Python Learning

# Data Type

## ****What are Data Types in Python?****

Data types in Python refer to the classification of data items. They define the type of value a variable can hold and determine the operations that can be performed on it. Python has several built-in data types that cater to various types of data handling needs.

## ****Python Data Types****

Here’s an overview of the main data types in Python with examples:

### ****Numeric Types****:

* + **int**: Integer values.

python

Copy code

x = 10 # Example of an integer

* + **float**: Decimal or floating-point numbers.

python

Copy code

y = 3.14 # Example of a float

* + **complex**: Complex numbers (a + bj, where j is the imaginary unit).

python

Copy code

z = 2 + 3j # Example of a complex number

### ****Sequence Types****:

* + **list**: Ordered, mutable collection of items.

python

Copy code

my\_list = [1, 2, 3, "hello"]

* + **tuple**: Ordered, immutable collection of items.

python

Copy code

my\_tuple = (1, 2, 3, "world")

* + **range**: Represents an immutable sequence of numbers.

python

Copy code

r = range(1, 5) # Represents [1, 2, 3, 4]

### ****Text Type****:

* + **str**: Sequence of Unicode characters (strings).

python

Copy code

text = "Hello, Python!"

### ****Set Types****:

* + **set**: Unordered collection of unique items.

python

Copy code

my\_set = {1, 2, 3, 3} # Result: {1, 2, 3}

* + **frozenset**: Immutable version of a set.

python

Copy code

frozen = frozenset([1, 2, 3])

### ****Mapping Type****:

* + **dict**: Collection of key-value pairs.

python

Copy code

my\_dict = {"name": "Alice", "age": 25}

### ****Boolean Type****:

* + **bool**: Represents True or False.

python

Copy code

is\_active = True

### ****None Type****:

* + **NoneType**: Represents a null or no value.

python

Copy code

value = None

## ****String Data Type****

Strings in Python are sequences of Unicode characters. You can work with strings in various ways:

### ****Literal Assignment****:

Strings can be assigned directly using single, double, or triple quotes:

python

Copy code

str1 = 'Hello'

str2 = "World"

str3 = '''This is a

multiline string.'''

### ****Construction Function****:

Strings can also be created using the str() constructor:

python

Copy code

str4 = str(123) # Converts the integer 123 to string '123'

### ****String**** Operations:

* + **Concatenation**: Combine two strings using +.

python

Copy code

result = "Hello" + " " + "World" # Output: 'Hello World'

* + **Repetition**: Repeat a string using \*.

python

Copy code

repeated = "Ha" \* 3 # Output: 'HaHaHa'

* + **Slicing**: Extract parts of a string.

python

Copy code

text = "Python"

print(text[0:3]) # Output: 'Pyt'

* + **Length**: Get the number of characters in a string.

python

Copy code

print(len("Hello")) # Output: 5

### ****String Methods****:

Strings in Python have many built-in methods:

* + upper(): Converts to uppercase.

python

Copy code

"hello".upper() # Output: 'HELLO'

* + lower(): Converts to lowercase.

python

Copy code

"HELLO".lower() # Output: 'hello'

* + strip(): Removes leading/trailing whitespace.

python

Copy code

" hello ".strip() # Output: 'hello'

* + split(): Splits a string into a list.

python

Copy code

"a,b,c".split(",") # Output: ['a', 'b', 'c']

* + join(): Joins elements of a list into a string.

python

Copy code

",".join(['a', 'b', 'c']) # Output: 'a,b,c'

## ****Key Notes****

* Strings are **immutable**, meaning you cannot change a string after it’s created. Instead, you create a new string.
* Strings can handle Unicode, making them suitable for a wide range of applications, including internationalization.

**Computational Molecular Science**

**(Python Coding)**

# Defining Absolute and Relative File paths

*We are* ***defining*** *the file paths in python and* ***not finding*** *the file path using python!!*

## ****Importing Libraries and Setting Relative File Path****

python

Copy code

import os

# Define a relative file path

ethanol\_file = os.path.join('data', 'outfiles', 'ethanol.out')

print(ethanol\_file)

#### ****Explanation:****

* **import os**:
  + The os module in Python provides functions for interacting with the operating system, such as handling file paths.
* **os.path.join()**:
  + This function creates a **file path** by joining directory names and file names in a way that is compatible with your operating system.
  + For example, on Windows, it creates paths with backslashes (\), while on Linux/Mac, it uses forward slashes (/).
* **What is happening here?**
  + 'data' and 'outfiles' are directories, and 'ethanol.out' is the file name.
  + The function combines them to produce a relative file path: data/outfiles/ethanol.out.
  + The relative path is printed so you can verify it.

## Absolute File Path

python

CopyEdit

absolute\_path = os.path.abspath(ethanol\_file)

print(f"Absolute Path: {absolute\_path}")

* **os.path.abspath()**:
  + Converts a relative file path (like data/outfiles/ethanol.out) into an absolute file path, which includes the full path starting from the root directory.

## ****Relative Path****:

Specifies the location of a file relative to the current working directory of your script or notebook. Example:

python

Copy code

ethanol\_file = os.path.join('data', 'outfiles', 'ethanol.out')

Here, ethanol\_file points to a location relative to where your Python program is running.

## Absolute Path:

Specifies the full path to the file, starting from the root directory. Example:

python

Copy code

ethanol\_file = "/Users/YOUR\_USER\_NAME/Desktop/data/outfiles/ethanol.out"

This explicitly defines where the file is located on your system, regardless of where your script is running.

You don't need to convert a relative path to an absolute path every time. If you already know the absolute path to the file, you can directly define and use it.

### ****Code Example of Absolute File Path****

If you know the absolute path to your file, you can skip the relative path entirely:

python

Copy code

ethanol\_file = "/Users/YOUR\_USER\_NAME/Desktop/data/outfiles/ethanol.out"

# Print the absolute file path

print(f"Absolute Path: {ethanol\_file}")

or simply

print(ethanol\_file)

### ****Explaining the**** print ****Command and**** f ****Strings****

1. **print()**:
   * Outputs text or variables to the console.
   * In this case:

python

Copy code

print(f"Absolute Path: {absolute\_path}")

This prints the message "Absolute Path:" followed by the value stored in the variable absolute\_path.

1. **f String (Formatted String Literals)**:
   * Introduced in Python 3.6, an f string allows you to embed variables directly inside a string using curly braces {}.
   * Syntax:

python

Copy code

f"Text {variable\_name}"

* + Example:

python

Copy code

name = "Ethanol"

print(f"File Name: {name}")

Output:

arduino

Copy code

File Name: Ethanol

1. **Benefits of f Strings**:
   * More readable and concise compared to older methods like str.format() or string concatenation.
   * Example of older method:

python

Copy code

print("Absolute Path: {}".format(absolute\_path))

* + With f strings, the same can be written as:

python

Copy code

print(f"Absolute Path: {absolute\_path}")

### ****Understanding How**** ethanol\_file ****is Defined****

In the example:

python

CopyEdit

absolute\_path = os.path.abspath(ethanol\_file)

print(absolute\_path)

The variable ethanol\_file **must already be defined** before this code is executed. If it's not defined, Python will raise a NameError, saying that the variable ethanol\_file is not recognized.

### ****Where**** ethanol\_file ****is Defined****

Earlier in the code, ethanol\_file was defined using a **relative path**:

python

CopyEdit

ethanol\_file = os.path.join('data', 'outfiles', 'ethanol.out')

Here, the relative path 'data/outfiles/ethanol.out' was assigned to the variable ethanol\_file. This is what the later code (os.path.abspath(ethanol\_file)) relies on to determine the absolute path.

### ****Is a Relative Path Necessary for Absolute Path Code?****

No, a **relative path is not mandatory** for defining an absolute path. You can directly define the variable ethanol\_file as an absolute path if you already know it. For example:

python

CopyEdit

ethanol\_file = "/Users/YOUR\_USER\_NAME/Desktop/data/outfiles/ethanol.out"

absolute\_path = os.path.abspath(ethanol\_file)

print(absolute\_path)

In this case:

* ethanol\_file contains an absolute path, and calling os.path.abspath() on it is redundant because it's already absolute.
* Python does not require that the file path starts as relative; it just needs a defined file path in the variable.

### ****Key Points to Remember****

1. **Variables Must Be Defined**:
   * The variable ethanol\_file must be defined somewhere in your code before you try to use it. If it's not defined, Python will throw an error.
2. **Relative vs. Absolute Paths**:
   * A relative path is often defined first because it is more flexible for scripts that may run on different systems or environments.
   * An absolute path can be used directly if you know the exact location of the file.
3. **When to Use os.path.abspath()**:
   * If you only have a relative path, os.path.abspath() can convert it into an absolute path.
   * If you already have an absolute path, os.path.abspath() doesn't add any value.

### Examples

#### ****Defining Using a Relative Path****

python

CopyEdit

# Relative path definition

ethanol\_file = os.path.join('data', 'outfiles', 'ethanol.out')

# Convert to absolute path

absolute\_path = os.path.abspath(ethanol\_file)

print(absolute\_path)

#### ****Defining Using an Absolute Path Directly****

python

CopyEdit

# Direct absolute path definition

ethanol\_file = "/Users/YOUR\_USER\_NAME/Desktop/data/outfiles/ethanol.out"

# Print the absolute path (no conversion needed)

print(ethanol\_file)

Both methods work, but the choice depends on the context of your project. If you're sharing your code with others, a **relative path** is better since others may not have the same directory structure as you.

### ****Why Define a Relative Path?****

* When working on a project, your files are often organized in a directory structure relative to the project root.
* If you hardcode an **absolute path**, like this:

python

CopyEdit

ethanol\_file = "/Users/YOUR\_USER\_NAME/Desktop/data/outfiles/ethanol.out"

the code will work **only on your system** because the full path may not exist on someone else's machine.

