  0.33832088]]
```

Note: Make sure that you have NumPy installed (**pip install numpy**)
if you haven't already done so.

Note: The import statement **from scipy import *** is generally
discouraged as it can lead to namespace pollution. It's better to import
specific functions or objects needed. Additionally, the **math** library is
not extensively used in this code, as most operations are performed
using **numpy** and **scipy** functions.

28.1 What is SVD?

The *Singular Value Decomposition (SVD)* of a matrix is a factorization of that matrix into three matrices. It also has some important applications in data science.

Mathematics behind SVD:

The SVD of mxn matrix A is given by the formula $\mathbf{A=U\sum V^T}$ where:

- U: *mxm* matrix of the orthonormal eigenvectors of AA^T.
- $\sum$: diagonal matrix with r elements equal to the root of the positive eigenvalues of AA^T or $A^T A$ (both matrics have the same positive eigenvalues anyway).
- V^T: transpose of a *nxn* matrix containing the orthonormal eigenvectors of A^TA.

For more about SVD, visit https://www.geeksforgeeks.org/singular-value-decomposition-svd/

29. Data visualization-random curve and fill

```python
import numpy as np
from matplotlib import pyplot as plt


ys = 200 + np.random.randn(100)
# Creating an array ys containing 100 random values drawn from a
normal distribution with mean 200.
x = [x for x in range(len(ys))]
# The line is using a list comprehension to create an array x containing the indices
of the array ys. So, after executing this line, x will be a list containing the indices
[0, 1, 2, ..., len(ys) - 1].


plt.plot(x, ys, '-')
plt.fill_between(x, ys, 195, where=(ys > 195),
facecolor='g', alpha=0.6)
# Fills the area between the line plot and the y-axis where ys is greater than 195
with a green color ('g') and an alpha (transparency) of 0.6.


plt.title("Sample Visualization")
plt.show()
```

Sample Visualization
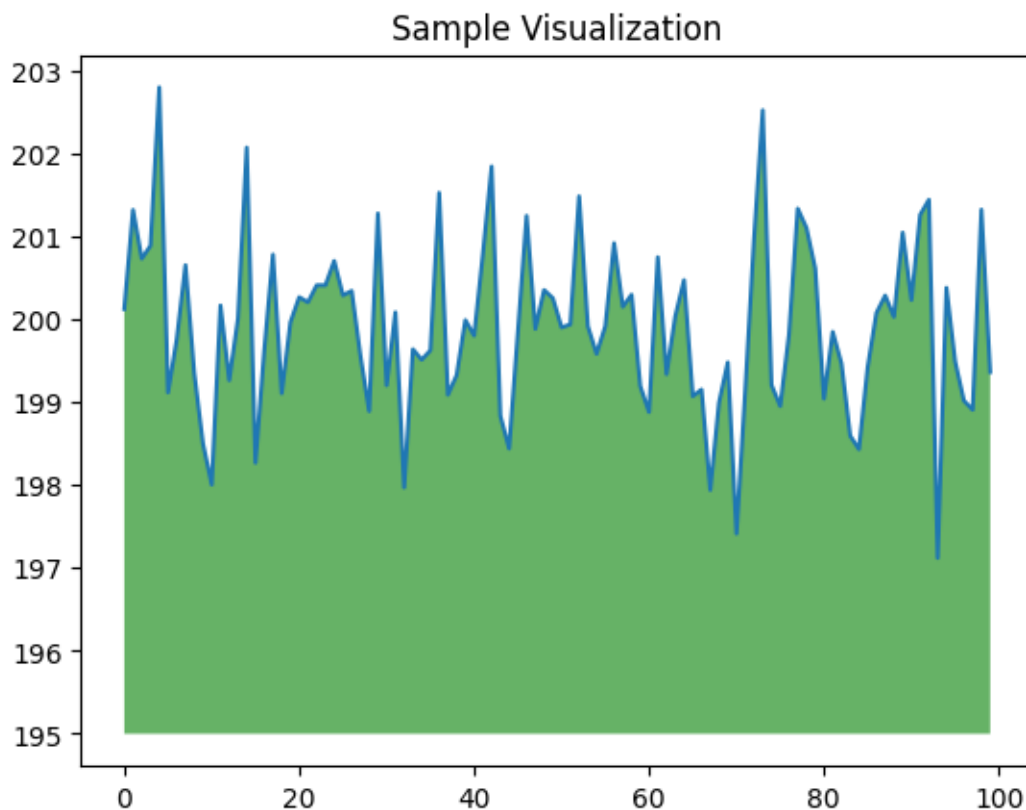
29.1 np.random.randn() & np.random.random()

what is the differences of randn and random?

Both `np.random.randn(100)` and `np.random.random(100)` generate random numbers, but they sample from different distributions:

1. **np.random.randn(100)**:

Generates an array of 100 random numbers sampled from a standard normal distribution (mean=0, standard deviation=1).

The numbers are drawn from a Gaussian (normal) distribution.

The range of values you can get from **np.random.randn** is theoretically unbounded. In practice, most of the values will be within a few standard deviations of the mean. Specifically, about 68% of the values will fall within one standard deviation of the mean, about 95% within two standard deviations, and about 99.7% within three standard deviations.

2. np.random.random(100):

Generates an array of 100 random numbers sampled from a <u>uniform distribution</u> over the interval [0.0, 1.0).

The numbers are drawn from a uniform distribution, meaning they are equally likely to fall anywhere in the specified range.

30. Ex: Ball falling under gravity.

A ball is initially at a height 10 m ($x=10$). Plot a graph to show the position of the ball both numerically and ~~graphically~~ analytically.

Differential equation:-

$$\frac{dv}{dt} = -g \quad [\because \text{down-ward acceleration, -ve sign}]$$

$$\Rightarrow \frac{x_{i+1} - 2x_i + x_{i-1}}{(\Delta t)^2} = -g \quad [\text{Euler's explicit method}]$$

$$\Rightarrow x_{i+1} - 2x_i + x_{i-1} = -g(\Delta t)^2 \cdots (i)$$

Considers $t_{max} = 5s$ and total 30 time-steps

$$\therefore \Delta t = \left(\frac{5}{30}\right) = \frac{1}{6} \quad \therefore -g(\Delta t)^2 = -0.2725$$

For initial conditions, $x_0 = 10 \cdots (ii)$

After 1st time step, $x_1 = x_0 + u_0 \cdot (\Delta t)$

$$= x_0 + 0 \cdot \Delta t \quad [\text{I.T} \Rightarrow \text{For this scheme,}$$
$$\Rightarrow -x_0 + x_1 = 0 \quad \text{velocity for 1st time}$$
$$\cdots (iii) \quad \text{is considered as initial velocity}]$$

For other cases, (i) is applicable.

$$\therefore \quad AX = B$$

$$\Rightarrow \begin{bmatrix} 1 & & & & & \\ -1 & 1 & & & & O \\ & 1 & -2 & 1 & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -2 & 1 \\ O & & & & & \ddots \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{30} \end{bmatrix} = \begin{bmatrix} 10 \\ 0 \\ -0.2725 \\ -0.2725 \\ \vdots \\ -0.2725 \end{bmatrix}$$

For analytical solution,
$$\left[x(t) = h + ut + \tfrac{1}{2}at^2 \right] \text{equation is used.}$$

(I.T$\Rightarrow$ For explicit euler, out of 3; 2 variable's value should be known. But, that is not applicable for 1st two conditions. So, Be's are used for 1st two rows)

```python
import numpy as np
import statistics
from scipy.linalg import solve
from matplotlib import pyplot as plt

t_max = 5;
n_max = 31;
#np = n_max;
h = 10;
g = -9.81;
v_ini=0;
```

```python
t_domain = np.linspace(0, t_max, n_max)
res = [t_domain[i + 1] - t_domain[i] for i in
range(len(t_domain)-1)]
```
#Time step sizes calculated as the difference between consecutive time values.
'range' is an one-dimensional array with total 30 delta_t values (1 less than
t_domain)

```python
dt = statistics.mean(res)
```
#Mean of the time step sizes. Though, here all-time steps are equal.

```python
#rhs side
b=np.zeros((n_max,1))
b[0,0] = h
b[1,0] = v_ini * dt
for i in range(2, n_max):
  b[i,0] = g * dt**2
```

```python
#lhs side
A=np.zeros((n_max,n_max))
A[0,0]=1;
A[1,0]=-1;
A[1,1]=1;
for i in range(2,n_max):
  A[i,i-2]=1;
  A[i,i-1]=-2;
  A[i,i]=1;
```

#Explanation of LHS and RHS are in handwritten part.


```python
X = np.linalg.inv(A).dot(b)
```
#This gives numerical solution. Obviously, this is a column vector. You may also
*use, **X = np.linalg.solve(A, b)**. This gives more stable solution and more precision.*
#Also note: A.dot(B) gives matrix multiplication and A*B gives element wise
multiplication.

```python
X_A=h+0.5*g*(np.square(t_domain))+v_ini*t_domain
```
 # This is analytical solution. It is calculated based on a mathematical expression
*rather than being the solution to a system of equations. So, X_A is a **1D array or***
***row vector**. Note that, 't_domain' is also an array.*

```python
print(X)
print(X_A)
```

```python
plt.rcParams["figure.figsize"] = [5,3.8]
```
#sets the default figure size for Matplotlib plots. 5 inches in width and 3.8 inches in height.
```python
plt.rcParams["figure.autolayout"] = True
```
#This line enables automatic layout adjustment for figures. When `figure.autolayout` is set to `True`, Matplotlib automatically adjusts the subplot parameters (e.g., margins, spacings) to fit the plot elements within the figure area. This can be useful to prevent issues where labels or titles get cut off.
```python
plt.rcParams['axes.linewidth'] = 1
```
 # sets the default linewidth for the axes (the border around the plot).
```python
plt.rc('xtick',labelsize=14)
plt.rc('ytick',labelsize=14)
```
 # sets the default font size for x and y-axis tick labels to 14 points.


```python
h1, =plt.plot(t_domain,X_A,'r-
',linewidth=2,label="analytical")
h2,=plt.plot(t_domain,X,'bo',linewidth=2,label="numer
ical")
```
*# creates another line plot with x-values from `t_domain` and y-values from `X`. It uses blue circles ('bo') as markers for the data points and sets the line width to 2. The label is set to "numerical" for use in the legend. **See 30.3** for knowing the significances of 'h1,' and 'h2,'..*

```python
plt.xlabel('t (s)',fontsize=20)
plt.ylabel('x (m)',fontsize=20)
```

```python
plt.xlim(0,t_max)
plt.ylim(-150,1.5*h)
```
#These represents highest and lowest values of x and t axes.
```python
plt.margins(x=0, y=0)
```
#zero margin. The plot will be stretched to fill the entire figure window. They provide a buffer between the plot and the edge of the figure. (I played with margins, but couldn't notice any difference)!!.

```python
first_legend = plt.legend(handles=[h1,h2], loc='lower
right')
```
*#creates a legend for the plot and assigns it to the variable first_legend. The **plt.legend**() function takes two arguments: **handles** and **loc**. The **handles** argument is a list of line objects that will be included in the legend. The **loc** argument specifies the location of the legend (lower right corner) on the plot.*

```
plt.show()
```
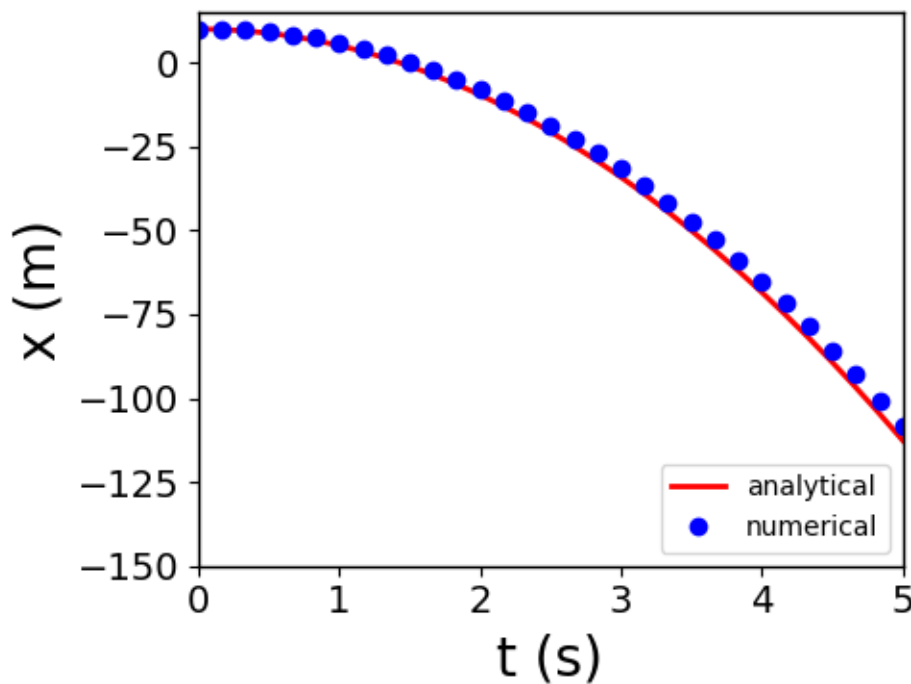

(30.1) x=2+ np.array([[1, 2, 3]])+np.array([[2, 5, 9]]) what is the result?

Adds the scalar 2 to each element of the two arrays element-wise. The result will be:

array([[5, 9, 14]])

(30.2) How to use column vectors and row vectors.