* By defining a **relative path**, like this:

python

CopyEdit

ethanol\_file = os.path.join('data', 'outfiles', 'ethanol.out')

you can share the script with others, and it will work as long as they maintain the same relative structure in their project directory.

### ****Why Use**** os.path.abspath()****?****

* The os.path.abspath() function converts a relative path into an **absolute path**, which is sometimes required by certain operations or libraries that need the full path.
* It ensures your script can work no matter where the current working directory is set. For example:

python

CopyEdit

absolute\_path = os.path.abspath(ethanol\_file)

print(absolute\_path)

If your working directory is /Users/your\_user\_name/project, this might output:

kotlin

CopyEdit

/Users/your\_user\_name/project/data/outfiles/ethanol.out

### ****Why Is This Useful If We Already Know the Path?****

1. **Flexibility**: Using a relative path combined with os.path.abspath() ensures the script works even if:
   * The project is moved to another directory.
   * Someone else is running the script on their system.
2. **Portability**: Hardcoding an absolute path ties your script to your specific machine and directory structure. By using relative paths, your code works in different environments without modification.
3. **Dynamic Handling**: Sometimes, you may not know the exact absolute path of a file, but you know its relative position in the project. For example:
   * Your script may dynamically discover or generate files in a specific directory.
   * You can avoid hardcoding paths by building them dynamically.

### ****Conclusion****

* If you're running the code **only on your machine**, you can directly define an absolute path and skip all this.
* If you're writing code for reuse or collaboration, defining a **relative path** and converting it to an **absolute path** ensures compatibility across systems, making your code portable and robust.

# ****Reading the File****

python

CopyEdit

# Open the file and read its contents

with open(ethanol\_file, "r") as outfile:

data = outfile.readlines()

# Check how many lines are in the file

print(f"The file contains {len(data)} lines.")

## ****Explanation:****

* **with open(ethanol\_file, "r")**:
  + Opens the file located at the path stored in ethanol\_file.
  + The "r" argument stands for **read mode** (we're only reading the file, not writing or modifying it).
* **as outfile**:
  + Creates a **file object** (here named outfile) to interact with the file.
* **outfile.readlines()**:
  + Reads all the lines of the file and stores them in a list called data.
  + Each line of the file becomes a separate element of the list.
* **len(data)**:
  + Calculates the number of lines in the file by finding the length of the data list.

# Plotting and Data Visualization

When you use matplotlib.pyplot.plot() with a single argument, it assumes the following:

### 1. ****Interpreting the Single Argument****

* The argument you provide, data[:,1], is treated as the **Y-axis values**.
* The **X-axis values** are automatically generated as a sequence of integers starting from 0, incrementing by 1 for each data point. Essentially, X = [0, 1, 2, ..., N-1], where N is the number of elements in your Y-axis data.

### 2. ****How Python Determines Axes****

* **X-axis**: Automatically generated as range(len(Y)) (e.g., [0, 1, 2, ...]).
* **Y-axis**: Explicitly defined by the argument you pass (data[:,1]).

### Example in Your Case

Let’s break it down:

python

CopyEdit

matplotlib.pyplot.plot(data[:,1])

* data[:,1]: Contains all rows of the second column, so it gives an array of values that will be plotted on the Y-axis.
* Python generates the X-axis automatically, which will be [0, 1, 2, ..., len(data[:,1]) - 1].

### Why Does This Happen?

matplotlib.pyplot.plot() is designed for quick plotting. In many cases, data has a natural order (e.g., time series, index, or sequence), and the automatic generation of the X-axis values makes it easier for users who only care about visualizing trends.

### Explicit X and Y Axes

If you want to define both X and Y axes explicitly, you can pass two arguments to plot():

python

CopyEdit

matplotlib.pyplot.plot(x\_values, y\_values)

This ensures you control exactly what gets plotted on both axes.

### ****Multiple Plots with Subplot****

* **Meaning:** This introduces the concept of subplots in Matplotlib, which allows you to create a single figure containing multiple plots.

### 2. ****fig, ax = plt.subplots(nrows, ncolumns)****

* **plt.subplots:** A Matplotlib function that creates a grid of subplots within a single figure.
* **fig:** Represents the overall figure container for all the subplots.
* **ax:** A NumPy array or list of axes objects, each corresponding to an individual subplot.
* **nrows:** Number of rows in the grid.
* **ncolumns:** Number of columns in the grid.

### 3. ****Subplot allows for you to lay plots out on a grid.****

* **Meaning:** Subplots enable arranging multiple plots within a single figure in a structured grid.

### 4. ****fig, ax = plt.subplots(len(headers)-1, 1)****

* **len(headers)-1:** The number of rows in the grid is set to the number of columns in the dataset (minus the header row).
* **1:** Specifies one column, making the layout a single column with multiple rows.
* **Purpose:** Creates multiple vertical plots (stacked in one column) for each data column.

### 5. ****This should show four empty plots...****

* **Meaning:** If len(headers) - 1 equals 4, this will create a single figure containing 4 empty subplots.

### 6. ****fig is our figure, while ax is a numpy array that represents our figure axes.****

* **fig:** The overarching figure object containing all subplots.
* **ax:** A NumPy array (1D or 2D) where each element is an individual subplot axis.

### 7. ****ax.shape (4,)****

* **Meaning:** ax has 4 axes (1D array) because we created 4 subplots arranged in one column.

### 8. ****fig2, ax2 = plt.subplots(2, 2)****

* **Meaning:** Creates a 2x2 grid of subplots.
* **fig2:** The figure containing the subplots.
* **ax2:** A 2D NumPy array of shape (2, 2) representing 4 subplot axes in a grid.

### 9. ****ax2.shape (2, 2)****

* **Meaning:** Confirms the shape of the ax2 array (a 2D grid with 2 rows and 2 columns).

### 10. ****ax[0].plot(data[0::100, 0], data[0::100, col])****

* **ax[0]:** Refers to the first subplot in the 1D array ax.
* **.plot:** Plots the data on the specified axis.
* **data[0::100, 0]:** Selects every 100th row from column 0 for the x-axis values.
* **data[0::100, col]:** Selects every 100th row from the specified column for the y-axis values.

### 11. ****fig****

* **Purpose:** Displays the updated figure with the plot added.

### 12. ****ax2[0,0].plot(data[0::100, 0], data[0::100, col])****

* **ax2[0,0]:** Refers to the first subplot in the 2D array ax2 (top-left subplot in a 2x2 grid).
* **.plot:** Plots the data on the specified axis.
* **data[0::100, 0]:** X-axis data (every 100th row from column 0).
* **data[0::100, col]:** Y-axis data (every 100th row from the specified column).

### 13. ****fig2****

* **Purpose:** Displays the updated figure fig2 containing the new plot in the specified subplot.

### Key Takeaways:

1. **Subplots provide a structured way to include multiple plots in a single figure.**
2. **The fig object manages the entire figure, while ax (or ax2) manages individual subplots.**
3. **You use slicing (1D or 2D, depending on the layout) to access specific subplots for customization.**
4. **The shape of ax or ax2 helps identify the grid layout and guides indexing.**

# Writing Functions

### ****File Import and Reading****

python

CopyEdit

import os

import numpy

* import os: Imports the os module, which allows interaction with the operating system, such as file paths.
* import numpy: Imports the numpy library for numerical operations.

python

CopyEdit

xyz\_file\_path = os.path.join('CMS Workshp (MolSSI data)', 'water.xyz')

* xyz\_file\_path: Variable storing the file path to the water.xyz file.
* os.path.join(...): Combines directory ('CMS Workshp (MolSSI data)') and file name ('water.xyz') into a valid path for the operating system.

python

CopyEdit

xyx\_file = numpy.genfromtxt(fname=xyz\_file\_path, skip\_header=2, dtype='unicode')

* xyx\_file: Variable storing the data read from the file.
* numpy.genfromtxt: Reads data from a text file.
  + fname=xyz\_file\_path: The file path to read the data from.
  + skip\_header=2: Skips the first two lines of the file (e.g., file header).
  + dtype='unicode': Reads the data as strings (useful for mixed data types like molecule names and coordinates).

### ****Data Extraction and Conversion****

python

CopyEdit

molecule = xyx\_file[:, 0]

* molecule: Extracts the first column from xyx\_file, which contains atom names (e.g., O, H1, H2).
* xyx\_file[:, 0]: Takes all rows (:) and the first column (0) of the array.

python

CopyEdit

coordinates = xyx\_file[:, 1:]

* coordinates: Extracts the remaining columns (coordinates) from xyx\_file.
* xyx\_file[:, 1:]: Takes all rows (:) and all columns starting from the second (1:).

python

CopyEdit

coordinates = coordinates.astype(numpy.float64)

* Converts the coordinates array (currently strings) into numeric float64 type for mathematical calculations.

### ****Coordinate Explanation****

python

CopyEdit

# 1st column is x\_coordinates, 2nd column is y, 3rd column is z.

# 1st row is for O\_atoms, 2nd for H1, and 3rd for H2

* Explains how the coordinates array is structured:
  + Columns: x, y, z coordinates.
  + Rows: Correspond to atoms O, H1, and H2.

### ****Distance Function****

python

CopyEdit

def distances(atom1\_coord, atom2\_coord):

* def: Defines a function named distances.
* distances: The function calculates the Euclidean distance (bond length) between two atoms.
* atom1\_coord, atom2\_coord: Inputs, each representing the x, y, z coordinates of an atom.

python

CopyEdit

x\_distances = atom1\_coord[0] - atom2\_coord[0]

y\_distances = atom1\_coord[1] - atom2\_coord[1]

z\_distances = atom1\_coord[2] - atom2\_coord[2]

* x\_distances: Difference between the x-coordinates of atom1\_coord and atom2\_coord.
* y\_distances: Difference between the y-coordinates.
* z\_distances: Difference between the z-coordinates.

python

CopyEdit

bond\_len = numpy.sqrt(x\_distances\*\*2 + y\_distances\*\*2 + z\_distances\*\*2)

* bond\_len: Computes the Euclidean distance using the formula: Distance=(x2−x1)2+(y2−y1)2+(z2−z1)2\text{Distance} = \sqrt{(x\_2 - x\_1)^2 + (y\_2 - y\_1)^2 + (z\_2 - z\_1)^2}Distance=(x2​−x1​)2+(y2​−y1​)2+(z2​−z1​)2​

python

CopyEdit

return bond\_len

* Returns the calculated bond length to the calling code.

### ****Row Number Calculation****

python

CopyEdit

row\_num = len(coordinates[:, 0])

* row\_num: Calculates the number of rows in the coordinates array.
* coordinates[:, 0]: Extracts the first column (x-coordinates) for all rows.
* len(...): Returns the number of rows.

### ****Nested Loops****

python

CopyEdit

for i\_1 in range(0, row\_num):

for i\_2 in range(0, row\_num):

* for i\_1 in range(0, row\_num): Loops through the rows of coordinates for the first atom (i\_1).
* for i\_2 in range(0, row\_num): Loops through the rows of coordinates for the second atom (i\_2).

python

CopyEdit

if i\_1 < i\_2:

* Ensures that each pair of atoms is only considered once (avoids duplicates or self-pairing).

### ****Bond Length Calculation****

python

CopyEdit

bond\_len = distances(coordinates[i\_1], coordinates[i\_2])

* Calls the distances function to calculate the bond length between atoms at rows i\_1 and i\_2.
* coordinates[i\_1]: Retrieves the x, y, z coordinates of atom i\_1.
* coordinates[i\_2]: Retrieves the x, y, z coordinates of atom i\_2.

### ****Filter and Print Bond Length****

python

CopyEdit

if bond\_len <= 1.5 and bond\_len > 0:

real\_bond\_len = bond\_len

print(F'{molecule[i\_1]} to {molecule[i\_2]}:', real\_bond\_len)

* if bond\_len <= 1.5 and bond\_len > 0: Filters bond lengths to include only valid bonds:
  + Must be less than or equal to 1.5 Å.
  + Must be greater than 0 Å (to exclude self-pairing).
* real\_bond\_len = bond\_len: Stores the bond length for display.
* print(...): Prints the bond length and the atom pair, using formatted strings (F'{...}'):
  + molecule[i\_1]: Atom name for i\_1 (e.g., O, H1).
  + molecule[i\_2]: Atom name for i\_2.

### ****What This Does****

This script:

1. Reads atomic coordinates from a file.
2. Calculates bond lengths between pairs of atoms.
3. Filters out bonds longer than 1.5 Å or zero-length bonds.
4. Prints valid bond lengths and the corresponding atom pairs.

**I am not able to understand that first we defined a function "distances(atom1\_coord, atom2\_coord)" as a function of atom1\_coord atom2\_coord. Then defined the same function as "bond\_len = distances(coordinates[i\_1] , coordinates[i\_2])". Why do we need to do it so complexly? isn't the previous code with utilizing function the bond length is simple to use?**

Let me break this down and explain why a **function** is used in this case and why it might seem "complex" compared to your earlier code.

### ****Your Concern: Why is it more complex?****

At first glance, yes, the earlier code without a function seems simpler because everything is written directly in one block:

python

CopyEdit

for i\_1 in range(0, col\_num):

for i\_2 in range(0, col\_num):

if i\_1 < i\_2:

x\_distances = coordinates[i\_1, 0] - coordinates[i\_2, 0]

y\_distances = coordinates[i\_1, 1] - coordinates[i\_2, 1]

z\_distances = coordinates[i\_1, 2] - coordinates[i\_2, 2]

bond\_len = numpy.sqrt(x\_distances\*\*2 + y\_distances\*\*2 + z\_distances\*\*2)

print(F'{molecule[i\_1]} to {molecule[i\_2]}:', bond\_len)

Here, **everything happens in one place**, and you can clearly see the calculation steps for bond length.

### ****Why Use a Function?****

Using a function like distances(atom1\_coord, atom2\_coord) adds flexibility, modularity, and reusability. Here's why it is beneficial:

#### 1. ****Avoid Repetition****

Imagine you need to calculate bond lengths in multiple places or for different datasets in the future. Without a function, you would need to **repeat the same code** for calculating distances, which is prone to errors and makes your script unnecessarily long.

With a function:

python

CopyEdit

def distances(atom1\_coord, atom2\_coord):

x\_distances = atom1\_coord[0] - atom2\_coord[0]

y\_distances = atom1\_coord[1] - atom2\_coord[1]

z\_distances = atom1\_coord[2] - atom2\_coord[2]

bond\_len = numpy.sqrt(x\_distances\*\*2 + y\_distances\*\*2 + z\_distances\*\*2)

return bond\_len

The logic for calculating distances is **written once**, and you can reuse it whenever and wherever needed.

#### 2. ****Better Readability****

The main loop becomes cleaner because the bond length calculation is **abstracted away** into the function. Compare these two snippets:

Without a function:

python

CopyEdit

for i\_1 in range(0, col\_num):

for i\_2 in range(0, col\_num):

if i\_1 < i\_2:

x\_distances = coordinates[i\_1, 0] - coordinates[i\_2, 0]

y\_distances = coordinates[i\_1, 1] - coordinates[i\_2, 1]

z\_distances = coordinates[i\_1, 2] - coordinates[i\_2, 2]

bond\_len = numpy.sqrt(x\_distances\*\*2 + y\_distances\*\*2 + z\_distances\*\*2)

print(F'{molecule[i\_1]} to {molecule[i\_2]}:', bond\_len)

With a function:

python

CopyEdit

for i\_1 in range(0, col\_num):

for i\_2 in range(0, col\_num):

if i\_1 < i\_2:

bond\_len = distances(coordinates[i\_1], coordinates[i\_2])

print(F'{molecule[i\_1]} to {molecule[i\_2]}:', bond\_len)

The **second version** makes the logic of the main loop easier to follow because the bond length calculation is **encapsulated** in a separate function.

#### 3. ****Encapsulation and Debugging****

By using a function, you encapsulate the bond length calculation. If there's an error in the bond length calculation, you only need to **debug the function**, not the entire script.

#### 4. ****Generalization****

The function makes the code **more general**. For example:

* You can calculate distances between any two points, not just those in coordinates.
* The same function can be used in other scripts or for other datasets without modification.

Example:

python

CopyEdit

atom1 = [0.0, 0.0, 0.0] # x, y, z coordinates of Atom 1

atom2 = [1.0, 1.0, 1.0] # x, y, z coordinates of Atom 2

print("Distance between atom1 and atom2:", distances(atom1, atom2))

### ****Why Call the Function Like This?****

python

CopyEdit

bond\_len = distances(coordinates[i\_1], coordinates[i\_2])

This simply **calls the function** distances and passes the x, y, z coordinates of two atoms (coordinates[i\_1] and coordinates[i\_2]) as arguments.

* coordinates[i\_1]: Gets the row corresponding to atom i\_1 (e.g., [x1, y1, z1]).
* coordinates[i\_2]: Gets the row corresponding to atom i\_2 (e.g., [x2, y2, z2]).

The function then uses these coordinates to compute the bond length.

### ****Comparison: Code Without a Function****

Yes, the earlier code **without a function** is simpler if:

* You only need to calculate bond lengths in one place.
* You’re writing a small, single-use script.

But, if the script grows in complexity or if you need to reuse the distance calculation logic in other places, **using a function becomes more efficient**.

### ****Takeaway****

Using a function like distances(atom1\_coord, atom2\_coord) may seem more complex initially, but it:

1. Makes the code reusable and modular.
2. Improves readability and maintainability.
3. Encapsulates logic for easier debugging and flexibility.

For a small, one-time script, the direct approach works. But for **scalable, reusable, and maintainable** code, functions are the way to go.

**import** **numpy**

**import** **os**

**def** calculate\_distance(atom1\_coord, atom2\_coord):

"""

Calculate the distance between two three-dimensional points.

"""

x\_distance = atom1\_coord[0] - atom2\_coord[0]

y\_distance = atom1\_coord[1] - atom2\_coord[1]

z\_distance = atom1\_coord[2] - atom2\_coord[2]

bond\_length\_12 = numpy.sqrt(x\_distance \*\* 2 + y\_distance \*\* 2 + z\_distance \*\* 2)

**return** bond\_length\_12

**def** bond\_check(atom\_distance, minimum\_length=0, maximum\_length=1.5):

"""Check if a distance is a bond based on a minimum and maximum bond length"""

**if** atom\_distance > minimum\_length **and** atom\_distance <= maximum\_length:

**return** True

**else**:

**return** False

**def** open\_xyz(filename):

"""

Open and read an xyz file. Returns tuple of symbols and coordinates.

"""

xyz\_file = numpy.genfromtxt(fname=filename, skip\_header=2, dtype='unicode')

symbols = xyz\_file[:,0]

coord = (xyz\_file[:,1:])

coord = coord.astype(numpy.float)

**return** symbols, coord

**def** print\_bonds(atom\_symbols, atom\_coordinates):

"""

Prints atom symbols and bond length for a set of atoms.