```python
import numpy as np

# define a column vector
x = np.array([[1], [2], [3]])

# define a row vector
y = np.array([[1, 2, 3]])

# calculate the dot product of the two vectors
result = x.dot(y)

# print the result
print(result)
```

```
⇨ [[1 2 3]
   [2 4 6]
   [3 6 9]]
```

30.3. The comma after h1 and h2 in the lines of code?

`h1, =plt.plot(t_domain,X_A,'r-',linewidth=2,label="analytical")` and `h2, =plt.plot(t_domain,X,'bo',linewidth=2,label="numerical")` is used?

This is to unpack the returned value of the `plt.plot()` function.

The `plt.plot()` function returns a tuple of line objects. In the given code, the first element of the tuple is assigned to the variable `h1` and the second element of the tuple is assigned to the variable `h2`. The comma after `h1` and `h2` tells Python to ignore the other elements of the tuple.

This is a common technique used to unpack the returned value of a function that returns multiple values. It is a convenient way to assign the returned values to separate variables without having to use explicit indexing.

Here is an example of how to unpack the returned value of a function that returns multiple values:

Python

```python
def my_function():
    return 1, 2, 3

# unpack the returned value of the function
x, y, z = my_function()

# print the variables
print(x)
print(y)
print(z)
```

Output:

```
1
2
3
```

This example shows that unpacking the returned value of a function can be a convenient way to assign the returned values to separate variables.

In the given code, unpacking the returned value of the `plt.plot()` function is used to assign the first two line objects to the variables `h1` and `h2`. This is done so that the line objects can be used to customize the plot, such as by changing the line color, line width, and line label.

Here is an example of how the variables `h1` and `h2` can be used to customize the plot:

Python

```python
# change the line color of the analytical solution to red
h1.set_color('red')

# change the line width of the numerical solution to 2
h2.set_linewidth(2)

# change the line label of the numerical solution to 'numerical'
h2.set_label('numerical')
```

By unpacking the returned value of the `plt.plot()` function, we can easily customize the plot without having to explicitly index the line objects.

31. Spring-mass damper system.

It solves 2nd order ODE using finite-difference. It represents simple mass-spring damper system.

Solution to simple harmonic motion,

$$x(t) = A\cos(\omega t + \phi)$$

A = Amplitude of the motion. (in code, $A = h = 0.1\,m$).

here, ϕ = initial phase angle = 0

We know, $\omega = \sqrt{\frac{K}{m}}$

$$\therefore x(t) = h\cos\left(\sqrt{\frac{K}{m}} \cdot t\right) \cdots (i)$$

ODE for damped harmonic oscillation,

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + Kx = 0$$

$$\Rightarrow \frac{m}{(\Delta t)^2}(x_{i+1} - 2x_i + x_{i-1}) + \frac{c}{2\Delta t}(x_{i+1} - x_{i-1}) + Kx_i = 0 \cdots (ii)$$

Where, x_i = displacement of the mass at i^{th} time-step.

1st and 2nd eq's will be formed using Bc's.

For other cases,

$A_1(i, i-2) = 1$, $A_1(i, i-1) = -2$, $A_1(i, i) = 1$

$A_2(i, i-2) = -1$ $A_2(i, i) = 1$

$A_3(i, i-1) = 1$

Without Bc's, the matrices looks like,

$$\Rightarrow [A]\begin{Bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{100} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{Bmatrix} \cdots (iii)$$

Boundary conditions,

$$1 \times x_0 = h \quad [\text{This is initial condition}] \cdots (iv)$$

for 1st time step, $x_1 = x_0 + v_{ini} * (\Delta t)$
$$= x_0 + 0 \times (\Delta t)$$
$$\Rightarrow -1 \cdot x_0 + 1 \times x_1 = 0 \cdots (v)$$

Applying (iv) and (v) in (iii),

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -1 & 0 & 1 & \cdots & 0 \\ & & \text{Others are} & \\ & & \text{same as (iii)} & \end{bmatrix}}_{A} \underbrace{\begin{Bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{100} \end{Bmatrix}}_{X} = \underbrace{\begin{Bmatrix} h \\ 0 \\ 0 \\ \vdots \\ 0 \end{Bmatrix}}_{b.}$$

```python
import numpy as np
import statistics
from scipy.linalg import solve
from matplotlib import pyplot as plt

t_max = 5
n_max = 101
m = 1
c = 0.1
k = 10
h = 0.1
v_ini = 0
#m: Mass of the system.
#c: Damping coefficient.
#k: Spring constant.
#h: Amplitude of the motion. (maximum displacement from equilibrium)
#v_ini: Initial velocity.
#t_max: Maximum time.
#n_max: Number of time steps.

t_domain = np.linspace(0, t_max, n_max)
res = [t_domain[i + 1] - t_domain[i] for i in
range(len(t_domain) - 1)]
dt = statistics.mean(res)
#t_domain: An array of time values using np.linspace from 0 to t_max with n_max
steps.
#res: array containing all time steps; and dt: Calculation of mean time step.

# LHS side
A1 = np.zeros((n_max, n_max))
A2 = np.zeros((n_max, n_max))
A3 = np.zeros((n_max, n_max))

for i in range(2, n_max):
    A1[i, i - 2] = 1
    A1[i, i - 1] = -2
    A1[i, i] = 1
    A2[i, i - 2] = -1
    A2[i, i] = 1
    A3[i, i - 1] = 1
```

```python
# whole coefficient matrix
A = (m / (dt * dt)) * A1 + (c / (2 * dt)) * A2 + k * A3
print(np.shape(A))
```
#This represents the size of A. i.e, 101×101.

```python
#Boundary conditions
A[0, 0] = 1
A[1, 0] = -1
A[1, 1] = 1


# RHS side
b = np.zeros((n_max, 1))


#Boundary conditions for RHS.
b[0, 0] = h
b[1, 0] = v_ini * dt
```
#for All other equations, RHS is 0 (As per the ODE of damped oscillation)


```python
X = np.linalg.inv(A).dot(b)
```
#Numerical solutions.

```python
# Calculate analytical solution
X_A = h * np.cos(np.sqrt(k / m) * t_domain)

plt.rcParams["figure.figsize"] = [5, 3.8]
plt.rcParams["figure.autolayout"] = True
plt.rcParams['axes.linewidth'] = 1
plt.rc('xtick', labelsize=14)
plt.rc('ytick', labelsize=14)

h1, = plt.plot(t_domain, X_A, 'r-', linewidth=2,
label="analytical")
h2, = plt.plot(t_domain, X, 'bo-', linewidth=2,
label="numerical")

plt.xlabel('t (s)', fontsize=20)
plt.ylabel('x (mm)', fontsize=20)

plt.xlim(0, t_max)
plt.ylim(-1.1 * h, 1.1 * h)
plt.margins(x=0, y=0)
```

```
# Combine legends
first_legend = plt.legend(handles=[h1, h2],
loc='lower right')
plt.show()
```
#For detailed explanation on plotting, see question no 30 .

(101, 101)
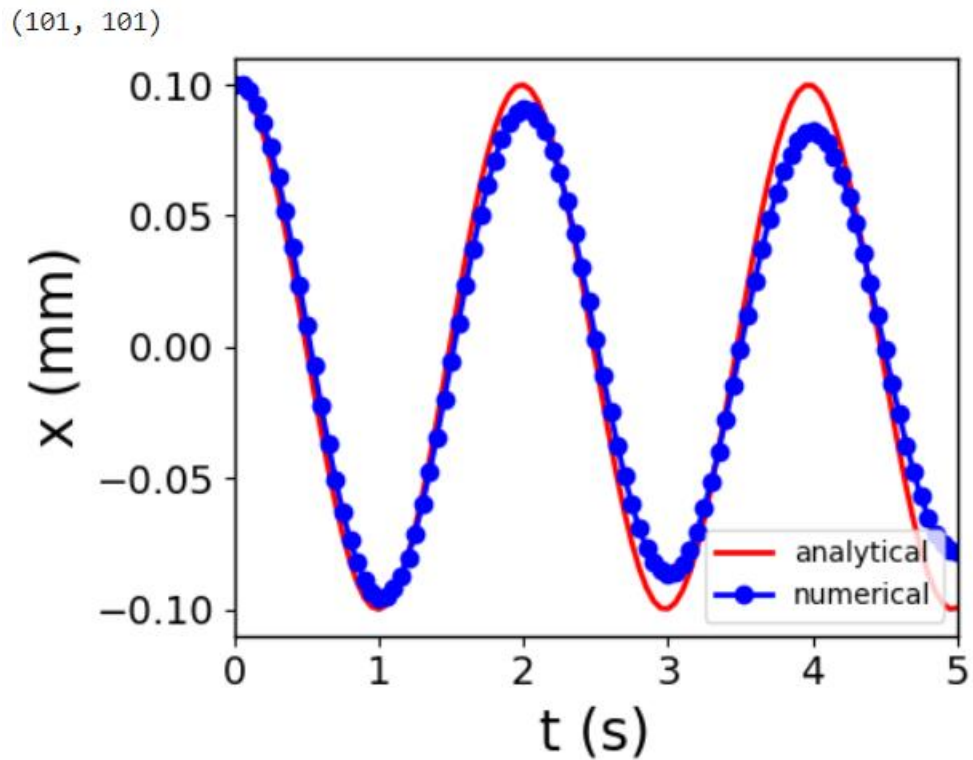
32. Ex: Spring-mass-damper with forcing

For mass-spring damper system,

according to Newton's 2nd law of motion,

Total force = -spring force + Damping force + External force.

$$\Rightarrow m \cdot a = -kx - c\frac{dx}{dt} + F$$

$$\Rightarrow m \cdot \frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = F$$

Discretizing, we get,

$$m\frac{x_{k+1} - 2x_k + x_{k-1}}{\Delta t^2} + c\frac{x_{k+1} - x_k}{\Delta t} + kx_k = F$$

If x^1 represents position and x^2 as velocity,

$$x^1_{k+1} = a_{11}x^1_k + a_{12}x^2_k + b_1 F$$

$$x^2_{k+1} = a_{21}x^1_k + a_{22}x^2_k + b_2 F$$

where, $a_{11} = 1$

$a_{12} = \Delta t$

$a_{21} = -\frac{\Delta t \cdot k}{m}$

$a_{22} = 1 - \frac{c\Delta t}{m}$

$b_1 = 0$

$b_2 = \frac{\Delta t}{m}$

Putting these values,

$$x^1_{k+1} = x^1_k + \Delta t\, x^2_k$$

$$x^2_{k+1} = -\frac{k\Delta t}{m}x^1_k + \left(1 - \frac{c\Delta t}{m}\right)x^2_k + \frac{\Delta t}{m} F.$$

(How they derived is not clean. Try it later).

```python
# Simulation of Mass-Spring-Damper System

import numpy as np
import matplotlib.pyplot as plt


# Model Parameters
c = 4 # Damping constant
k = 2 # Stiffness of the spring
m = 20 # Mass
F = 5 # Force


# Simulation Parameters
Ts = 0.1
Tstart = 0
Tstop = 60


N = int((Tstop-Tstart)/Ts) # Simulation length
x1 = np.zeros(N+2)
x2 = np.zeros(N+2)


x1[0] = 0 # Initial Position
x2[0] = 0 # Initial Speed


a11 = 1
a12 = Ts
a21 = -(Ts*k)/m
a22 = 1 - (Ts*c)/m
b1 = 0
b2 = Ts/m
#!!???

# Simulation
for k in range(N+1):
  x1[k+1] = a11 * x1[k] + a12 * x2[k] + b1 * F
  x2[k+1] = a21 * x1[k] + a22 * x2[k] + b2 * F

# Plot the Simulation Results
t = np.arange(Tstart,Tstop+2*Ts,Ts)
```
*# creates a NumPy **array containing the time steps** for the simulation. 2*Ts is added to ensure that the simulation runs for at least two time steps beyond the*

```
#plt.plot(t, x1, t, x2)
plt.plot(t,x1)
plt.plot(t,x2)
plt.title('Simulation of Mass-Spring-Damper System')
plt.xlabel('t [s]')
plt.ylabel('x(t)')
plt.grid()
plt.legend(["x1", "x2"])
plt.show()
```
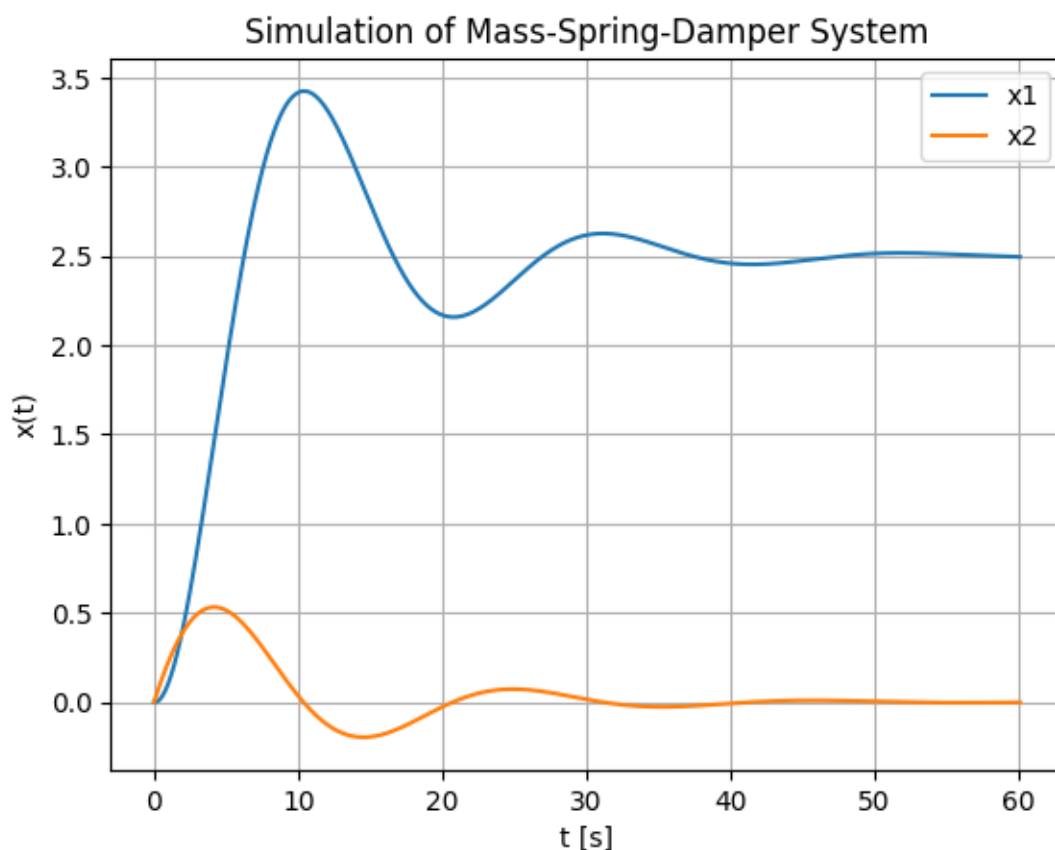
33. Spring-mass-damper with direct integration

```python
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

tstart=0.0
tend=60.0

incre=0.1
t=np.arange(tstart,tend+1,incre)
```
*# **np.arange** function is used to create an array **t** with time values from **tstart** to **tend** with a step size of **incre**.1 more time step was taken in simulation for better accuracy.*

```python
# initial condition
a=0
b=0.1
```
 # a is the initial position, and b is the initial velocity.
```python
x_init=[a,b]


def smdiff(x,t):
    c=4
    k=2
    m=20
    w=2  #frequency in radian per second.
    F=5*np.sin(w*t)
    dx1dt=x[1]
```
*# Assigns the velocity (**x[1]**) to the derivative of position ($dx1/dt$).*
```python
    dx2dt=(F-c*x[1]-k*x[0])/m
```
 # Calculates the acceleration or derivative of velocity ($dx2/dt$) based on the mass-spring-damper equation. It considers the external force (F), damping term ($c \cdot x[1]$), and spring term ($k \cdot x[0]$). The result is divided by the mass (m) to get acceleration.
```python
    dxdt =[dx1dt,dx2dt]
```
 # Combines the derivatives into a list, representing the rate of change of position and velocity.
```python
    return dxdt
```
#In Above, a function named smdiff is just defined which returns dxdt.