"""

num\_atoms = len(atom\_symbols)

**for** num1 **in** range(0, num\_atoms):

**for** num2 **in** range(0, num\_atoms):

**if** num1 < num2:

bond\_length\_12 = calculate\_distance(atom\_coordinates[num1], atom\_coordinates[num2])

**if** bond\_check(bond\_length\_12) **is** True:

**print**(F'**{**atom\_symbols[num1]**}** to **{**atom\_symbols[num2]**}** : **{**bond\_length\_12:.3f**}**')

### ****1. Why do we need the**** print\_bonds ****function?****

The purpose of the print\_bonds function is **modularity and clarity**. Here’s why it is added:

1. **Separation of Concerns:**
   * Each function in the code has a single responsibility:
     + calculate\_distance: Calculates bond distances.
     + bond\_check: Verifies if the distance qualifies as a bond.
     + open\_xyz: Reads and processes the .xyz file.
     + print\_bonds: Handles the printing of bond information in a readable format.
   * By creating print\_bonds, we separate the logic of calculating and checking bonds from the task of displaying the results.
2. **Improved Code Reusability:**
   * print\_bonds makes the code easier to reuse and extend. If you later decide to format the bond information differently or save it to a file instead of printing it, you only need to modify this function.
3. **Enhanced Readability:**
   * The logic of printing bond lengths is contained in one place, making the rest of the code cleaner and easier to follow.
4. **Scalability:**
   * As your project grows, separating functionality into modular components like print\_bonds ensures your code remains manageable and easier to debug.

### ****2. Why is your second code not printing bond lengths?****

There are several issues with your second code. Let’s go through them step by step:

#### ****Error 1: Incorrect**** bond\_check ****Function Logic****

In your bond\_check function:

python

CopyEdit

def bond\_check(bond\_len, min\_length = 0, max\_length = 1.5):

if bond\_len > 0 and bond\_len <= 1.5:

True

else:

False

You are not **returning** anything from the function. The True or False is just written in isolation, so the function essentially returns None. The correct implementation should include return:

python

CopyEdit

def bond\_check(bond\_len, min\_length=0, max\_length=1.5):

if min\_length < bond\_len <= max\_length:

return True

else:

return False

#### ****Error 2: Incorrect Distance Formula****

In your atomic\_dist function:

python

CopyEdit

bond\_len = numpy.square(x\_distances\*\*2 + y\_distances\*\*2 + z\_distances\*\*2)

The formula is incorrect. The distance should be calculated using the **square root**, not numpy.square. The correct formula is:

python

CopyEdit

bond\_len = numpy.sqrt(x\_distances\*\*2 + y\_distances\*\*2 + z\_distances\*\*2)

#### ****Error 3: Incorrect**** bond\_check ****Call in the Loop****

In the for loop:

python

CopyEdit

if bond\_check is True:

This line is incorrect because you are checking if the **function itself** is True, instead of calling the function with the bond length as input. The correct usage is:

python

CopyEdit

if bond\_check(bond\_len):

#### ****Error 4: Missing**** coordinates ****and**** molecule ****Assignment Before Loop****

You attempt to calculate bond lengths with this loop:

python

CopyEdit

row\_num = len(coordinates[:,0])

But you have not assigned values to coordinates or molecule in the script. The variables should come from your open\_xyz function.

You need to first call open\_xyz to get these variables:

python

CopyEdit

molecule, coordinates = open\_xyz(xyz\_file\_path)

### ****Corrected Code****

Here’s the corrected version of your code:

python

CopyEdit

import os

import numpy

# Function to open and parse an XYZ file

def open\_xyz(xyz\_file\_path):

"""

Open an XYZ file and return molecule symbols and coordinates.

"""

any\_xyz\_file = numpy.genfromtxt(fname=xyz\_file\_path, skip\_header=2, dtype='unicode')

molecule = any\_xyz\_file[:, 0]

coordinates = any\_xyz\_file[:, 1:].astype(numpy.float64)

return molecule, coordinates

# Function to calculate the bond length between two atoms

def atomic\_dist(atom1\_coord, atom2\_coord):

"""

Calculate the distance between two three-dimensional points.

"""

x\_distances = atom1\_coord[0] - atom2\_coord[0]

y\_distances = atom1\_coord[1] - atom2\_coord[1]

z\_distances = atom1\_coord[2] - atom2\_coord[2]

bond\_len = numpy.sqrt(x\_distances\*\*2 + y\_distances\*\*2 + z\_distances\*\*2)

return bond\_len

# Function to check if a distance qualifies as a bond

def bond\_check(bond\_len, min\_length=0, max\_length=1.5):

"""

Check if a bond length is valid based on minimum and maximum lengths.

"""

if min\_length < bond\_len <= max\_length:

return True

else:

return False

# Main script to calculate and print bond lengths

xyz\_file\_path = os.path.join('CMS Workshp (MolSSI data)', 'benzene.xyz')

molecule, coordinates = open\_xyz(xyz\_file\_path)

row\_num = len(coordinates[:, 0])

for i\_1 in range(row\_num):

for i\_2 in range(row\_num):

if i\_1 < i\_2: # Avoid redundant calculations

bond\_len = atomic\_dist(coordinates[i\_1], coordinates[i\_2])

if bond\_check(bond\_len): # Call bond\_check correctly

print(F'{molecule[i\_1]} to {molecule[i\_2]}: {bond\_len:.3f}')

### ****Explanation of Fixes****

1. **Corrected bond\_check to return True or False.**
2. **Fixed the distance formula in atomic\_dist to use numpy.sqrt.**
3. **Correctly called the bond\_check function inside the loop.**
4. **Properly assigned molecule and coordinates using the open\_xyz function.**

When you run this corrected code, it should correctly calculate and print the bond lengths for atoms in your .xyz file.

4o

##### You said:

molecule, coordinates = open\\_xyz(xyz\\_file\\_path)

I dont get that why do I need to add this line, while I was not adding it before and I have already defined them in "open\_xyz(xyz\_file\_path)" function

###### ChatGPT said:

ChatGPT

The reason you need to explicitly add this line:

python

CopyEdit

molecule, coordinates = open\_xyz(xyz\_file\_path)

is because **functions in Python don't automatically make variables globally accessible**. When you define variables (molecule and coordinates) inside the open\_xyz function, they are local to that function. These variables do not exist outside the function unless you explicitly return them and assign them to variables in the outer scope.

### Why You Didn’t Need It Before

In the earlier code snippets, you might have skipped encapsulating the logic in a function like open\_xyz. Instead, you probably worked directly with the variables at the global level. For example:

python

CopyEdit

xyz\_file = numpy.genfromtxt(fname=xyz\_file\_path, skip\_header=2, dtype='unicode')

molecule = xyz\_file[:, 0]

coordinates = xyz\_file[:, 1:].astype(numpy.float64)

Here, molecule and coordinates were defined directly in the main body of the script, so they were accessible globally. This works, but it’s **not modular or reusable**, which is why breaking it into a function (open\_xyz) is better.

### Why You Need the Line Now

When you encapsulate the code to read the .xyz file into the open\_xyz function, the molecule and coordinates are returned from the function. You need to explicitly assign them in your main script like this:

python

CopyEdit

molecule, coordinates = open\_xyz(xyz\_file\_path)

This assignment makes the variables available in the main script.

### Why It’s Better to Use a Function

1. **Reusability:** You can reuse the open\_xyz function to read multiple .xyz files without rewriting the logic every time.
2. **Readability:** Functions make your code easier to understand by grouping related tasks together.
3. **Separation of Concerns:** By defining open\_xyz, you separate file-handling logic from bond-length calculations, making the code modular.

### What Happens If You Skip the Assignment

If you don’t assign the return values of open\_xyz to molecule and coordinates, these variables will not exist in the scope of your main script, and Python will throw an error like:

plaintext

CopyEdit

NameError: name 'molecule' is not defined

This is because molecule and coordinates only exist **inside** the open\_xyz function unless you explicitly assign their returned values to variables in the main script.

If you’re still confused, think of a function as a black box. It doesn’t automatically share its internal data with the outside world—you have to **request** the data by using return and explicitly capturing the output.

**If I want to have this output, what should I do "Printing bonds for water. O to H1 : 0.969 O to H2 : 0.969 Printing bonds for benzene. C to H : 1.088 C to C : 1.403 C to C : 1.403 C to H : 1.088 C to C : 1.403 C to H : 1.088 C to C : 1.403 C to H : 1.088 C to C : 1.403 C to H : 1.088 C to C : 1.403 C to H : 1.088"**

To achieve the desired output where bonds are calculated and printed for multiple .xyz files (e.g., water and benzene), you can organize your script as follows:

### Complete Code for the Desired Output

python

CopyEdit

import os

import numpy

def open\_xyz(xyz\_file\_path):

"""

Open and read an xyz file. Returns tuple of atom symbols and coordinates.

"""

xyz\_file = numpy.genfromtxt(fname=xyz\_file\_path, skip\_header=2, dtype='unicode')

molecule = xyz\_file[:, 0]

coordinates = xyz\_file[:, 1:].astype(numpy.float64)

return molecule, coordinates

def calculate\_distance(atom1\_coord, atom2\_coord):

"""

Calculate the distance between two three-dimensional points.

"""

x\_distance = atom1\_coord[0] - atom2\_coord[0]

y\_distance = atom1\_coord[1] - atom2\_coord[1]

z\_distance = atom1\_coord[2] - atom2\_coord[2]

bond\_length = numpy.sqrt(x\_distance\*\*2 + y\_distance\*\*2 + z\_distance\*\*2)

return bond\_length

def bond\_check(bond\_length, minimum\_length=0, maximum\_length=1.5):

"""

Check if a distance is a bond based on a minimum and maximum bond length.

"""

return minimum\_length < bond\_length <= maximum\_length

def print\_bonds(atom\_symbols, atom\_coordinates):

"""

Prints atom symbols and bond lengths for a set of atoms.