```python
x=odeint(smdiff,x_init,t)
```

*#odeint is used to numerically solve the ODEs defined by **smdiff**. It takes the function **smdiff**, initial conditions **x_init**, and time values **t** as inputs and returns an array **x** representing the system's state variables over time. X has two columns here.having position and velocity over time.*

```
print('x=',x)
```

```
x1=x[:,0]
x2=x[:,1]
```
#x1 and x2 takes values of position and velocity from column1 and column 2 of 'x' matrix respectively.

```
plt.plot(t,x1)
plt.plot(t,x2)
plt.xlabel('t')
plt.ylabel('..')
plt.grid()
plt.show()
```

```
=>    x= [[ 0.          0.1       ]
      [ 0.00998175  0.10044544]
      [ 0.02024684  0.10561742]
      ...
      [-0.05536826  0.06728618]
      [-0.04760101  0.08754405]
      [-0.03797525  0.10433233]]
```
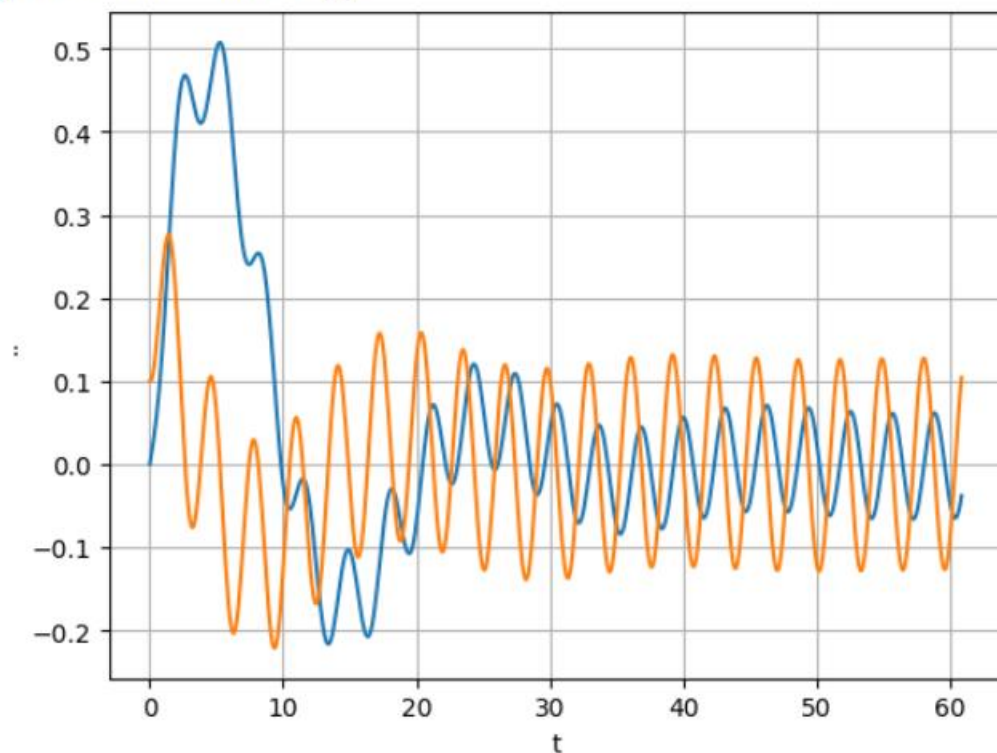
33.1 More about odeint function.

The `odeint` function in SciPy's `integrate` module is used for numerical integration of a system of ordinary differential equations (ODEs). The function signature for `odeint` is as follows:

Here's an explanation of the main arguments:

`scipy.integrate.odeint(func, y0, t, args=(), ...)`

`func`: The system of ordinary differential equations (ODEs) to be solved. It should take two arguments: the current state `y` and the current time `t`, and return the derivatives of the state variables.

`y0`: The initial conditions for the system. It represents the values of the state variables at the starting time `t[0]`. It can be a list or array.

`t`: The time points at which the solution should be computed. It can be an array of time values.

`args`: (Optional) Additional arguments to be passed to the ODE function (`func`). If your ODE function requires additional parameters beyond `y` and `t`, you can pass them through `args` as a tuple.

```python
from scipy.integrate import odeint
import numpy as np

def myODE(y, t, a, b):
    # Example ODE function
    dydt = a * y[0] - b * y[1]
    dzdt = b * y[0] + a * y[1]
    return [dydt, dzdt]

# Initial conditions
y0 = [1.0, 0.0]

# Time points
t = np.linspace(0,5,10)

# Additional parameters
a = 2.0
b = 1.0
```

#I think, if you provided the values of a and b within the function definition portion, you should not provide additional arguments in 'odeint'.

```
# Solve the ODEs
result = odeint(myODE, y0, t, args=(a, b))
print(result)
```

```
[[ 1.00000000e+00  0.00000000e+00]
 [ 2.58087982e+00  1.60214647e+00]
 [ 4.09406749e+00  8.26989504e+00]
 [-2.68328662e+00  2.79029013e+01]
 [-5.16297751e+01  6.77150181e+01]
 [-2.41739621e+02  9.20458646e+01]
 [-7.71371881e+02 -1.49742960e+02]
 [-1.75090783e+03 -1.62231927e+03]
 [-1.91968962e+03 -6.99222183e+03]
 [ 6.24807519e+03 -2.11217086e+04]]
```

34. Laplace equation solution

```
# Simple Numerical Laplace Equation Solution using
Finite Difference Method
import numpy as np
import matplotlib.pyplot as plt


# Set maximum iteration
maxIter = 500


# Set Dimension and delta
lenX = lenY = 20
```
#we set it rectangular. 20 points are taken in x and y directions.
```
delta = 1
```
 # Grid spacing.

```
# Boundary condition
Ttop = 100
Tbottom = 0
Tleft = 0
Tright = 30


# Initial guess of interior grid
```

```
Tguess = 30

# Set array size and set the interior value with Tguess
T = np.empty((lenX, lenY))
T.fill(Tguess)
```
 # Create an array T of size ($lenX$, $lenY$) and fill it with the initial guess value ($Tguess$).

```
# Set Boundary condition
T[(lenY-1):, :] = Ttop
T[:1, :] = Tbottom
T[:, (lenX-1):] = Tright
T[:, :1] = Tleft
```
#These things are explained in 34.2.

```
# Iteration (We assume that the iteration is
convergence in maxIter = 500)
print("Please wait for a moment")
for iteration in range(0, maxIter):
```
#The for loop iterates over the iteration numbers. The range() function creates a list of integers from 0 to maxIter-1.
```
    for i in range(1, lenX-1, delta):
        for j in range(1, lenY-1, delta):
```
#The inner for loop iterates over the grid points. The range() function creates a list of integers from 1 to lenX-2 and from 1 to lenY-2(because range function excludes the last value). This ensures that the loop does not iterate over the boundary points.
```
            T[i, j] = 0.25 * (T[i+1][j] + T[i-1][j] +
T[i][j+1] + T[i][j-1])
```
#This line of code updates the value of the scalar field at the grid point (i, j). The new value is equal to the average of the values at the four adjacent grid points.

```
print("Iteration finished")


# Set colour interpolation and colour map
colorinterpolation = 50
```
#The contour plot will have 50 different color levels. A higher value of $colorinterpolation$ will result in a smoother contour plot, but it will also take longer to render.
```
colourMap = plt.cm.jet
```

#The Matplotlib library provides a variety of built-in color maps. you can try: colourMap = plt.cm.coolwarm also.

```python
# Set meshgrid
X, Y = np.meshgrid(np.arange(0, lenX), np.arange(0, lenY))
```
 # to create a 2D grid of coordinates based on the 1D arrays generated by np.arange. See 34.3 for explanation.

```python
# Configure the contour
plt.title("Contour of Temperature")
plt.contourf(X, Y, T, colorinterpolation, cmap=colourMap)
```
 # This line generates a filled contour plot with colours representing different temperature levels. See 34.4 for explanation.

```python
# Set Colorbar
plt.colorbar()
```

```python
# Show the result in the plot window
plt.show()
```
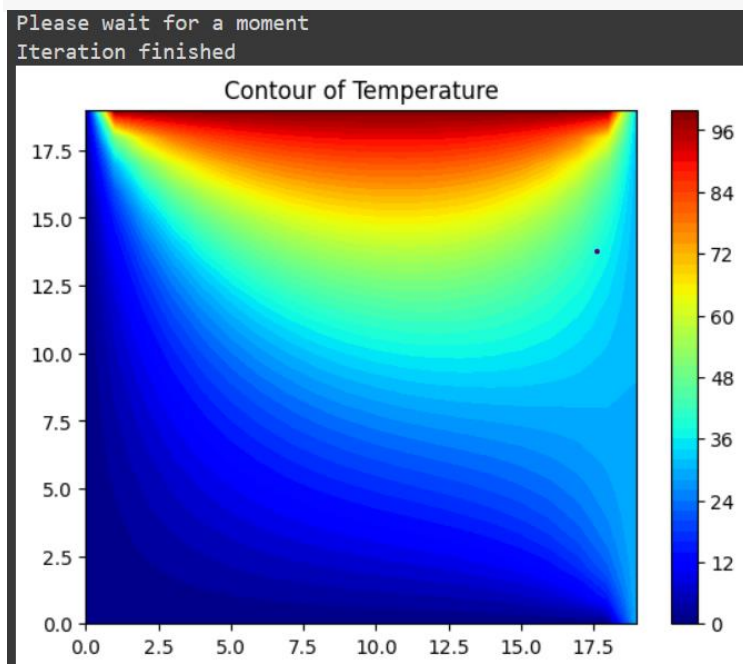
34.1 T = np.empty((lenX, lenY)). Can we place 'zeros' in place of 'empty'.

`np.zeros((lenX, lenY))`: This creates a 2D array of shape `(lenX, lenY)` filled with zeros.

`np.empty((lenX, lenY))`: This creates a 2D array of shape `(lenX, lenY)` without initializing the elements. The elements of the array may contain any values.

For this specific application, the use of `np.zeros` followed by `T.fill(Tguess)` is preferable for clarity and to ensure a well-defined initial state for the `T` array.

34.2 How boundary conditions are assigned in 2D grid.

```python
import numpy as np
lenX=5
lenY=5
T = np.zeros((lenX, lenY))
T[(lenY-1):, :] = 1    #Ttop
#"(lenY-1):" means: from row index =4 to all after that; means only 5th row for
this case.
T[:1, :] = 2           #Tbottom
# ":1" means:  from starting row to the row having row index=1 (i.e 2nd row) but
excludes that. That means , this BC is applicable for only 1st row and all columns.
T[:, (lenX-1):] =3     #Tright
T[:, :1] = 4           #Tleft
#The boundary conditions mentioned later overwrites the previous ones. So,
"Tleft" and "Tright" takes corner values.
print(T)

[[4. 2. 2. 2. 3.]
```

```
[4. 0. 0. 0. 3.]
[4. 0. 0. 0. 3.]
[4. 0. 0. 0. 3.]
[4. 1. 1. 1. 3.]]
```

Note that: Bottom boundary conditions takes the top row of the matrix. So, value entries to the matrix follows the below mentioned pattern. (Picture is taken from Computational hydraulics notes):
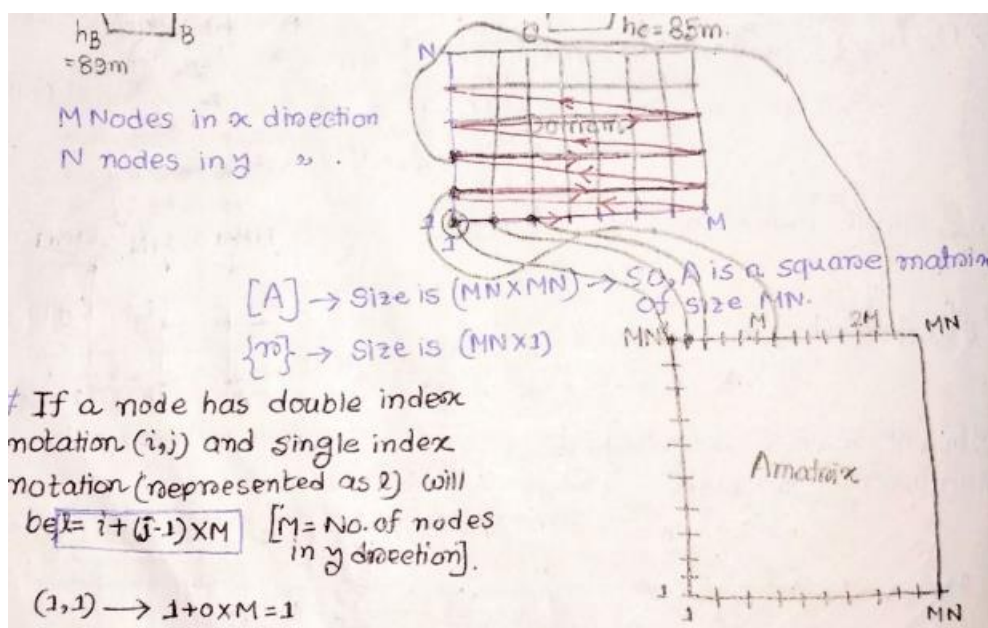
34.3. X, Y = np.meshgrid(np.arange(0, lenX), np.arange(0, lenY))

This line is using NumPy's **meshgrid** function to create a 2D grid of coordinates based on the 1D arrays generated by **np.arange**. Let's go step by step:

np.arange(0, lenX) and np.arange(0, lenY):

- These create 1D arrays of evenly spaced values from 0 to **lenX-1** and from 0 to **lenY-1**, respectively. These arrays represent the x and y coordinates along the respective axes of your grid.

np.meshgrid(...):

- This function takes multiple 1D arrays and produces 2D arrays representing a grid of coordinates.

- **np.meshgrid(np.arange(0, lenX), np.arange(0, lenY))** returns two 2D arrays, **X** and **Y**.

- **X** represents the x-coordinates for each point in the grid, and **Y** represents the y-coordinates.

Here, one example to elaborate this:

```python
import numpy as np
A=np.arange(0, 5)
B=(np.arange(0, 5), np.arange(0, 5))
X, Y = np.meshgrid(np.arange(0, 5), np.arange(0, 5))
print('A=',A)
print('B=',B)

print('X=',X)
print('Y=',Y)
```

⇨
```
A= [0 1 2 3 4]
B= (array([0, 1, 2, 3, 4]), array([0, 1, 2, 3, 4]))
X= [[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
Y= [[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]]
```

34.4. plt.contourf(X, Y, T, colorinterpolation, cmap=colourMap)

X and Y: These are the 2D arrays representing the coordinates of the grid. They were created using np.meshgrid and define the positions where the temperature values are sampled.

T: This is the 2D array representing the temperature values at each point on the grid.

colorinterpolation: We have assigned 50 distinct colors will be used to represent different temperature levels.

cmap=colourMap: This parameter specifies the colormap to be used for coloring the contours (previously defined as **plt.cm.jet**).

35. Class project: Temperature distribution over plate.