"""

num\_atoms = len(atom\_symbols)

for num1 in range(0, num\_atoms):

for num2 in range(0, num\_atoms):

if num1 < num2: # Avoid duplicates and self-comparisons

bond\_length = calculate\_distance(atom\_coordinates[num1], atom\_coordinates[num2])

if bond\_check(bond\_length): # Check if the bond is valid

print(F'{atom\_symbols[num1]} to {atom\_symbols[num2]} : {bond\_length:.3f}')

# Main script to handle multiple files

xyz\_files = {

"water": os.path.join('CMS Workshp (MolSSI data)', 'water.xyz'),

"benzene": os.path.join('CMS Workshp (MolSSI data)', 'benzene.xyz')

}

for molecule\_name, file\_path in xyz\_files.items():

print(F'Printing bonds for {molecule\_name}.')

symbols, coordinates = open\_xyz(file\_path)

print\_bonds(symbols, coordinates)

### Explanation of the Code

1. **Function Definitions:**
   * open\_xyz: Reads .xyz files and extracts atom symbols and coordinates.
   * calculate\_distance: Computes the Euclidean distance between two atoms.
   * bond\_check: Determines if a calculated bond length is within valid bounds.
   * print\_bonds: Iterates through all atom pairs, calculates bond lengths, and prints valid bonds.
2. **Iterating Over Multiple Files:**
   * A dictionary (xyz\_files) maps molecule names to their respective .xyz file paths.
   * The script loops through each file, printing the bonds for each molecule using print\_bonds.
3. **Formatting Output:**
   * Each molecule's bonds are labeled (e.g., "Printing bonds for water").
   * Bond lengths are formatted to three decimal places using F-string formatting.

### Output Example

Assuming water.xyz and benzene.xyz are in the specified paths, running this script will produce:

plaintext

CopyEdit

Printing bonds for water.

O to H1 : 0.969

O to H2 : 0.969

Printing bonds for benzene.

C to H : 1.088

C to C : 1.403

C to C : 1.403

C to H : 1.088

C to C : 1.403

C to H : 1.088

C to C : 1.403

C to H : 1.088

C to C : 1.403

C to H : 1.088

C to C : 1.403

C to H : 1.088

### Key Changes to Your Original Code

1. **Dictionary for Multiple Files:** A dictionary (xyz\_files) maps file names to paths, enabling batch processing.
2. **Generalization:** The print\_bonds function handles atom symbols and coordinates for any molecule.
3. **bond\_check Fix:** The bond\_check function now **returns** True or False (instead of having no return value, which defaulted to None).

This structure ensures the code is modular, reusable, and outputs the required bond data for any given .xyz file.

**\# Main script to handle multiple files xyz\\_files = { "water": os.path.join('CMS Workshp (MolSSI data)', 'water.xyz'), "benzene": os.path.join('CMS Workshp (MolSSI data)', 'benzene.xyz') } for molecule\\_name, file\\_path in xyz\\_files.items(): print(F'Printing bonds for {molecule\\_name}.') symbols, coordinates = open\\_xyz(file\\_path) print\\_bonds(symbols, coordinates)**

**Explain the Main script to handle multiple files in detail line by line**

Here’s a detailed explanation of the **main script** that handles multiple files, line by line:

### Code:

python

CopyEdit

xyz\_files = {

"water": os.path.join('CMS Workshp (MolSSI data)', 'water.xyz'),

"benzene": os.path.join('CMS Workshp (MolSSI data)', 'benzene.xyz')

}

#### ****Explanation:****

1. **xyz\_files = {...}**:
   * A dictionary is defined where:
     + **Keys** represent the names of the molecules ("water" and "benzene").
     + **Values** are file paths to the corresponding .xyz files containing atomic coordinates and atom types.
   * This approach allows us to store and process multiple files in an organized manner.
2. **os.path.join(...)**:
   * This function ensures file paths are constructed correctly for the operating system in use.
     + 'CMS Workshp (MolSSI data)' is the folder where the .xyz files are located.
     + 'water.xyz' and 'benzene.xyz' are the names of the .xyz files.
   * The result is platform-independent paths like:
     + **Windows**: CMS Workshp (MolSSI data)\water.xyz
     + **Linux/Mac**: CMS Workshp (MolSSI data)/water.xyz

### Code:

python

CopyEdit

for molecule\_name, file\_path in xyz\_files.items():

#### ****Explanation:****

1. **for molecule\_name, file\_path in xyz\_files.items():**:
   * This is a **loop** that iterates over all items (key-value pairs) in the xyz\_files dictionary.
   * **molecule\_name**: The key of the current item (e.g., "water", "benzene").
   * **file\_path**: The value of the current item (e.g., file path for "water" or "benzene").
   * This allows us to process multiple .xyz files without duplicating code for each molecule.

### Code:

python

CopyEdit

print(F'Printing bonds for {molecule\_name}.')

#### ****Explanation:****

1. **print(F'Printing bonds for {molecule\_name}.')**:
   * Prints a message indicating the molecule currently being processed.
   * **F-string** formatting is used to dynamically insert the molecule's name (molecule\_name) into the string.
   * Example output:
     + For "water": Printing bonds for water.
     + For "benzene": Printing bonds for benzene.

### Code:

python

CopyEdit

symbols, coordinates = open\_xyz(file\_path)

#### ****Explanation:****

1. **symbols, coordinates = open\_xyz(file\_path)**:
   * Calls the open\_xyz function with the path to the .xyz file (file\_path) as an argument.
   * **open\_xyz**:
     + Reads the .xyz file and extracts:
       - symbols: A list of atom types (e.g., ['O', 'H1', 'H2'] for water).
       - coordinates: A NumPy array of atomic coordinates (x, y, z).
   * **symbols** and **coordinates** are then used for bond calculation.

### Code:

python

CopyEdit

print\_bonds(symbols, coordinates)

#### ****Explanation:****

1. **print\_bonds(symbols, coordinates)**:
   * Calls the print\_bonds function with:
     + symbols: The list of atom types.
     + coordinates: The NumPy array of atomic coordinates.
   * **print\_bonds**:
     + Iterates through all pairs of atoms.
     + Calculates the bond length between them using the calculate\_distance function.
     + Checks if the bond length is valid using the bond\_check function.
     + Prints the bond information (e.g., O to H1 : 0.969).

### How the Script Works Overall:

1. The dictionary xyz\_files organizes molecule names and their corresponding .xyz file paths.
2. A **for loop** iterates through each molecule-file pair in xyz\_files.
3. For each molecule:
   * A message is printed (e.g., "Printing bonds for water").
   * The .xyz file is read using open\_xyz, extracting atom symbols and coordinates.
   * The print\_bonds function is called to calculate and print valid bonds for that molecule.

### Example Output for Two Files:

#### ****Input Files:****

1. water.xyz:

mathematica

CopyEdit

3

Water molecule

O 0.000 0.000 0.000

H 0.000 0.759 0.587

H 0.000 0.759 -0.587

1. benzene.xyz:

mathematica

CopyEdit

12

Benzene molecule

C 0.000 1.402 0.000

C 1.214 0.701 0.000

C 1.214 -0.701 0.000

C 0.000 -1.402 0.000

C -1.214 -0.701 0.000

C -1.214 0.701 0.000

H 0.000 2.490 0.000

H 2.156 1.245 0.000

H 2.156 -1.245 0.000

H 0.000 -2.490 0.000

H -2.156 -1.245 0.000

H -2.156 1.245 0.000

#### ****Output:****

mathematica

CopyEdit

Printing bonds for water.

O to H1 : 0.969

O to H2 : 0.969

Printing bonds for benzene.

C to H : 1.088

C to C : 1.403

C to C : 1.403

C to H : 1.088

C to C : 1.403

C to H : 1.088

C to C : 1.403

C to H : 1.088

C to C : 1.403

C to H : 1.088

C to C : 1.403

C to H : 1.088

**Why is my code not generating the same results?**

Let's break this into two parts:

1. **Why your code is not generating the same results**:
   * The code appears to be logically correct, but it may not generate the expected results due to the **input files (water.xyz and benzene.xyz)**.
   * If the .xyz files **do not have the exact atomic positions**, the bond lengths calculated could differ from the expected values.
   * Double-check the contents of water.xyz and benzene.xyz:
     + Ensure that **atom coordinates** in the files are correct.
     + Ensure the .xyz files have no formatting issues (e.g., extra spaces, missing data).

To verify, you can debug the script by printing the symbols and coordinates returned from the open\_xyz function:

python

CopyEdit

symbols, coordinates = open\_xyz(file\_path)

print("Symbols:", symbols)

print("Coordinates:", coordinates)

print\_bonds(symbols, coordinates)

This will confirm whether your files are being read correctly and whether the atomic positions are suitable for bond-length calculation.

1. **Why is the function defined as print\_bonds(atom\_symbols, atom\_coordinates) but called as print\_bonds(symbols, coordinates)?**

This is a common aspect of **function usage** in Python. Here's the explanation:

* + **Function Definition (print\_bonds(atom\_symbols, atom\_coordinates))**:
    - When defining a function, you use **parameters** like atom\_symbols and atom\_coordinates. These are **placeholders** representing data that will be passed to the function when it is called.
    - In this case:
      * atom\_symbols: Placeholder for a list of atom types.
      * atom\_coordinates: Placeholder for a NumPy array of coordinates.
  + **Function Call (print\_bonds(symbols, coordinates))**:
    - When calling the function, you pass **actual data** (called **arguments**) to replace the placeholders.
    - In your main script:
      * symbols (list of atom symbols) is passed as the argument to replace atom\_symbols.
      * coordinates (NumPy array of coordinates) is passed as the argument to replace atom\_coordinates.

So, effectively:

* + **atom\_symbols** inside the print\_bonds function refers to the symbols variable passed from the calling code.
  + **atom\_coordinates** inside the print\_bonds function refers to the coordinates variable passed from the calling code.

This flexibility allows functions to be reused with different datasets by simply passing different arguments.

### Debugging Steps for Your Script:

To ensure your code works as expected:

1. **Verify File Paths**:
   * Check that the file paths (CMS Workshp (MolSSI data)/water.xyz and benzene.xyz) are correct.
   * Ensure the files exist in the specified folder.
2. **Check File Content**:
   * Open the .xyz files and ensure they follow the correct format:

php

CopyEdit

<Number of atoms>

<Comment line>

<Atom type> <X-coordinate> <Y-coordinate> <Z-coordinate>

Example for water.xyz:

mathematica

CopyEdit

3

Water molecule

O 0.000 0.000 0.000

H 0.000 0.759 0.587

H 0.000 0.759 -0.587

1. **Add Debugging Prints**: Add the following to check intermediate steps:

python

CopyEdit

symbols, coordinates = open\_xyz(file\_path)

print("Symbols:", symbols)

print("Coordinates:", coordinates)

1. **Verify Bond-Length Calculation**: Print bond lengths during debugging:

python

CopyEdit

bond\_length = calculate\_distance(atom\_coordinates[num1], atom\_coordinates[num2])

print(F"Distance between {atom\_symbols[num1]} and {atom\_symbols[num2]}: {bond\_length:.3f}")

If after verifying these points you still don't get the expected output, share the content of your .xyz files so I can help identify any potential issues.

**\*\*Add Debugging Prints\*\*: Add the following to check intermediate steps: python CopyEdit `symbols, coordinates = open\_xyz(file\_path) print("Symbols:", symbols) print("Coordinates:", coordinates)`**

**This in printing only benzene molecules and coordinates, even for water**

If the debugging prints are only showing benzene's molecules and coordinates, even for the water file, this indicates that the script is likely **reading the wrong file** for both iterations in the loop. The most likely issues could be:

1. **File paths are incorrect or overlapping:**

Both water and benzene might be pointing to the same file (e.g., benzene.xyz) due to a problem in the xyz\_files dictionary or how file paths are being resolved.

1. **File content mismatch:**

The file that is supposed to contain water molecule data may actually contain benzene data.

### Steps to Diagnose and Fix

#### 1. ****Double-check the**** xyz\_files ****dictionary:****

Ensure that the dictionary has **correct paths** for both files:

python

CopyEdit

xyz\_files = {

"water": os.path.join('CMS Workshp (MolSSI data)', 'water.xyz'),

"benzene": os.path.join('CMS Workshp (MolSSI data)', 'benzene.xyz')

}

* Verify that CMS Workshp (MolSSI data) is the correct folder name.
* Check that both water.xyz and benzene.xyz exist in the folder.

#### 2. ****Print File Paths in the Loop:****

Add a print statement to confirm which file is being accessed for each molecule:

python

CopyEdit

for molecule\_name, file\_path in xyz\_files.items():

print(F'Reading file for {molecule\_name}: {file\_path}')

symbols, coordinates = open\_xyz(file\_path)

print("Symbols:", symbols)

print("Coordinates:", coordinates)

* This will confirm if the correct file path is being passed for each molecule.

#### 3. ****Verify**** .xyz ****File Contents:****

Manually open the water.xyz and benzene.xyz files and ensure their contents match the expected data:

* Example water.xyz:

mathematica

CopyEdit

3

Water molecule

O 0.000 0.000 0.000

H 0.000 0.759 0.587

H 0.000 0.759 -0.587

* Example benzene.xyz:

mathematica

CopyEdit

12

Benzene molecule

C 0.000 1.402 0.000

C 1.214 0.701 0.000

...

#### 4. ****Clear the Python Environment:****

Sometimes, lingering variables in the environment (e.g., symbols and coordinates) from previous runs may interfere. Restart the script to ensure no stale data is being used.

### Example Code with Fixes and Debugging:

Here's the updated code with debugging lines:

python

CopyEdit

import os

import numpy

def open\_xyz(xyz\_file\_path):

"""

Open and read an xyz file. Returns tuple of atom symbols and coordinates.

"""

xyz\_file = numpy.genfromtxt(fname=xyz\_file\_path, skip\_header=2, dtype='unicode')

molecule = xyz\_file[:, 0]

coordinates = xyz\_file[:, 1:].astype(numpy.float64)

return molecule, coordinates

def calculate\_distance(atom1\_coord, atom2\_coord):

"""

Calculate the distance between two three-dimensional points.

"""

x\_distance = atom1\_coord[0] - atom2\_coord[0]

y\_distance = atom1\_coord[1] - atom2\_coord[1]

z\_distance = atom1\_coord[2] - atom2\_coord[2]

bond\_length = numpy.sqrt(x\_distance\*\*2 + y\_distance\*\*2 + z\_distance\*\*2)

return bond\_length

def bond\_check(bond\_length, minimum\_length=0, maximum\_length=1.5):

"""

Check if a distance is a bond based on a minimum and maximum bond length.

"""

return minimum\_length < bond\_length <= maximum\_length

def print\_bonds(atom\_symbols, atom\_coordinates):

"""

Prints atom symbols and bond lengths for a set of atoms.

"""

num\_atoms = len(atom\_symbols)

for num1 in range(0, num\_atoms):

for num2 in range(0, num\_atoms):

if num1 < num2: # Avoid duplicates and self-comparisons

bond\_length = calculate\_distance(atom\_coordinates[num1], atom\_coordinates[num2])

if bond\_check(bond\_length): # Check if the bond is valid

print(F'{atom\_symbols[num1]} to {atom\_symbols[num2]} : {bond\_length:.3f}')

# Main script to handle multiple files

xyz\_files = {

"water": os.path.join('CMS Workshp (MolSSI data)', 'water.xyz'),

"benzene": os.path.join('CMS Workshp (MolSSI data)', 'benzene.xyz')

}

for molecule\_name, file\_path in xyz\_files.items():

print(F'Reading file for {molecule\_name}: {file\_path}')

symbols, coordinates = open\_xyz(file\_path)

print(F'Symbols for {molecule\_name}: {symbols}')

print(F'Coordinates for {molecule\_name}: {coordinates}')

print(F'Printing bonds for {molecule\_name}.')

print\_bonds(symbols, coordinates)

### Expected Debug Output:

If all is working correctly, the output should look something like this:

plaintext

CopyEdit

Reading file for water: CMS Workshp (MolSSI data)/water.xyz

Symbols for water: ['O' 'H' 'H']

Coordinates for water: [[0. 0. 0. ]

[0. 0.759 0.587]

[0. 0.759 -0.587]]

Printing bonds for water.

O to H : 0.969

O to H : 0.969

Reading file for benzene: CMS Workshp (MolSSI data)/benzene.xyz

Symbols for benzene: ['C' 'C' 'H' ...]

Coordinates for benzene: [[...]]

Printing bonds for benzene.

C to H : ...

...

If the debugging prints do not show the correct file content for water and benzene, the issue is in the xyz\_files paths or the file contents. Let me know what you observe!

**Both the file paths and the file content is correct, but still i am not getting the expected output.**

If the file paths and contents are correct, but you're still not getting the expected output, we should carefully re-check the logic in the **main script**, especially how files are read and processed.

Here's a **step-by-step debugging and adjustment process** to pinpoint and fix the issue:

### Possible Causes

1. **File Handling Issue**

If the files are not being properly read, ensure that open\_xyz is correctly returning the symbols and coordinates for each file.

1. **Bond Check Logic Issue**

If bonds aren't being detected correctly, ensure that:

* + The bond lengths calculated are reasonable.
  + The bond\_check function is correctly returning True or False.

1. **Variable Overwriting**  
   If the symbols and coordinates are being overwritten between iterations, ensure the loop variables are isolated properly.
2. **Logic Errors in the print\_bonds Function**  
   If bond lengths aren't being printed, verify that the loops and conditions are correct.

### Debugging Steps

#### Step 1: Add Detailed Debugging in the Loop

Insert debug print statements to trace the flow of the program:

python

CopyEdit

for molecule\_name, file\_path in xyz\_files.items():

print(F'\nReading file for {molecule\_name}: {file\_path}')

# Open the file

symbols, coordinates = open\_xyz(file\_path)

# Debug: Verify file data

print(F'\nSymbols for {molecule\_name}: {symbols}')

print(F'Coordinates for {molecule\_name}: \n{coordinates}')

print(F'\nPrinting bonds for {molecule\_name}.')

# Print bonds

print\_bonds(symbols, coordinates)

#### Step 2: Verify the open\_xyz Function

Manually check that symbols and coordinates are returned correctly by the open\_xyz function for each file:

python

CopyEdit

symbols, coordinates = open\_xyz('CMS Workshp (MolSSI data)/water.xyz')

print("Water symbols:", symbols)

print("Water coordinates:", coordinates)

symbols, coordinates = open\_xyz('CMS Workshp (MolSSI data)/benzene.xyz')

print("Benzene symbols:", symbols)

print("Benzene coordinates:", coordinates)

The output should clearly show the data from each file.

#### Step 3: Test the bond\_check and print\_bonds Functions

Ensure that the bond lengths are calculated and printed correctly. You can do this by calling print\_bonds directly with known values:

python

CopyEdit

water\_symbols = ['O', 'H', 'H']

water\_coordinates = numpy.array([[0.000, 0.000, 0.000],

[0.000, 0.759, 0.587],

[0.000, 0.759, -0.587]])

print("Testing bonds for water:")

print\_bonds(water\_symbols, water\_coordinates)

benzene\_symbols = ['C', 'C', 'H', 'H', 'C', 'H', 'C', 'H', 'C', 'H', 'C', 'H']

benzene\_coordinates = numpy.array([[0.000, 1.402, 0.000], [1.214, 0.701, 0.000], ...]) # Fill in full coordinates

print("Testing bonds for benzene:")

print\_bonds(benzene\_symbols, benzene\_coordinates)

#### Step 4: Check the print\_bonds Logic

Ensure that:

1. num1 < num2 condition avoids duplicate and self-comparisons.
2. Bond lengths fall within the range defined by bond\_check.