```
from scipy import *
import numpy as np
from math import *
```

```python
import pandas as pd
import matplotlib.pyplot as plt

mnode = 5
nnode = 5
Lx = 1
Ly = 1
delta_x = Lx / (mnode - 1)
delta_y = Ly / (nnode - 1)
x = np.linspace(0, 1, mnode)
y = np.linspace(0, -1, nnode)
#print(y)
alphax = 1 / (delta_x) ** 2
#print(alphax)
alphay = 1 / (delta_y) ** 2
mnnode = mnode * nnode
A = np.zeros((mnnode, mnnode))
#print(A)
r = np.zeros((mnnode, 1))
#print(r)
for j in range(0, nnode):
    for i in range(0, mnode):
        l = i + j * mnode
        if i == 0 and j in range(1, mnode-1):
            A[l, l] = 1
            r[l] = 10
        elif i in range(1, nnode-1) and j == 0:
            A[l, l] = 1
            r[l] = 100
        elif i in range(1, nnode-1) and j == (mnode -
1):
            A[l, l] = 1
            r[l] = 50
        elif i == (nnode - 1) and j in range(1, mnode -
1):
            A[l, l] = 1
            r[l] = 10
        elif i == 0 and j == 0:
            A[l, l] = 1
            r[l] = 100
```

```python
        elif i == 0 and j == mnode - 1:
            A[l, l] = 1
            r[l] = 50
        elif i == nnode - 1 and j == 0:
            A[l, l] = 1
            r[l] = 100
        elif i == nnode - 1 and j == mnode - 1:
            A[l, l] = 1
            r[l] = 50
        else:
            A[l, l - mnode] = alphay
            A[l, l - 1] = alphax
            A[l, l] = -2 * (alphax + alphay)
            A[l, l + 1] = alphax
            A[l, l + mnode] = alphay
            r[l] = sin(x[i] + y[j])


T = np.linalg.solve(A, r)
#print(T)
Temp=np.zeros((mnode,nnode))
for j in range(0,nnode):
  for i in range(0,mnode):
    Temp[i,j]=T[i+j*nnode]
print('Temp=',Temp)
```
#'T' is an one dimensional array where temperature values were arranged according to single index notation. Then in 'Temp' matrix, I arranged those values as per grid point locations.

```python
plt.contourf(x,y,Temp,cmap=plt.cm.jet)
```
 # The function `plt.contourf(x, y, Temp, cmap=plt.cm.jet)` in Python's Matplotlib library plots a filled contour plot of the scalar field `Temp` at the coordinates `x` and `y`.
*# The cmap keyword argument specifies the color map that will be used to represent the different levels of the scalar field in the contour plot. The Matplotlib library provides a variety of built-in color maps, including **plt.cm.jet, plt.cm.coolwarm**, and **plt.cm.rainbow.** The plt.cm.jet color map is a rainbow color map, with the colors ranging from blue to red. The plt.cm.gray color map is a grayscale color map, with the colors ranging from black to white.*
```python
plt.colorbar()
plt.xlabel('X')
plt.ylabel('Y')
```

```python
plt.title('Spatial Distribution of Temperature in
Square Plate')
plt.show()
```

```
Temp= [[100.          10.          10.          10.          50.         ]
 [100.          51.42857143 34.38155822 33.58219871 50.         ]
 [100.          61.33272749 42.5         39.91727251 50.         ]
 [100.          51.41780129 34.36844178 33.57142857 50.         ]
 [100.          10.          10.          10.          50.        ]]
```
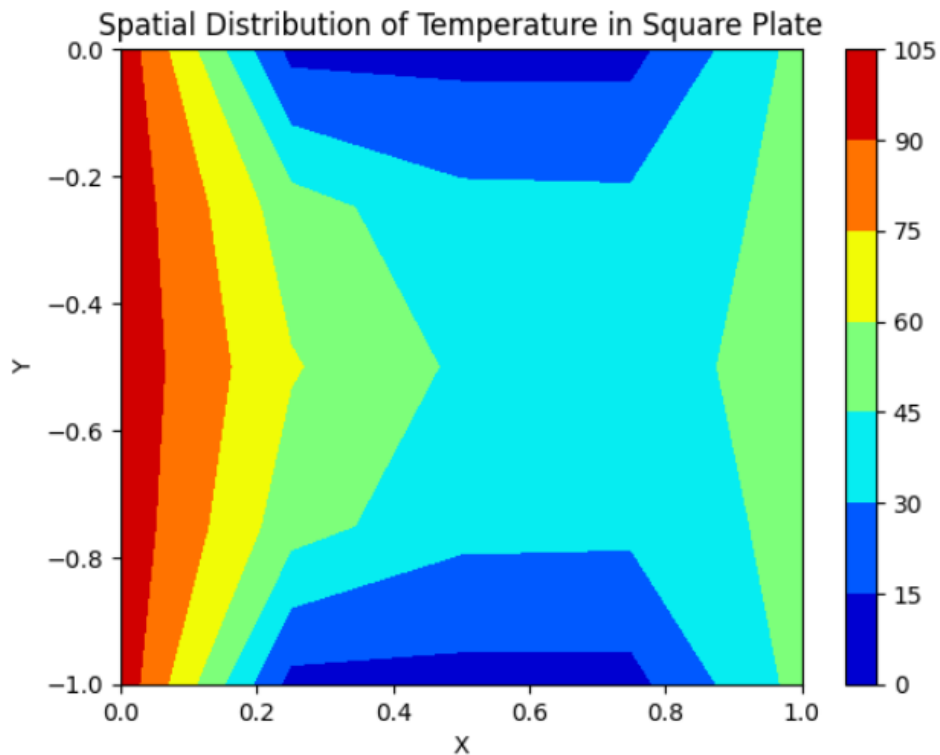

36. lists, loops, string manipulation, and basic Python syntax.

```python
from scipy import *
import numpy as np
from math import *
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import Image

friends=['joseph','gleen', 'sally']
print(friends)
```

output>> ['joseph', 'gleen', 'sally']

```python
x=2
x='hello'
x=4
print(x)
# Reassigns the variable x multiple times and prints its final value., which is 4.
```

Output>> 4

```python
print([1,24,76])
```

Output>> [1, 24, 76]

```python
print(['red','yellow','blue'])
```

Output>> ['red', 'yellow', 'blue']

```python
print([1,[5,6],7])
```

Output>> [1, [5, 6], 7]

```python
print(['red',24,98.6])
```

Output>> ['red', 24, 98.6]

```python
print([])
```

Output>> []

```python
for i in [5,4,3,2,1]:
  print (i)
print ('Blastoff!')
```

Output>>

5

4

3

2

1

Blastoff!

```python
friends=['joseph','gleen', 'sally']
for friend in friends:
  print ('Happy New Year', friend)
print ('done')
# Prints a New Year's greeting for each friend in the list.
```

Output>>

Happy New Year joseph

Happy New Year gleen

Happy New Year sally

done

```python
friends=['joseph' , 'gleen', 'sally']
print(friends[1])
# prints the second element of the list friends to the console. Index in python starts
from 0.
```

Output>> gleen

```python
fruit='Banana'
fruit[0]='b'
# it tries to reassign the first character of the string fruit to the character 'b'.
This is not possible, however, because strings are immutable in Python.
```

Output>>

**TypeError: 'str' object does not
support item assignment**

```python
fruit='Banana'
```

```
x=fruit.lower()
print (x)
```
#The code converts the string `fruit` to lowercase using the `lower()` method and then assigns the new string to the variable `x`. string 'fruit' is not changed; because it is immutable.

Output>> banana

```
lotto=[2,14,26,41,63]
print (lotto)
lotto[2]=28
# String are immutable in python, but lists are not. See 36.1 for details.
print (lotto)
```

Output>>

```
[2, 14, 26, 41, 63]

[2, 14, 28, 41, 63]
```

```
name=['joseph','gleen', 'sally']
name[1]='raja'
name[2]=90
print(name)
```

Output>> ['joseph', 'raja', 90]

```
greet='hello bob'
print (len(greet))
x=[1,2,44.3,'joe']
print (len(x))
```
#'len' function returns the number of characters in the string and the number of elements in list.

Output>>

9

4

```
print(range(4))
print (list(range(4)))
```

 # First line prints a `range` object, while the second line prints a list containing the values of the `range` object. A `range` object is a sequence of numbers. It is not a list, but it can be converted to a list using the `list()` function.

 # It is better to use a list than a range object because lists are more efficient for accessing elements than range objects. But, if you need to iterate over the sequence in a for loop, then you must use a range object.

Output>>

range(0, 4)

[0, 1, 2, 3]

#Use of 'range' in for loop:

```python
name=['joseph','gleen', 'sally']
print(len(name))
for i in range (len(name)):
  friend=name[i]
  print('happy new year:', friend)
```

Output >>

3

happy new year: joseph

happy new year: gleen

happy new year: sally

#Addition with and without Numpy.

#Without numpy

```python
a=[1,2,3]
b=[4,5,6]
c=a+b
d=[a[1],a[2]]
e=a+d
print('c=',c)
print('d=',d)
print('e=',e)
```

 # In python, the + operator for lists results in a new list that contains all the elements from the first list followed by all the elements from the second list. it doesn't perform element-wise addition.

```
Output >>
c= [1, 2, 3, 4, 5, 6]
d= [2, 3]
e= [1, 2, 3, 2, 3]
```

If you want to perform element-wise addition, you should consider using NumPy arrays.

#With NumPy:

```python
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = a + b
print('c=',c)
```

```
Output >> c= [5 7 9]
```

```python
t=[9,41,12,3,74,15,5]
```
#Remember, array indexing in python starts from 0, not 1.
```python
print(t[1:3])
```
#Prints from element_1 (including) to element_3 (excluding)[means up to element 2].

```python
print(t[:4])
```
 # Prints from element_0 to element_3

```python
print(t[3:])
```
 # Prints from element_3 to last element.

```python
print(t[:])
```
#Prints all element.

```python
print(t[::2])  #alternate terms
print(t[::3])  #every third terms
```

Output >>

[41, 12]

[9, 41, 12, 3]

[3, 74, 15, 5]

[9, 41, 12, 3, 74, 15, 5]

[9, 12, 74, 5]

[9, 3, 5]

```
x=[1,2,3]
print(type(x))
dir(x)
#dir(x) function to get a list of valid attributes and methods for the list object x.
This will provide you with a list of names containing the attributes and methods
supported by lists.
```

Output >>

```
<class 'list'>
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

```
stuff=list()
#This creates an empty list named stuff.


stuff.append('book')
stuff.append('99')
stuff.append('cookie')
print(stuff)
#This appends three elements ('book', '99', and 'cookie') to the list and prints the
contents of the list after the append operations.


stuff[2]=' '
print(stuff)
#This line modifies the element at index 2 (the third element, 'cookie') and replaces
it with a space (' ') and prints the updated list.
```

Output >>

['book', '99', 'cookie']

['book', '99', ' ']

```
some =[1,33,23,56]
print(9 in some)
```
#This checks if the value 9 is present in the list some. The output will be False because 9 is not in the list.

```
print(9 not in some)
```
#This checks if the value 9 is not present in the list some.

```
some.append(2)
```
#This appends the value 2 to the end of the list some.
```
print(some)
```

Output >>

False

True

[1, 33, 23, 56, 2]

```
friends=['joseph','gleen', 'sally']
friends.sort()
print (friends)
```
#sorts the list friends in alphabetical order and then prints it.

Output >> ['gleen', 'joseph', 'sally']

```
numbers=[1, 89, 3, 12, 234]
numbers.sort()
numbers.remove(12)
print(numbers)
```
#This sorts the list numbers in ascending order, then removes the element 12 from the list and prints the updated list.

```
print(len(numbers))    # length of the list 'numbers'
print(max(numbers))    #the maximum value in the list 'numbers'
```

```
print (min(numbers)) #the minimum value in the list 'numbers'
print (sum(numbers)) #sum of all elements in the list 'numbers'
print (sum(numbers)/len(numbers)) #average of all elements.
```

Output >>

[1, 3, 89, 234]

4

234

1

327

81.75

```
abc='with three words'
stuff=abc.split()
print(stuff)
```
#The split() method is then applied to this string, which splits the string into a list of words based on whitespace (spaces and tabs are the default delimiters). The resulting list is assigned to the variable stuff.
```
print(len(stuff))
print(stuff[0])
```

Output >>

['with', 'three', 'words']

3

With

```
line='A lot of        spaces'
etc=line.split()
```
 # spaces and tabs are the default delimiters for 'split'. So, no effect of lot of spaces between words.
```
print(etc)
```

Output >> ['A', 'lot', 'of', 'spaces']

```
New='with;three;words'
```

```python
# Semicolon is not a default delimiter for split method. So, everything in 'New'
string is considered as one element.
thing=New.split()
print(thing)
print(len(thing))
```

```
Output >>

['with;three;words']

1
```

36.1. String are immutable in python, but lists are not?

Yes, that's correct. In Python, strings are immutable, meaning you cannot change their content once they are created. If you want to modify a string, you need to create a new string. This is why the line `# fruit[0] = 'b'` in your script would raise an error.

On the other hand, lists in Python are mutable. You can modify their elements by assigning new values to specific indices. In your script, the line `lotto[2] = 28` is an example of modifying a list; it changes the value of the element at index 2 to 28.

37.Data handling with Pandas and Visualisation with Matplotlib

38. Pandas

Brief Overview of the Pandas Library

Pandas is a fast, powerful, and flexible open-source data analysis and data manipulation Python library. Built on top of the Python programming language, it provides data structures like Series and DataFrames for handling and analyzing structured data.

Data Structures:

- **Series**: 1D labeled array capable of holding any data type.

- **DataFrame**: 2D labeled data structure with columns that can be of different types.

- **Functionalities**: It offers various functionalities to perform tasks ranging from data cleaning, transformation, and visualization to more complex operations like aggregation and reshaping.

Relevance in Data Analysis

- **Data Wrangling**: Pandas makes it straightforward to clean and process messy and raw data into a more suitable and clean form.

- **Exploratory Data Analysis (EDA)**: With functions that help to inspect data, compute summary statistics, and visualize distributions, Pandas is a go-to tool for preliminary data investigation.

Why Pandas?

Importance in Data Manipulation and Analysis

- **Ease of Use**: The library's syntax is straightforward, which makes it easy for beginners to get started.

- **Flexibility**: Can handle a variety of data formats (like CSV, Excel, SQL databases, and even HDF5), and its DataFrame structure allows you to transform data quickly.

- **Performance**: Built on top of C libraries like NumPy, it's fast and efficient for data manipulation.

- **Community Support**: As one of the most popular data science libraries, it has a strong community and a plethora of tutorials, making it easier to find help and resources.

Ease of Use

- **Intuitive Syntax**: The Pandas API is designed to be intuitive and easy, making it accessible for new users and comprehensive for experienced ones.

- **Comprehensive Documentation**: An array of examples and community-contributed tutorials mean that most problems you encounter have already been solved and documented.

By integrating seamlessly with other data science libraries like Matplotlib for plotting and Scikit-learn for machine learning, Pandas enables end-to-end data analysis right within Python, making it an essential tool for anyone engaged in data science or data analysis.

import pandas and numpy

It is convention import these libraries in the following way.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Make sure you have Pandas installed in your Python environment. You can install it using pip if it's not already installed: `pip install pandas`

39. Basic Data Structures in Pandas: Series

A Pandas Series is essentially a one-dimensional array that can hold any data type. It comes with an index, which allows for both positional and label-based indexing.

Syntax: pd.Series(data)

Creation of a Pandas Series: You can create a Series from a list, dictionary, or NumPy array.

39.1 Panda series from a List

```python
import pandas as pd
data = [1, 2, 3, 4]
ds = pd.Series(data)
ds
```

```
0    1
1    2
2    3
3    4
dtype: int64
```

 # "dtype" stands for "data type," indicating the type of data the Series holds. "int64" is the specific data type, which represents 64-bit integers.

39.2 Panda series from a Dictionary

```python
import pandas as pd

data = {'a': 1, 'b': 2, 'c': 3}
print("The data dictionary is", data)
s = pd.Series(data)
print(s)
```

#This code creates a Pandas Series s from the dictionary data. In this Series, the keys from the dictionary become the index of the Series, and the values in the dictionary become the data in the Series. The default data type is "int64" because all values in this Series are integers.

```
⇨  The data dictionary is {'a': 1, 'b': 2, 'c': 3}
   a    1
   b    2
   c    3
```

39.3 Panda series from a NumPy Array

```python
import pandas as pd
import numpy as np
data = np.array([1, 2, 3])
print("The data as a numpy array is:", data)
s = pd.Series(data)
print(s)
# This code first creates a NumPy array data with elements [1, 2, 3] and then
converts it into a Pandas Series s.
```

⇨ **The data as a numpy array is: [1 2 3]**
   0 1
   1 2
   2 3
   dtype: int64

40. Difference between NumPy arrays and Pandas Series

If you are working with homogeneous numerical data and need performance (faster numerical computations), NumPy is a better choice. If you have labelled or heterogeneous data and need powerful data manipulation tools, Pandas Series and Data Frames are more suitable. In many cases, both NumPy and Pandas are used together to take advantage of their respective strengths.

Advantages of NumPy Arrays:

1. Homogeneous Data: NumPy arrays are great for numerical data with a single data type, which allows for efficient operations.

```python
1. import numpy as np
2.
3. # Example of a NumPy array
4. data = np.array([1, 2, 3])
5. data
```

⇨   **array([1, 2, 3])**

2.Performance: NumPy is optimized for numerical operations. In the following example, you'll likely see that NumPy is faster for numerical operations. Here's a simple performance comparison:

```python
import numpy as np
import pandas as pd
import timeit
# This library is for measuring execution time.

# Using NumPy array
numpy_data = np.arange(1, 10001)
# It creates a NumPy array numpy_data containing values from 1 to 10000 using
np.arange(1, 10001).

def numpy_sum():
    return np.sum(numpy_data)
# It defines a function numpy_sum() that calculates the sum of all elements in the
numpy_data using np.sum().

numpy_time = timeit.timeit(numpy_sum, number=10000)
# It uses the timeit module to measure the execution time of the numpy_sum function
by calling it 10,000 times. The result is stored in the numpy_time variable.


# Using Pandas Series
pandas_data = pd.Series(range(1, 10001))

def pandas_sum():
    return pandas_data.sum()

pandas_time = timeit.timeit(pandas_sum, number=10000)


print("sum with numpy=",numpy_sum())
print("sum with panda=",pandas_sum())
print("NumPy time:", numpy_time)
print("Pandas time:", pandas_time)

sum with numpy= 50005000
```

```
sum with panda= 50005000
NumPy time: 0.07156486299982134
Pandas time: 0.3480711040001552
```

Disadvantages of NumPy Arrays:

Homogeneity Requirement: NumPy arrays require homogeneous data. If you need to store different data types or have labeled data, NumPy is not well-suited.

Advantages of Pandas Series:

1. Heterogeneous Data: Pandas Series can store data with different data types in a single structure, which is useful for handling real-world data.

```python
import pandas as pd


# Example of a Pandas Series with heterogeneous data
data = pd.Series([1, 'two', 3.0])
# Here, elements are integer, string and float respectively. Since the data types are
# different, Pandas has inferred the data type of the Series as object.


print(data)


    ⇨ 0       1
      1       two
      2       3.0
      dtype: object
```

If you want to ensure that all elements have the same data type in your Series, you can explicitly specify the data type when creating the Series. For example:

```python
import pandas as pd
data = pd.Series([1, 'two', 3.0], dtype='string')
print(data)
```

```
⇨ 0       1
⇨ 1      two
⇨ 2      3.0
⇨ dtype: string
```

2.Labelled Indexing: Pandas Series provide labelled indices for easier and more intuitive data access.

```python
import pandas as pd

# Creating a Series with labeled index
data = pd.Series([10, 20, 30], index=['A', 'B', 'C'])
print(data['B'])   # Accessing data by label
```

⇨ **20**

3.Additional Data Manipulation Features: Pandas Series offer numerous data manipulation functions, like filtering, grouping, and handling missing data.

```python
import pandas as pd

data = pd.Series([10, 20, None, 30])
print(data.dropna())
#The output of data.dropna() will be a new Series with the missing value (None)
removed.
print(data)
```

```
⇨ 0      10.0
  1      20.0
  3      30.0
  dtype: float64
  0      10.0
  1      20.0
  2       NaN
  3      30.0
  dtype: float64
```

As you can see, the missing value has been removed from the Series, and the resulting Series has a data type of **float64** because Pandas

automatically converts the integers to floats to accommodate the missing value (**NaN**).

To replace missing values with 0, we use fillna().

```python
import pandas as pd

data = pd.Series([10, 20, None, 30])
data.fillna(0, inplace=True)
# Fills missing values with 0.
print(data)
```
```
⇨ 0     10.0
  1     20.0
  2      0.0
  3     30.0

  dtype: float64
```

Disadvantages of Pandas Series:

1. **Performance**: Due to their flexibility and heterogeneity, Pandas Series may not be as efficient as NumPy arrays for numerical operations, especially with large datasets.

2. **Overhead**: Pandas Series have additional overhead due to their data structure and indexing, which can lead to increased memory usage.

41. DataFrame

Definition: A Pandas DataFrame is a 2D table with labelled axes (rows and columns). Each column can have its own data type, and you can perform operations much like you would in a SQL table or an Excel spreadsheet.

Syntax:

```
pd.DataFrame(data, columns=columns, index=index)
```

Creation of a Pandas DataFrame DataFrames can be created from lists, dictionaries, Series, and even other DataFrames.

41.1. DataFrame from a List of Lists

```python
import pandas as pd

# Create a list of lists
data = [[1, 'a'], [2, 'b'], [3, 'c']]

# Create a DataFrame using the list of lists and column
names
df = pd.DataFrame(data, columns=['Number', 'Letter'])
# The first column is named "Number" and contains the values [1, 2, 3]. The
second column is named "Letter" and contains the values ['a', 'b', 'c'].


# Print the DataFrame
print(df)
# provides a nicely formatted output in a non-interactive Python script to display
the DataFrame's content in the console or save it to a file.

df
#is more concise and is often used for quick inspection of the DataFrame's content
in an interactive environment like Jupyter Notebook, but it may not be as nicely
formatted.
```

```
   Number Letter
0       1      a
1       2      b
2       3      c
```

	Number	Letter
0	1	a
1	2	b
2	3	c

41.2 DataFrame from a Dictionary

```python
import pandas as pd

data = {'Number': [1, 2, 3], 'Letter': ['a', 'b', 'c']}
```

```
df = pd.DataFrame(data)
print(df)
```
#Your code efficiently creates a DataFrame by providing a dictionary where the keys become column names, and the values become the data in those columns. This is a common and convenient way to create Pandas DataFrames for various data analysis tasks.

```
⇨    Number Letter
0        1      a
1        2      b
2        3      c
```

41.2 DataFrame from a Series

```
s1 = pd.Series([1, 2, 3])
s2 = pd.Series(['a', 'b', 'c'])
df = pd.DataFrame({'Number': s1, 'Letter': s2})
```
#The keys ('Number' and 'Letter') are specified as the column names, and their corresponding values are the Series s1 and s2.
```
df
```

	Number	Letter	
0	1	a	
1	2	b	
2	3	c	

By understanding Series and DataFrames, you set the foundation for almost everything you'll do with Pandas, from data manipulation to advanced analysis.

42. Data Import and Export in Pandas

Molecular Property Data

Now, let us import data on molecular properties. For this tutorial, We take the solubility data from Guillaume Lambard . This file contains index column, so we add index_column=0.

```
import pandas as pd
url="https://raw.githubusercontent.com/GLambard/Molecul
es_Dataset_Collection/master/latest/ESOL_delaney-
processed.csv"

df = pd.read_csv(url, index_col=0)
```
 # This line uses Pandas' `read_csv` function to read the CSV file from the specified URL. The `index_col=0` argument indicates that the first column in the CSV file should be used as the index (row labels) of the DataFrame.

```
df
```

Output =>

	Compound ID	ESOL predicted log solubility in mols per litre	Minimum Degree	Molecular Weight	Number of H-Bond Donors	Number of Rings	Number of Rotatable Bonds	Polar Surface Area	measured log solubility in mols per litre	smiles
0	Amigdalin	-0.974	1	457.432	7	3	7	202.32	-0.770	N#CC(OC1OC(COC2OC(CO)C(O)C(O)C2O)C(O)C(O)C1O)C...
1	Fenfuram	-2.885	1	201.225	1	2	2	42.24	-3.300	CC1:O:C:C:C:1C(=O)NC1:C:C:C:C:C:1
2	citral	-2.579	1	152.237	0	0	4	17.07	-2.060	CC(C)=CCCC(C)=CC=O
3	Picene	-6.618	2	278.354	0	5	0	0.00	-7.870	C1:C:C:C2:C(:C:1):C:C:C1:C:2:C:C:C2:C3:C:C:C:C...
4	Thiophene	-2.232	2	84.143	0	1	0	0.00	-1.330	C1:C:C:S:C:1
...	...	...	...	...	...	...	...	...	...	...
1123	halothane	-2.608	1	197.381	0	0	0	0.00	-1.710	FC(F)(F)C(Cl)Br
1124	Oxamyl	-0.908	1	219.266	1	0	1	71.00	0.106	CNC(=O)ON=C(SC)C(=O)N(C)C
1125	Thiometon	-3.323	1	246.359	0	0	7	18.46	-3.091	CCSCCSP(=S)(OC)OC
1126	2-Methylbutane	-2.245	1	72.151	0	0	1	0.00	-3.180	CCC(C)C
1127	Stirofos	-4.320	1	365.964	0	1	5	44.76	-4.522	COP(=O)(OC)OC(=CCl)C1:C:C(Cl):C(Cl):C:C:1Cl

1128 rows × 10 columns

Minimum Degree:

Description: The minimum degree (number of edges connected to a node) in the molecular graph representation of the compound.
Type: Numerical/Integer

Smiles:

Description: Simplified Molecular Input Line Entry System (SMILES) notation, representing the structural formula of the compound.
Type: Categorical/String

ESOL model

The ESOL (Estimated SOLubility) model is an empirical model designed to predict the aqueous solubility of organic compounds. It was developed by John S. Delaney and published in a 2004 paper titled "ESOL: Estimating Aqueous Solubility Directly from Molecular Structure". The model uses a set of molecular descriptors, such as molecular weight, number of atoms, and polar surface area, to estimate the log solubility (logS) of a compound in water.

The ESOL model is relatively simple and interpretable compared to more complex machine learning models. It is often used as a baseline or comparative standard in solubility prediction tasks. The model is particularly useful for screening large compound libraries in drug discovery and other chemical engineering applications, where quick and reasonable solubility estimates are needed.