Add debug prints inside print\_bonds to verify:

python

CopyEdit

def print\_bonds(atom\_symbols, atom\_coordinates):

num\_atoms = len(atom\_symbols)

for num1 in range(0, num\_atoms):

for num2 in range(0, num\_atoms):

if num1 < num2: # Avoid duplicates and self-comparisons

bond\_length = calculate\_distance(atom\_coordinates[num1], atom\_coordinates[num2])

print(F"Checking bond between {atom\_symbols[num1]} and {atom\_symbols[num2]}: Length = {bond\_length:.3f}")

if bond\_check(bond\_length): # Check if the bond is valid

print(F'{atom\_symbols[num1]} to {atom\_symbols[num2]} : {bond\_length:.3f}')

### Updated Main Script with Debugging

Here is the complete script after incorporating debugging:

python

CopyEdit

import os

import numpy

def open\_xyz(xyz\_file\_path):

"""

Open and read an xyz file. Returns tuple of atom symbols and coordinates.

"""

xyz\_file = numpy.genfromtxt(fname=xyz\_file\_path, skip\_header=2, dtype='unicode')

molecule = xyz\_file[:, 0]

coordinates = xyz\_file[:, 1:].astype(numpy.float64)

return molecule, coordinates

def calculate\_distance(atom1\_coord, atom2\_coord):

"""

Calculate the distance between two three-dimensional points.

"""

x\_distance = atom1\_coord[0] - atom2\_coord[0]

y\_distance = atom1\_coord[1] - atom2\_coord[1]

z\_distance = atom1\_coord[2] - atom2\_coord[2]

bond\_length = numpy.sqrt(x\_distance\*\*2 + y\_distance\*\*2 + z\_distance\*\*2)

return bond\_length

def bond\_check(bond\_length, minimum\_length=0, maximum\_length=1.5):

"""

Check if a distance is a bond based on a minimum and maximum bond length.

"""

return minimum\_length < bond\_length <= maximum\_length

def print\_bonds(atom\_symbols, atom\_coordinates):

"""

Prints atom symbols and bond lengths for a set of atoms.

"""

num\_atoms = len(atom\_symbols)

for num1 in range(0, num\_atoms):

for num2 in range(0, num\_atoms):

if num1 < num2: # Avoid duplicates and self-comparisons

bond\_length = calculate\_distance(atom\_coordinates[num1], atom\_coordinates[num2])

print(F"Checking bond between {atom\_symbols[num1]} and {atom\_symbols[num2]}: Length = {bond\_length:.3f}")

if bond\_check(bond\_length): # Check if the bond is valid

print(F'{atom\_symbols[num1]} to {atom\_symbols[num2]} : {bond\_length:.3f}')

# Main script to handle multiple files

xyz\_files = {

"water": os.path.join('CMS Workshp (MolSSI data)', 'water.xyz'),

"benzene": os.path.join('CMS Workshp (MolSSI data)', 'benzene.xyz')

}

for molecule\_name, file\_path in xyz\_files.items():

print(F'\nReading file for {molecule\_name}: {file\_path}')

symbols, coordinates = open\_xyz(file\_path)

print(F'Symbols for {molecule\_name}: {symbols}')

print(F'Coordinates for {molecule\_name}: \n{coordinates}')

print(F'\nPrinting bonds for {molecule\_name}.')

print\_bonds(symbols, coordinates)

### Let Me Know What Happens

Run this updated script and observe the output. If the issue persists, share the debug output, and we'll further analyze it!

4o

##### You said:

**In this code, We will be usiing the same code that we developed in the last module. This is my input. There is an issue in your main script that is causing the program to fail. Specifically, the error lies in how the variables water\_molecule and coordinates are used in the loop. Here's the explanation and fix:**

### Issue 1: Variable Name Mismatch (molecule vs water\_molecule)

In your open\_xyz function, the variable for storing molecule symbols is named molecules.  
In the main script, you assign the output of open\_xyz(water\_file) to water\_molecule and coordinates. However, inside your loop, you're referring to a variable named molecule (which doesn't exist in your code).

This results in an error when the program tries to access molecule[i\_1] and molecule[i\_2].

### Fix for Issue 1

Replace all instances of molecule in the loop with water\_molecule (since that's what holds the atom symbols).

### Final, Corrected Code

python

CopyEdit

import os

import numpy

""" Function to get/open the file """

def open\_xyz(xyz\_file\_path): # "xyz\_file\_path" is just a placeholder to take the needed xyz file path

xyz\_file = numpy.genfromtxt(fname=xyz\_file\_path, skip\_header=2, dtype='unicode')

molecules = xyz\_file[:, 0]

coordinates = xyz\_file[:, 1:]

coordinates = coordinates.astype(numpy.float64)

return molecules, coordinates

""" Function to calculate the bond\_len for any\_xyz\_file """

def atomic\_dist(atom1\_coord, atom2\_coord):

x\_distances = atom1\_coord[0] - atom2\_coord[0]

y\_distances = atom1\_coord[1] - atom2\_coord[1]

z\_distances = atom1\_coord[2] - atom2\_coord[2]

bond\_len = numpy.sqrt(x\_distances\*\*2 + y\_distances\*\*2 + z\_distances\*\*2)

return bond\_len

""" Function to check if a distance qualifies as a bond """

def bond\_check(bond\_len, min\_length=0, max\_length=1.5):

if bond\_len > 0 and bond\_len <= 1.5:

return True

else:

return False

""" Main script to calculate and print bond lengths """

# Inputs:

water\_file = os.path.join('CMS Workshp (MolSSI data)', 'water.xyz') # This is my input for "xyz\_file"

water\_molecule, coordinates = open\_xyz(water\_file) # This assignment makes the "water\_file" variables available in the main script.

# Main Script:

row\_num = len(coordinates[:, 0])

for i\_1 in range(0, row\_num):

for i\_2 in range(0, row\_num):

if i\_1 < i\_2: # Avoid duplicates and self-comparisons

bond\_len = atomic\_dist(coordinates[i\_1], coordinates[i\_2])

if bond\_check(bond\_len): # Check if the bond length is valid

print(F'{water\_molecule[i\_1]} to {water\_molecule[i\_2]}: {bond\_len:.3f}')

### Explanation of the Fix

1. **Variable Renaming**:
   * molecule is replaced with water\_molecule because the output of open\_xyz is assigned to water\_molecule and coordinates.
2. **General Improvements**:
   * The loop logic (i\_1 < i\_2) ensures no self-comparisons or duplicate bond checks.
   * Added :.3f formatting to ensure bond lengths are printed with three decimal places.

### Example Output

For a water molecule (water.xyz file):

plaintext

CopyEdit

O to H : 0.969

O to H : 0.969

H to H : 1.527

If this still doesn't work as expected, ensure that:

* The file path 'CMS Workshp (MolSSI data)/water.xyz' is correct.
* The water.xyz file is correctly formatted. It should have the following structure:

mathematica

CopyEdit

3

Water molecule

O 0.000 0.000 0.000

H 0.000 0.759 0.587

H 0.000 0.759 -0.587

**My question is why I am not getting an error? despite of this mismatch, my code is working correctly, why?? also, while defining the function, I used "molecules" to get O H1 H2 from the file. But while defining the inputs for my code, I used "water\_molecule" to run to code. How does python know that it should pick O H1 H2 data for "water\_molecule"??**

This is an important question about how Python variables and assignments work. Let me explain why your code works without throwing an error, even though there’s an apparent mismatch between molecule and water\_molecule.

### 1. ****Why is there no error despite the mismatch?****

In the context of your code, **if it works correctly** without an error, then somewhere in your script or during testing, a variable named molecule **exists in the global scope** or has been assigned earlier in the session. Python will silently use the existing molecule variable if it's already defined.

For example, if at some point you did something like this earlier:

python

CopyEdit

molecule = water\_molecule

or accidentally assigned a value to molecule, Python will use it.

If molecule isn't defined at all in the session, **you should see an error** like:

plaintext

CopyEdit

NameError: name 'molecule' is not defined

### 2. ****How does Python know to associate**** water\_molecule ****with the atom symbols?****

This happens because when you call open\_xyz(water\_file), the function explicitly returns two values:

python

CopyEdit

return molecules, coordinates

You assign those two return values to the variables water\_molecule and coordinates in the main script:

python

CopyEdit

water\_molecule, coordinates = open\_xyz(water\_file)

From Python’s perspective:

* molecules (inside the function) contains the atom symbols (O, H1, H2, etc.).
* When you call the function, the returned molecules gets assigned to the variable water\_molecule. So, **water\_molecule now holds the atom symbols.**

Python doesn't "know" that water\_molecule should refer to the atom symbols. It works because **you explicitly made this connection** when assigning the return values of the function to water\_molecule and coordinates.

### 3. ****How Python Handles Variable Names****

* Variable names in Python are just **labels** that point to data in memory. You can call the same data by different names in different parts of your code. For example:

python

CopyEdit

x = 42

y = x # Now, y is pointing to the same data as x

print(y) # Output: 42

In your case:

* + Inside open\_xyz, the atom symbols are called molecules.
  + In your main script, the atom symbols are assigned to water\_molecule.
  + As long as you pass the correct variables to the loop, Python handles it seamlessly.

### 4. ****Why Does the Code Work Despite the**** molecule ****Mismatch?****

If your code works with the molecule variable in the loop, one of the following scenarios is true:

1. **molecule is defined elsewhere** in your script or session. If this variable contains the atom symbols (O, H1, H2), the code works correctly but relies on the prior definition of molecule.
2. **You accidentally reused or assigned a variable.** For example:

python

CopyEdit

molecule = water\_molecule

This can happen if you were experimenting interactively in a Python environment (e.g., Jupyter Notebook or IPython).