43. Data Inspection

Now, we will see the effects of different commands in the above csv file.

```
df.head()
# displays the first 5 rows of the DataFrame, along with the column names.
```

	Compound ID	ESOL predicted log solubility in mols per litre	Minimum Degree	Molecular Weight	Number of H-Bond Donors	Number of Rings	Number of Rotatable Bonds	Polar Surface Area	measured log solubility in mols per litre	smiles
0	Amigdalin	-0.974	1	457.432	7	3	7	202.32	-0.77	N#CC(OC1OC(COC2OC(CO)C(O)C(O)C2O)C(O)C(O)C1O)C...
1	Fenfuram	-2.885	1	201.225	1	2	2	42.24	-3.30	CC1:O:C:C:C:1C(=O)NC1:C:C:C:C:C:1
2	citral	-2.579	1	152.237	0	0	4	17.07	-2.06	CC(C)=CCCC(C)=CC=O
3	Picene	-6.618	2	278.354	0	5	0	0.00	-7.87	C1:C:C:C2:C(:C:1):C:C:C1:C:2:C:C:C2:C3:C:C:C:C...
4	Thiophene	-2.232	2	84.143	0	1	0	0.00	-1.33	C1:C:C:S:C:1

```
df.tail()
# displays the last 5 rows of the DataFrame, along with the column names.
```

	Compound ID	ESOL predicted log solubility in mols per litre	Minimum Degree	Molecular Weight	Number of H-Bond Donors	Number of Rings	Number of Rotatable Bonds	Polar Surface Area	measured log solubility in mols per litre	smiles
1123	halothane	-2.608	1	197.381	0	0	0	0.00	-1.710	FC(F)(F)C(Cl)Br
1124	Oxamyl	-0.908	1	219.266	1	0	1	71.00	0.106	CNC(=O)ON=C(SC)C(=O)N(C)C
1125	Thiometon	-3.323	1	246.359	0	0	7	18.46	-3.091	CCSCCSP(=S)(OC)OC
1126	2-Methylbutane	-2.245	1	72.151	0	0	1	0.00	-3.180	CCC(C)C
1127	Stirofos	-4.320	1	365.964	0	1	5	44.76	-4.522	COP(=O)(OC)OC(=CCl)C1:C:C(Cl):C(Cl):C:C:1Cl

df.info()

#The df.info() method in Pandas provides a summary of the DataFrame's basic information, including the number of non-null entries, data types, and memory usage.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1128 entries, 0 to 1127
Data columns (total 10 columns):
 #   Column                                        Non-Null Count  Dtype
---  ------                                        --------------  -----
 0   Compound ID                                   1128 non-null   object
 1   ESOL predicted log solubility in mols per litre  1128 non-null   float64
 2   Minimum Degree                                1128 non-null   int64
 3   Molecular Weight                              1128 non-null   float64
 4   Number of H-Bond Donors                       1128 non-null   int64
 5   Number of Rings                               1128 non-null   int64
 6   Number of Rotatable Bonds                     1128 non-null   int64
 7   Polar Surface Area                            1128 non-null   float64
 8   measured log solubility in mols per litre     1128 non-null   float64
 9   smiles                                        1128 non-null   object
dtypes: float64(4), int64(4), object(2)
memory usage: 96.9+ KB
```

df.shape

#The df.shape attribute in Pandas returns a tuple representing the dimensions of the DataFrame.

⇨   **(1128, 10)**

df.describe()

	ESOL predicted log solubility in mols per litre	Minimum Degree	Molecular Weight	Number of H-Bond Donors	Number of Rings	Number of Rotatable Bonds	Polar Surface Area	measured log solubility in mols per litre
count	1128.000000	1128.000000	1128.000000	1128.000000	1128.000000	1128.000000	1128.000000	1128.000000
mean	-2.988192	1.058511	203.937074	0.701241	1.390957	2.177305	34.872881	-3.050102
std	1.683220	0.238560	102.738077	1.089727	1.318286	2.640974	35.383593	2.096441
min	-9.702000	0.000000	16.043000	0.000000	0.000000	0.000000	0.000000	-11.600000
25%	-3.948250	1.000000	121.183000	0.000000	0.000000	0.000000	0.000000	-4.317500
50%	-2.870000	1.000000	182.179000	0.000000	1.000000	1.000000	26.300000	-2.860000
75%	-1.843750	1.000000	270.372000	1.000000	2.000000	3.000000	55.440000	-1.600000
max	1.091000	2.000000	780.949000	11.000000	8.000000	23.000000	268.680000	1.580000

In this output:

- **count**: Indicates the number of non-null values in each numerical column.

- **mean**: Represents the mean (average) of each numerical column.

- **std**: Shows the standard deviation, a measure of the spread or dispersion of the data.

- **min**: Displays the minimum value in each column.

- **25%**, **50%**, and **75%**: These are quartiles, where the 25% quartile corresponds to the first quartile (Q1), the 50% quartile is the median (Q2), and the 75% quartile is the third quartile (Q3).

- **max**: Shows the maximum value in each column.

```
df.index
```
#df.index is an attribute in Pandas that allows you to access the index (row labels) of a DataFrame.

⇨   `Int64Index([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,`
`... 1118, 1119, 1120, 1121, 1122, 1123, 1124,`
`1125, 1126, 1127], dtype='int64', length=1128)`

If there was a name of index column, then output should look like:

⇨   `Int64Index([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,`
`... 1118, 1119, 1120, 1121, 1122, 1123, 1124,`
`1125, 1126, 1127], dtype='int64',name=<'name of`
`the index column'>, length=1128)`

44. Row and Column Selection using iloc[] and loc[].

The loc[] method allows you to select rows and columns from a DataFrame based on their labels.

Syntax: `DataFrame.loc[<row_labels>, <column_labels>]`

Features

Label-based: Uses the explicit index and column names for selection.

Sliceable: Allows slicing of rows and columns using their labels.

Boolean Masking: Can be used with boolean conditions to filter data.

Examples.

44.1. With loc[]

Selecting a Single Row by Index Label:

```
df.loc[1]
# When you use df.loc[1], you are using the .loc indexer in Pandas to access a
specific row in the DataFrame by its label. Here, label=1 means 2nd row.
```

```
Compound ID                                                  Fenfuram
ESOL predicted log solubility in mols per litre               -2.885
Minimum Degree                                                     1
Molecular Weight                                               201.225
Number of H-Bond Donors                                            1
Number of Rings                                                    2
Number of Rotatable Bonds                                          2
Polar Surface Area                                             42.24
measured log solubility in mols per litre                      -3.3
smiles                               CC1:O:C:C:C:1C(=O)NC1:C:C:C:C:C:1
Name: 1, dtype: object
```

Selecting Multiple Rows by Index Labels

```
rows = df.loc[1:3]
rows
```
 # you are selecting rows (index numbers) 1 to 3 (inclusive) from your DataFrame and creating a new DataFrame called **rows**.

	Compound ID	ESOL predicted log solubility in mols per litre	Minimum Degree	Molecular Weight	Number of H-Bond Donors	Number of Rings	Number of Rotatable Bonds	Polar Surface Area	measured log solubility in mols per litre	smiles
1	Fenfuram	-2.885	1	201.225	1	2	2	42.24	-3.30	CC1:O:C:C:C:1C(=O)NC1:C:C:C:C:C:1
2	citral	-2.579	1	152.237	0	0	4	17.07	-2.06	CC(C)=CCCC(C)=CC=O
3	Picene	-6.618	2	278.354	0	5	0	0.00	-7.87	C1:C:C:C2:C(:C:1):C:C:C1:C:2:C:C:C2:C3:C:C:C:C...

Selecting Specific Columns with Rows

```
A=df.loc[1:3, ['Compound ID', 'smiles']]
A
```
#Here, 'Compound ID', 'smiles' are column indices.

	Compound ID	smiles
1	Fenfuram	CC1:O:C:C:C:1C(=O)NC1:C:C:C:C:C:1
2	citral	CC(C)=CCCC(C)=CC=O
3	Picene	C1:C:C:C2:C(:C:1):C:C:C1:C:2:C:C:C:C2:C3:C:C:C:C...

Selecting columns

```
B=df[['Compound ID', 'Molecular Weight']]
B
```

	Compound ID	Molecular Weight
0	Amigdalin	457.432
1	Fenfuram	201.225
2	citral	152.237
3	Picene	278.354
4	Thiophene	84.143
...	...	...
1123	halothane	197.381
1124	Oxamyl	219.266
1125	Thiometon	246.359
1126	2-Methylbutane	72.151
1127	Stirofos	365.964

1128 rows × 2 columns

45. Difference between loc and iloc?

In Pandas, both `loc` and `iloc` are used to select specific rows and columns from a DataFrame, but they differ in how they are used to index and select data:

`loc` (Label-Based Selection):

`loc` is primarily label-based. It allows you to select data by specifying the row and column labels.

You can use actual labels or boolean conditions to filter data.

The ending index in a range is included in the selection. For example, `df.loc[1:3]` selects rows with labels 1, 2, and 3.

You can also use string labels to select specific rows and columns, like `df.loc['row_label', 'column_label']`.

`iloc` (Integer-Based Selection):

`iloc` is primarily <mark>integer-based</mark>. It allows you to select data by specifying row and column positions using integers.

It is similar to Python list indexing where the first element is at position 0.

The <mark>ending index in a range is not included</mark> in the selection. For example, `df.iloc[1:3]` selects rows at positions 1 and 2, but not 3.

You use integer positions to select specific rows and columns, like `df.iloc[0, 1]`.
Here's a side-by-side comparison:

	`loc`	`iloc`
Selection by labels	Yes	No
Selection by integers	No	Yes
Inclusive range	Yes	No (end index not included)
Boolean conditions	Yes	No
Example usage	`df.loc['row_label']`	`df.iloc[row_position]`

46. Selecting with iloc[]

Syntax:

`DataFrame.iloc[<row_positions>, <column_positions>]`

The `iloc[]` method allows you to select rows and columns from a DataFrame based on their integer positions. Features

Position-based: Uses the implicit integer index for selection.

Boolean Masking: Can be used with boolean arrays to filter data.

Examples:

Selecting a Single Row by Position:

```
df.iloc[4]
```
#Extract information of 5th row (index=4).

```
Compound ID                                              Thiophene
ESOL predicted log solubility in mols per litre             -2.232
Minimum Degree                                                   2
Molecular Weight                                            84.143
Number of H-Bond Donors                                          0
Number of Rings                                                  1
Number of Rotatable Bonds                                        0
Polar Surface Area                                             0.0
measured log solubility in mols per litre                    -1.33
smiles                                                  C1:C:C:S:C:1
Name: 4, dtype: object
```

Selecting Multiple Rows by Position

```
df.iloc[0:3]
```
 # Row no 4 (i.e., index=3) is excluded while we use iloc.

	Compound ID	ESOL predicted log solubility in mols per litre	Minimum Degree	Molecular Weight	Number of H-Bond Donors	Number of Rings	Number of Rotatable Bonds	Polar Surface Area	measured log solubility in mols per litre	smiles
0	Amigdalin	-0.974	1	457.432	7	3	7	202.32	-0.77	N#CC(OC1OC(COC2OC(CO)C(O)C(O)C2O)C(O)C(O)C1O)C...
1	Fenfuram	-2.885	1	201.225	1	2	2	42.24	-3.30	CC1:O:C:C:C:1C(=O)NC1:C:C:C:C:C:1
2	citral	-2.579	1	152.237	0	0	4	17.07	-2.06	CC(C)=CCCC(C)=CC=O

Selecting Specific Columns with Rows

```
df.iloc[0:3, 0:2]
```

	Compound ID	ESOL predicted log solubility in mols per litre
0	Amigdalin	-0.974
1	Fenfuram	-2.885
2	citral	-2.579

Selecting Specific Cells

```
df.iloc[0, 1]
```
⇨ **-0.974**

46.1. To get a list of the column names.

```
df.columns
```
⇨ **Index(['Compound ID', 'ESOL predicted log solubility in
mols per litre', 'Minimum Degree', 'Molecular Weight',
'Number of H-Bond Donors', 'Number of Rings', 'Number of
Rotatable Bonds', 'Polar Surface Area', 'measured log
solubility in mols per litre', 'smiles'], dtype='object')**

47. Conditional Selection

47.1 Show how to filter rows based on column values.

Question: How many molecules have more than five rings?

```
condition = df['Number of Rings']>5
```
#This line creates a boolean Series `condition` *where each element is* `True` *if the corresponding element in the 'Number of Rings' column is greater than 5, and* `False` *otherwise.*
```
df[condition]
```
#it will give you a new DataFrame containing only the rows where the 'Number of Rings' is greater than 5.

	Compound ID	ESOL predicted log solubility in mols per litre	Minimum Degree	Molecular Weight	Number of H-Bond Donors	Number of Rings	Number of Rotatable Bonds	Polar Surface Area	measured log solubility in mols per litre	smiles
188	Diosgenin	-5.681	1	414.630	1	6	0	38.69	-7.320	CC1COC2CC3(C)C(CC4C5C=C6CC(O)CCC6(C)CC5CCC43C)...
219	Etoposide (148-167,25mg/ml)	-3.292	1	588.562	3	7	5	160.83	-3.571	COC1:C:C(C2C3:C:C4:C(:C:C:3C(OC3OC5COC(C)OC5C(...
429	Kepone	-5.112	1	490.639	0	6	0	17.07	-5.259	O=C1C2(Cl)C3(Cl)C4(Cl)C(Cl)(Cl)C5(Cl)C3(Cl)C1(...
555	Digoxin (L1=41,8mg/mL, L2=68,2mg/mL, Z=40,1mg/mL)	-5.312	1	780.949	6	8	7	203.06	-4.081	CC1OC(OC2C(O)CC(OC3C(O)CC(OC4CCC5(C)C(CCC6C5CC...
640	Digitoxin	-6.114	1	764.950	5	8	7	182.83	-5.293	CC1OC(OC2C(O)CC(OC3C(O)CC(OC4CCC5(C)C(CCC6C5CC...
676	Benzo[ghi]perylene	-6.446	2	276.338	0	6	0	0.00	-9.018	C1:C:C2:C:C:C3:C:C:C4:C:C:C5:C:C:C6:C(:C:1):...
718	Coronene	-6.885	2	300.360	0	7	0	0.00	-9.332	C1:C:C2:C:C:C3:C:C:C4:C:C:C5:C:C:C6:C:C:C:1:C1...
922	norbormide	-4.238	1	511.581	2	7	5	92.18	-3.931	O=C1NC(=O)C2C3C(C(O)(C4:C:C:C:C:4)C4:C:C:C:C...
988	Mirex	-6.155	1	545.546	0	6	0	0.00	-6.800	ClC1(Cl)C2(Cl)C3(Cl)C4(Cl)C(Cl)(Cl)C5(Cl)C3(Cl...

47.2 Conditional Selection with Multiple Conditions

```
condition = (df['Number of Rings'] >3) & (df['Number of
H-Bond Donors']>3)
df[condition]
```

Output =>

	Compound ID	ESOL predicted log solubility in mols per litre	Minimum Degree	Molecular Weight	Number of H-Bond Donors	Number of Rings	Number of Rotatable Bonds	Polar Surface Area	measured log solubility in mols per litre	smiles
62	Triamcinolone	-2.734	1	394.439	4	4	2	115.06	-3.680	CC12C=CC(=O)C=C1CCC1C3CC(O)C(O)(C...
555	Digoxin (L1=41,8mg/mL, L2=68,2mg/mL, Z=40,1mg/mL)	-5.312	1	780.949	6	8	7	203.06	-4.081	CC1OC(OC2C(O)CC(OC3C(O)CC(OC4CCC5(C...
640	Digitoxin	-6.114	1	764.950	5	8	7	182.83	-5.293	CC1OC(OC2C(O)CC(OC3C(O)CC(OC4CCC5(C...
790	hematein	-1.795	1	300.266	4	4	0	107.22	-2.700	O=C1C=C2CC3(O)COC4:C(:C:C:C(O):C:...

47.3. Which is the molecule with highest molecular weight?

```
condition = df['Molecular Weight'] == df['Molecular
Weight'].max()
```

*#**df['Molecular Weight']** refers to the 'Molecular Weight' column in your DataFrame **df**. It creates a Pandas Series containing all the 'Molecular Weight' values.*

***df['Molecular Weight'].max()** calculates the maximum value within the 'Molecular Weight' column. This is the largest value in that column.*

***df['Molecular Weight'] == df['Molecular Weight'].max()** is a comparison. It checks each value in the 'Molecular Weight' column to see if it is equal to the maximum 'Molecular Weight' value. This comparison results in a*

boolean Series, where each element is `True` if the corresponding value in the 'Molecular Weight' column is equal to the maximum value and `False` otherwise.

```
df[condition]
```

	Compound ID	ESOL predicted log solubility in mols per litre	Minimum Degree	Molecular Weight	Number of H-Bond Donors	Number of Rings	Number of Rotatable Bonds	Polar Surface Area	measured log solubility in mols per litre	smiles
555	Digoxin (L1=41,8mg/mL, L2=68,2mg/mL, Z=40,1mg/mL)	-5.312	1	780.949	6	8	7	203.06	-4.081	CC1OC(OC2C(O)CC(OC3C(O)CC(OC4CCC5(C)C(CCC6C5CC...

48. Resetting index

Let us set the index line of the DataFrame to the column named `'Compound ID'`.

```
df_new = df.set_index('Compound ID')
```
#This line creates a new DataFrame called `df_new` with the index set to the values in the 'Compound ID' column. The original DataFrame `df` remains unchanged.

```
df.set_index('Compound ID', inplace=True)
```
*#This line **modifies the original DataFrame `df` in place**, setting its index to the values in the 'Compound ID' column. This means that `df` will no longer have its default integer-based index, and the changes will be applied directly to `df` without creating a new DataFrame.*

49. Matplotlib

49.1 Plot a graph for molecular weight vs log solubility.

```
import matplotlib.pyplot as plt
df.plot('Molecular Weight','measured log solubility in mols per litre')
```
 # This code creates a line plot (line chart). It connects the data points with lines, assuming a continuous relationship between the points. It's suitable for showing trends or patterns over a continuous range of values.

```
df.plot('Molecular Weight', 'measured log solubility in mols per litre', kind='scatter')
```

#This code creates a scatter plot. A scatter plot displays individual data points as separate markers without connecting them with lines. It's useful for visualizing the distribution of data points, identifying outliers, and showing the relationship between two variables in a more discrete manner.

```
<Axes: xlabel='Molecular Weight', ylabel='measured log solubility in mols per litre'>
```
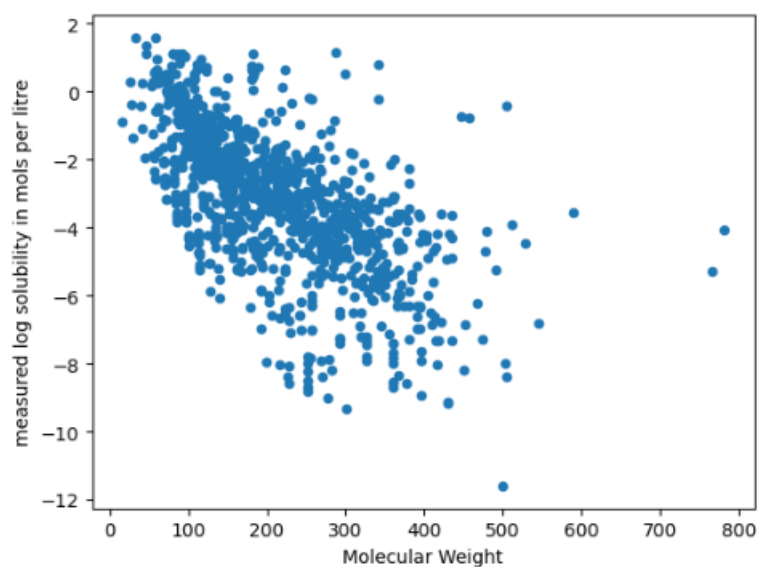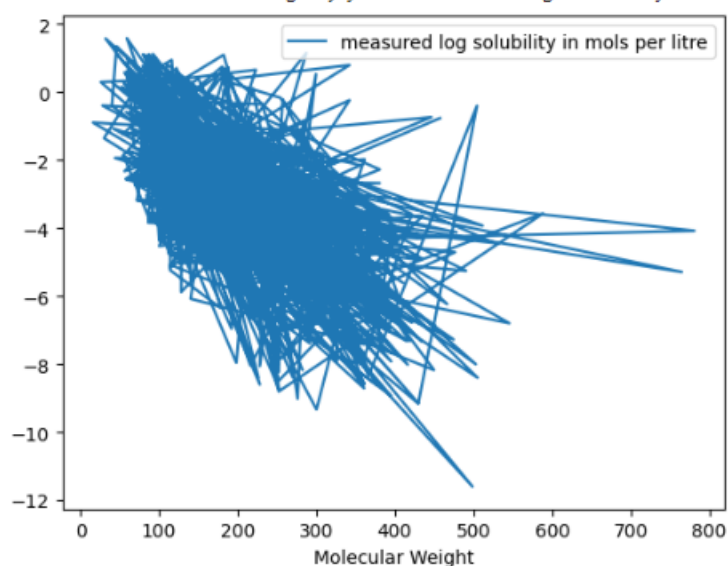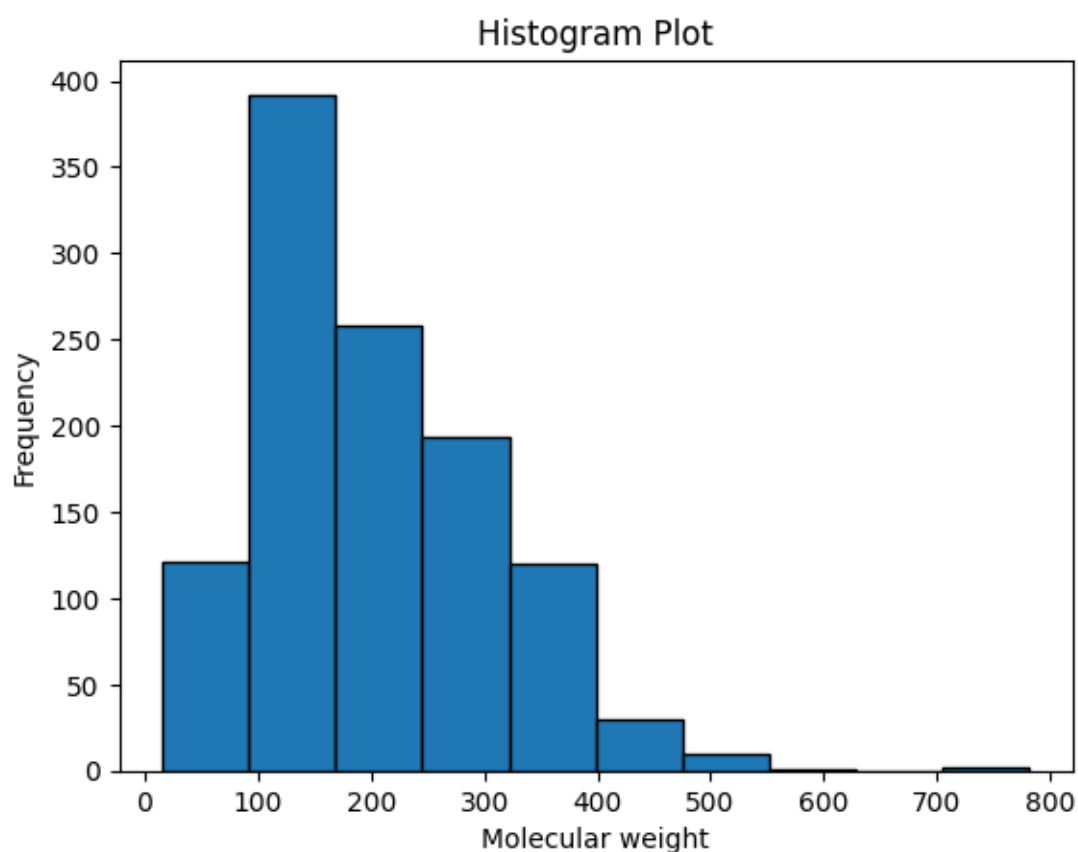
49.2 Make a histogram.

```python
import matplotlib.pyplot as plt

# Sample data
data = df['Molecular Weight']

# Plotting the histogram
plt.hist(data, bins=10, edgecolor='black')  # Adjust
the number of bins as needed

# Adding labels and title
plt.xlabel('Molecular weight')
plt.ylabel('Frequency')
plt.title('Histogram Plot')

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

To make your histogram plot more visually appealing, you can add various customizations and visual enhancements. Here's an improved version of your code with added styling:

```python
import matplotlib.pyplot as plt
import seaborn as sns  # Import seaborn for improved styling (you may need to install it)

# Sample data
data = df['Molecular Weight']

# Set up a Matplotlib figure with a specific size
plt.figure(figsize=(8, 6))
# 8-inch width and 6-inch height.

# Plotting the histogram with customizations
sns.histplot(data, bins=20, kde=True, color='skyblue', edgecolor='black')
# In Seaborn's sns.histplot() function, the kde parameter is used to
# control the display of a Kernel Density Estimation (KDE) plot overlay on top of the
# histogram. Setting kde=True means that you want to superimpose a KDE plot
# on your histogram. A KDE plot is a smoothed representation of the data's
# distribution and provides an estimate of the probability density function of the
# data. The peaks of the KDE curve correspond to regions of higher data density.

# Adding labels and title
plt.xlabel('Molecular Weight', fontsize=14)
plt.ylabel('Frequency', fontsize=14)
plt.title('Distribution of Molecular Weight', fontsize=16)

# Customize the axis labels and ticks
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)

# Add a legend
plt.legend(["Data"], fontsize=12)

# Add a grid
```
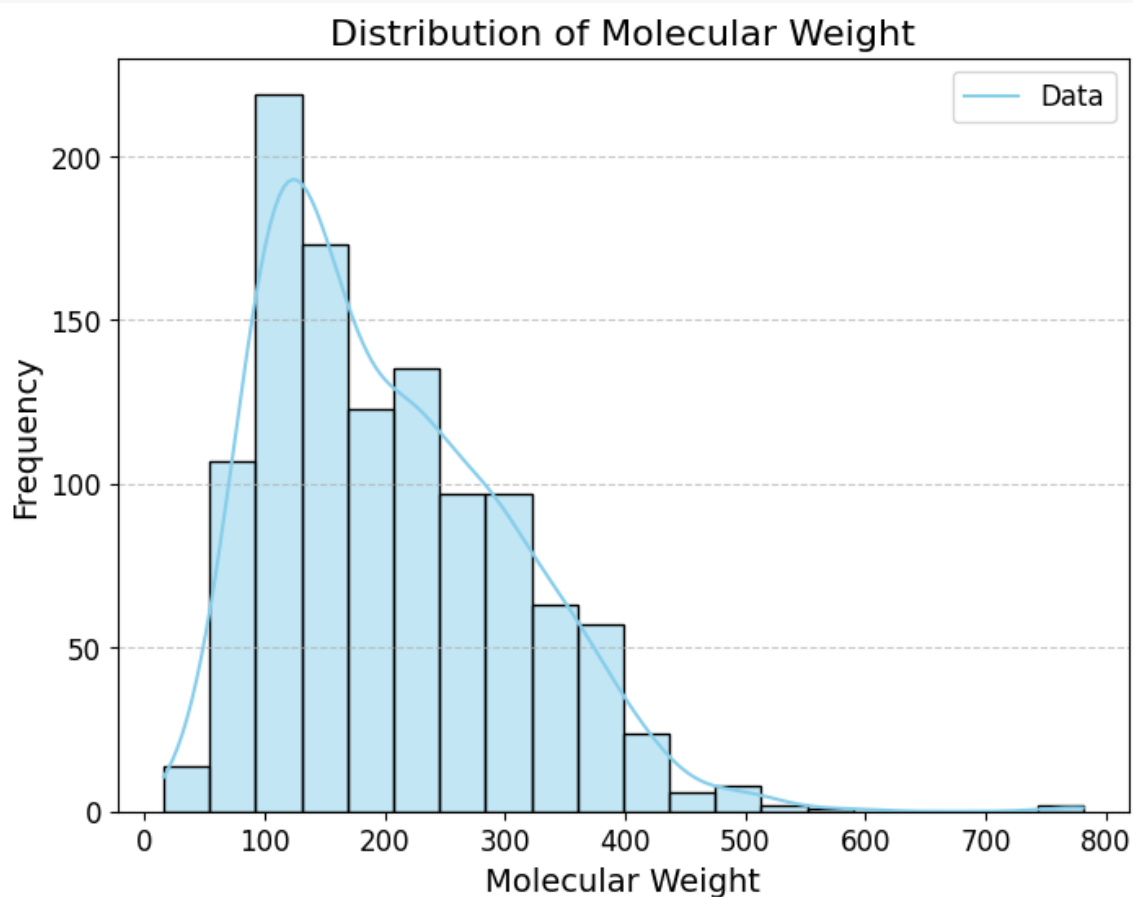
```
plt.grid(axis='y', linestyle='--', alpha=0.7)

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


49.3 Linear regression plot

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

# Sample data
x = df['Molecular Weight']
y = df['measured log solubility in mols per litre']

# Perform linear regression (fit a line to the data)
coefficients = np.polyfit(x, y, 1)
```
 # Use NumPy's `polyfit` function to perform linear regression (denoted by: 1). It fits a straight line (linear regression) to the data points and returns the coefficients of

the line. In this case, `coefficients` will contain the slope (a) and the intercept (b) of the linear regression line.

```
a, b = coefficients
```

```
# Create a linear regression line
regression_line = a * x + b
```
 # Compute the values of the linear regression line using the coefficients obtained in the previous step. This line represents the best-fit linear relationship between the two variables.
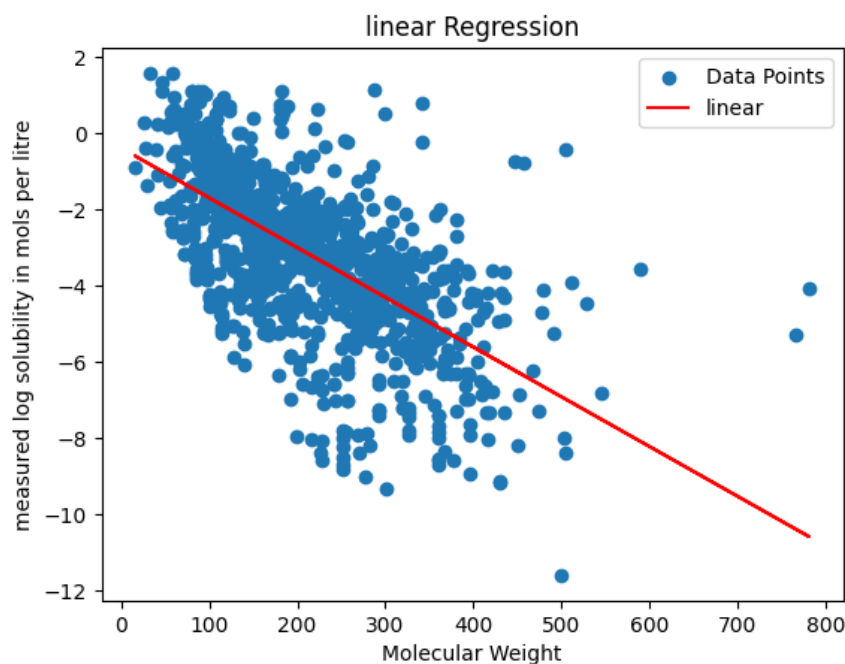
```
# Plot the data points
plt.scatter(x, y, label='Data Points')
```

```
# Plot the regression line
plt.plot(x, regression_line, color='red',
label='linear')
```

```
# Add labels and title
plt.xlabel('Molecular Weight')
plt.ylabel('measured log solubility in mols per litre')
plt.title('linear Regression')
```

```
# Add a legend
plt.legend()
```

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

linear Regression

50. Machine Learning Assignment (By me)

Assignment 08/11/2023

In this work, **Glambard Molecules Dataset Collection** was used and various existing features were combined from the dataset to predict log solubility. Then, **Random Forest Regressor model** was trained to predict the 'measured log solubility in mols per litre' using the new feature and the normalized molecular weight.

The solubility of a compound can be influenced by various molecular properties. Let's discuss how the properties can affect the solubility of a compound:

1. Molecular Weight:

• Generally, higher molecular weight might reduce solubility as larger molecules tend to have stronger intermolecular forces that may hinder interactions with the solvent.

2. Number of Hydrogen Bond Donors:

- Molecules with more hydrogen bond donor groups (e.g., -OH, -NH) might increase solubility as they can form hydrogen bonds with the solvent, aiding in dissolution.

3. Number of Rings:

- A higher number of rings might decrease solubility as it can increase the molecule's overall size and reduce its flexibility, affecting interactions with the solvent.

4. Number of Rotatable Bonds:

- Compounds with many rotatable bonds could have reduced solubility since increased flexibility might hinder effective interactions with the solvent.

5. Polar Surface Area:

- Higher polar surface area might generally increase solubility as polar groups can interact favorably with the solvent through hydrogen bonding or other polar interactions.

Finally, we get **Mean Absolute Error= 0.7312687222924569**; which is a quite satisfactory value.

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
#Imports necessary libraries for data handling (Pandas), visualization
(Matplotlib), numerical operations (NumPy), machine learning model
(RandomForestRegressor), splitting data for training and testing (train_test_split),
and model evaluation metric (mean_absolute_error).

url =
"https://raw.githubusercontent.com/GLambard/Molecules_D
ataset_Collection/master/latest/ESOL_delaney-
processed.csv"
df = pd.read_csv(url, index_col=0)
```

#Loads a dataset from the provided URL into a Pandas DataFrame. The index column is specified as 0th column.

```python
# Molecular weight normalization
df['Normalized MW'] = df['Molecular Weight'] /
np.mean(df['Molecular Weight'])
```
 # Normalizes the molecular weight values by dividing each value by the mean of all molecular weights in the dataset. This helps to ensure that the molecular weight feature is on a consistent scale with the other features.

```python
# Feature engineering
df['Other Properties'] =- df['Molecular Weight'] -
df['Number of Rotatable Bonds']+df['ESOL predicted log
solubility in mols per litre']-df['Number of
Rings']+df['Number of H-Bond Donors']+df['Polar Surface
Area']

# Data splitting
X = df[['Normalized MW', 'Other Properties']]
y = df['measured log solubility in mols per litre']

X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.2, random_state=42)
```
*# This code splits the data into two sets: a training set (**X_train** and **y_train**) and a test set (**X_test** and **y_test**). The **train_test_split**() function is used to split the data randomly, ensuring that both sets represent the distribution of the original data. 80% of the data is used for training (**X_train**, **y_train**), and 20% for testing (**X_test**, **y_test**).*

```python
# Model training with Random Forest
model = RandomForestRegressor(random_state=42)
model.fit(X_train, y_train)
```
*# Initializes and trains a Random Forest Regressor model using the training data (**X_train**, **y_train**).*

```python
# Model evaluation
y_pred = model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```

```
std_dev = np.std(y_test - y_pred)

print("Mean Absolute Error:", mae)
print("Root Mean Square Error (RMSE):", rmse)
print("Standard Deviation:", std_dev)
```
 # It uses the predict( ) method to make predictions for the test data and then calculates the mean absolute error (MAE) between the predicted values and the actual values. And finally, prints the mae.

Output >>
Mean Absolute Error: 0.7312687222924569
Root Mean Square Error (RMSE): 1.0719598758851299
Standard Deviation: 1.0703410961313409

51. Latex Classwork (1/11/2023)

51.1. By sir

\documentclass[12pt]{article}
\usepackage{graphicx}
\title{ETSC \LaTeX}
\author{Sayak Karmakar}

\begin{document}
\maketitle
Hello! This is my first \LaTeX Document.

\section{Introduction}
This is a new paragraph.\\[12pt]
This is new line.

\section{Maths}
F=m a

$F = m a $

$y=x_1^2+x^2+3$

$$y=x_1^2+x^2+3$$

$\sqrt[3]{x^2+y^2}$

$$\sqrt{1+\sqrt{x^2}}$$

\begin{equation}
x = a + b
\end{equation}

\subsection{Greek Symbols}
Greek Symbols: α, Π

Other: ∇, δ, Δ

Fractions: $\frac{1}{2}$

$\int_a^b{x^2}dx$

$a_1, a_2, a_3,\ldots, a_{100}$

Trigonometric: $\sin x + \cos^2 x$

logarithmic: $\log_2 x, \log X, \ln x$

$$(a+b)^2$$

$$\left(\frac{a}{b}+b\right)^2$$

$$\sum{x^2}$$
$$\epsilon$$
Here, I am creating table \ref{tab:my_label}
\begin{table}[h]
  \centering
  \begin{tabular}{|c|c|c|c|c|}\hline
    No & a & b & c &d \\\hline
    1 & 2 & 3 & 4& 4\\\hline
    2 & 4 & 5 & 6 & 2\\\hline
    3 & 4 & 4 & 4& 3\\\hline
  \end{tabular}
  \caption{Caption}
  \label{tab:my_label}
\end{table}

%These things for inserting figures.
\begin{figure}[h]
  \centering
  \includegraphics[width=0.2\linewidth]{Image/flower.jpeg}
  \caption{Flower}
  \label{fig:enter-label}
\end{figure}

%creating list
\begin{itemize}
  \item item1
  \item item2
\end{itemize}

Here is the reference \cite{DOI: 10.1021/acs.energyfuels.3c02680}
\bibliography{achs_enfuemAxA}

\end{document}

Fractions: $\frac{1}{2}$
$\int_a^b x^2 dx$
$a_1, a_2, a_3, \ldots, a_{100}$
Trigonometric: $\sin x + \cos^2 x$
logarithmic: $\log_2 x, \log X, \ln x$

$$(a+b)^2$$

$$\left(\frac{a}{b}+b\right)^2$$

$$\sum_c x^2$$

Here, I am creating table 1

No	a	b	c	d
1	2	3	4	4
2	4	5	6	2
3	4	4	4	3

Table 1: Caption

Figure 1: Flower

- item1
- item2

Here is the reference [?]

2

51.2 By me

\documentclass[14pt]{article}
\usepackage{graphicx, float} % Required for inserting images
\usepackage{amsfonts,amsmath, amssymb}

\title{Computational Hydraulics}
\author{Sayak Karmakar}
\date{November 2023}

\begin{document}
\begin{titlepage}

```
\begin{center}
   \vspace*{5cm}
   \Huge{\textbf{Computational Hydraulics}}\\
   \Huge{\textbf{Governing Equations}}\\
   \vfill
   \Large{\textbf{-by-}}\\
   \Huge {\textbf{Sayak Karmakar}}\\
   \Huge {\textbf{Roll no: 22CE92R05}}\\
   \today\\
  \end{center}
\end{titlepage}
```

Computational Hydraulics
Governing Equations

-by-
Sayak Karmakar
Roll no: 22CE92R05
November 8, 2023

.

```
\tableofcontents
```

\thispagestyle{empty}
\clearpage

Contents

\setcounter{page}{1}

\maketitle

\section{Learning objective}
To identify Governing Equations for Hydraulic Systems.

\section{Introduction}
\subsection{ODE}
Differential Equation with ONE independent variable.
\subsection{PDE}
Differential Equation with two or more independent variables.

\section{Basic Principles}
\begin{itemize}
    \item Conservation of Mass
    \item Conservation of Momentum
    \item Conservation of Energy

\end{itemize}

\subsection{Mass Conservation Equation}
\begin{equation}
\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}=0
\end{equation}

\subsection{Momentum Conservation Equation}
\begin{equation}
\frac{\partial u}{\partial t}+\frac{\partial uu}{\partial x}+\frac{\partial uv}{\partial y}+\frac{\partial uw}{\partial z}=-\frac{1}{\rho}\frac{\partial P}{\partial x}+ g_x+\frac{\mu}{\rho}\left(\frac{\partial^2 u}{\partial^2x}+\frac{\partial^2 u}{\partial^2y}+\frac{\partial^2 u}{\partial^2z}\right)
\end{equation}

Computational Hydraulics

Sayak Karmakar

November 2023

1 Learning objective

To identify Governing Equations for Hydraulic Systems.

2 Introduction

2.1 ODE

Differential Equation with ONE independent variable.

2.2 PDE

Differential Equation with two or more independent variables.

3 Basic Principles

- Conservation of Mass
- Conservation of Momentum
- Conservation of Energy

3.1 Mass Conservation Equation

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{1}$$

3.2 Momentum Conservation Equation

$$\frac{\partial u}{\partial t} + \frac{\partial uu}{\partial x} + \frac{\partial uv}{\partial y} + \frac{\partial uw}{\partial z} = -\frac{1}{\rho}\frac{\partial P}{\partial x} + g_x + \frac{\mu}{\rho}\left(\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} + \frac{\partial^2 u}{\partial^2 z}\right) \tag{2}$$

1

\section{Solution of Governing Equations}
Solution of Governing Equations in computational Hydraulics are of two types i.e. Implicit and Explicit. Difference between these two are described below in table \ref{tab:my_label}
\begin{table}[h]
    \centering
    \def\arraystretch{1.5}
    \begin{tabular}{|p{4cm}|p{4cm}|p{4cm}|}\hline
        Aspect & Implicit Solution & Explicit Solution\\\hline
        Numerical Stability & Stable for a wider range of time steps. & Can be conditionally stable based on the CFL (Courant-Friedrichs-Lewy) condition, limiting the time step for stability.\\\hline
        Time Dependency & Computationally more expensive due to solving nonlinear systems of equations at each time step, suitable for stiff systems. & Computationally less expensive since the solution progresses explicitly through time, applicable for simpler systems or those that are not stiff.\\\hline
        Convergence & Requires iterative methods for convergence, often using techniques like Newton-Raphson, Gauss-Seidel, or other matrix solvers. & Does not require iterative solving and typically proceeds directly to the solution at the next time step.\\\hline
    \end{tabular}
    \caption{Difference between Implicit and Explicit Solutions}
    \label{tab:my_label}
\end{table}

\section{Exercise}
Estimate the depth of the channel reach for different section using implicit algorithm.

4 Solution of Governing Equations

Solution of Governing Equations in computational Hydraulics are of two types i.e. Implicit and Explicit. Difference between these two are described below in table 1

Aspect	Implicit Solution	Explicit Solution
Numerical Stability	Stable for a wider range of time steps.	Can be conditionally stable based on the CFL (Courant-Friedrichs-Lewy) condition, limiting the time step for stability.
Time Dependency	Computationally more expensive due to solving nonlinear systems of equations at each time step, suitable for stiff systems.	Computationally less expensive since the solution progresses explicitly through time, applicable for simpler systems or those that are not stiff.
Convergence	Requires iterative methods for convergence, often using techniques like Newton-Raphson, Gauss-Seidel, or other matrix solvers.	Does not require iterative solving and typically proceeds directly to the solution at the next time step.

Table 1: Difference between Implicit and Explicit Solutions

5 Exercise

Estimate the depth of the channel reach for different section using implicit algorithm.

2

\begin{figure}[H]
  \centering

\includegraphics[width=1.0\linewidth]{image/Assignment_Channel.png}
  \caption{Channel with rough bed}
  \label{fig:enter-label}
\end{figure}

\end{document}

Figure 1: Channel with rough bed