1. **If molecule is truly undefined, you would get an error.** If you're not seeing an error, it means molecule was defined somewhere. You can confirm this by adding:

python

CopyEdit

print(molecule)

before your loop.

### 5. ****Best Practice****

To avoid such confusion, always use consistent variable names throughout your code. For example:

* Use molecules consistently if that's what the data represents.
* If you assign it to another variable (like water\_molecule), ensure you use that name consistently in the rest of your code.

Corrected loop:

python

CopyEdit

for i\_1 in range(0, row\_num):

for i\_2 in range(0, row\_num):

if i\_1 < i\_2:

bond\_len = atomic\_dist(coordinates[i\_1], coordinates[i\_2])

if bond\_check(bond\_len):

print(F'{water\_molecule[i\_1]} to {water\_molecule[i\_2]}: {bond\_len:.3f}')

### Summary

1. Your code works because molecule is defined somewhere in the session or script (either explicitly or accidentally).
2. Python doesn’t automatically associate water\_molecule with atom symbols. It works because **you assigned the output of open\_xyz to water\_molecule.**
3. To avoid these issues, use consistent and clear variable names throughout your script.

**Okay, if this is the case, that why python does not assign O H1 H2 to coordinates. I returned both the molecules, coordinates. Then why does it assigns "molecules" data to "water\_molecules" but not "coordinates"**

This is an excellent observation, and the reason Python does not assign O, H1, H2 (the atom symbols) to coordinates is due to how **tuple unpacking** works in Python when returning and assigning values. Let me break it down:

### 1. ****How**** return ****Works in**** open\_xyz

The function open\_xyz is written to return **two values** in a tuple:

python

CopyEdit

return molecules, coordinates

* molecules: Contains atom symbols like O, H1, H2.
* coordinates: Contains the corresponding numerical coordinate arrays.

### 2. ****How the Returned Tuple is Assigned****

When you call open\_xyz(water\_file) and assign its result to water\_molecule and coordinates, Python does **tuple unpacking** like this:

python

CopyEdit

water\_molecule, coordinates = open\_xyz(water\_file)

This line does the following:

1. Takes the first returned value (molecules) and assigns it to water\_molecule.
2. Takes the second returned value (coordinates) and assigns it to coordinates.

### 3. ****Why Python Assigns Atom Symbols to**** water\_molecule

When you write water\_molecule, coordinates = open\_xyz(water\_file), **the order of variables on the left-hand side matches the order of returned values from the function**:

python

CopyEdit

molecules (atom symbols) → water\_molecule

coordinates (numerical coordinates) → coordinates

The reason Python assigns the first returned value (molecules) to water\_molecule and not to coordinates is because **the order matters**. Python unpacks the tuple sequentially:

* First value → first variable
* Second value → second variable
* And so on.

If you wrote:

python

CopyEdit

coordinates, water\_molecule = open\_xyz(water\_file)

Now coordinates would receive the atom symbols, and water\_molecule would receive the numerical coordinates. **The order on the left-hand side determines what gets assigned.**

### 4. ****Why**** coordinates ****Always Gets the Numerical Data****

The second value returned by open\_xyz is the array of numerical coordinates:

python

CopyEdit

return molecules, coordinates

Since the second variable in your assignment (coordinates) corresponds to the second value returned by the function, it always gets the numerical data.

### 5. ****Python Doesn't "Know" What Each Variable Represents****

Python doesn’t inherently understand what water\_molecule or coordinates represent. It only assigns values based on the position in the return statement. This means:

* You, the programmer, determine what names you use for the variables that receive the returned values.
* Python simply maps values in the order they appear in the return tuple.

### 6. ****Example to Clarify****

Here’s an analogy:

Suppose your function open\_xyz returns a tuple like this:

python

CopyEdit

('apples', 'oranges')

If you write:

python

CopyEdit

fruit1, fruit2 = open\_xyz()

fruit1 gets 'apples' and fruit2 gets 'oranges'.

But if you write:

python

CopyEdit

fruit2, fruit1 = open\_xyz()

Now fruit2 gets 'apples' and fruit1 gets 'oranges'.

### 7. ****Conclusion****

The key is **order and tuple unpacking**:

* return molecules, coordinates sends the atom symbols first and the numerical data second.
* water\_molecule, coordinates = open\_xyz(water\_file) matches the order of returned values:
  + First value (molecules) → water\_molecule
  + Second value (coordinates) → coordinates

This behavior is entirely dependent on **how you write your assignment statement** and has nothing to do with Python "understanding" what these variables mean.

# Running code from the Linux Command Line

**CHatGTP generated:**

import os

""" Function to extract Etot values from a user-defined mdout file """

def extract\_etot(mdout\_file\_path):

"""

Reads an mdout file, extracts total energy (Etot) values,

and writes them to a file named 'Etot.txt'.

"""

# Define the output file

output\_file\_path = "Etot.txt"

# Open the mdout file for reading

with open(mdout\_file\_path, 'r') as mdout\_file:

data = mdout\_file.readlines() # Read all lines from the mdout file

# Open the output file for writing

with open(output\_file\_path, 'w+') as etot\_txt\_file:

for line in data: # Iterate through each line of the file

if "Etot" in line: # Check if the line contains the string "Etot"

split\_line = line.split() # Split the line into parts

etot\_value = split\_line[2] # Extract the third element (Etot value)

print(f"Etot found: {etot\_value}") # Print the Etot value

etot\_txt\_file.write(f'The total energy value is {etot\_value} \n') # Write to the file

**This is what I wrote to get the Etot from mdout file and create a txt file with the Etot data. What is wrong with this. See “analyze\_mdout.ipynb” file for the code.**

Your code has a few issues, including syntax errors, missing functionality, and logic inconsistencies. Here’s an explanation of what’s wrong and how to fix it.

### Issues in the Code

1. **open\_mdout function is empty**:
   * The open\_mdout function is defined but does nothing. This should either process the .mdout file or be removed if unnecessary.
2. **Incorrect argparse method**:
   * parser.addargument is a typo. The correct method is parser.add\_argument.
3. **Logic for writing the .txt file**:
   * You are opening the same file (mdout\_file) twice: once for reading and once for writing ('w+'). This will overwrite the file and erase its contents before reading. You need to specify **separate files** for input and output.
4. **Indentation Errors**:
   * Python is indentation-sensitive. The block for writing the .txt file is not indented correctly.
5. **Unnecessary Etot\_txt\_file.close()**:
   * Since you're using with open(), the file will automatically close when the block ends. You don’t need to call .close() manually.
6. **No output file path**:
   * You are not dynamically naming the output file based on the input file. You need to construct a meaningful file name for the .txt file.
7. **No handling for edge cases**:
   * Your code assumes that "Etot" will always appear in the file, but this may not be the case. You should include error handling for missing data.

### Corrected Code

Here’s the corrected and improved version:

python

CopyEdit

import os

import argparse

def extract\_etot(mdout\_file\_path):

"""

Reads an mdout file, extracts total energy values (Etot), and writes them to a .txt file.

"""

# Determine output file name based on the input file name

base\_name = os.path.basename(mdout\_file\_path).split('.')[0] # Get the base file name without extension

output\_file\_path = f"{base\_name}\_Etot.txt"

# Open the mdout file for reading

with open(mdout\_file\_path, 'r') as infile:

data = infile.readlines()

# Open the output file for writing

with open(output\_file\_path, 'w') as outfile:

for line in data:

if "Etot" in line:

# Extract the total energy value from the line

split\_line = line.split()

etot\_value = split\_line[2]

print(f"Found Etot: {etot\_value}")

outfile.write(f"{etot\_value}\n") # Write the value to the output file

print(f"Etot values written to {output\_file\_path}")

if \_\_name\_\_ == "\_\_main\_\_":

# Initialize argparse

parser = argparse.ArgumentParser(description="This script analyzes an mdout file and extracts Etot values.")

# Generate argument for the mdout file

parser.add\_argument("mdout\_file", help="The file path to the mdout file to analyze.")

args = parser.parse\_args()

# Call the function to extract Etot

extract\_etot(args.mdout\_file)

### Key Fixes and Improvements

1. **Added the extract\_etot function**:
   * Encapsulates the logic for processing the .mdout file and writing Etot values.
2. **Fixed add\_argument typo**:
   * Correctly uses parser.add\_argument to capture the command-line input.
3. **Dynamic output file naming**:
   * Creates an output file name (<base\_name>\_Etot.txt) based on the input file name.
4. **Separate file handling**:
   * The input .mdout file is opened for **reading** ('r'), and the output .txt file is opened for **writing** ('w').
5. **Automatic file closing**:
   * Uses with open() for both files, ensuring they are properly closed after processing.
6. **Print statements for clarity**:
   * Prints extracted values and confirms where the output is written.
7. **Error handling**:
   * The script won’t crash if the input file is missing Etot. However, you can add additional error handling if needed.

### Usage

Save the script as analyze\_mdout.py, and run it from the command line like this:

bash

CopyEdit

python analyze\_mdout.py "path/to/03\_Prod.mdout"

This will create an output file named 03\_Prod\_Etot.txt in the same directory, containing the extracted total energy values.

### Example Input and Output

#### Input: 03\_Prod.mdout (Sample Content)

css

CopyEdit

NSTEP = 1 TIME(PS) = 0.000 Etot = -12345.678

NSTEP = 2 TIME(PS) = 0.002 Etot = -12340.000

NSTEP = 3 TIME(PS) = 0.004 Etot = -12350.321

#### Output: 03\_Prod\_Etot.txt

diff

CopyEdit

-12345.678

-12340.000

-12350.321

**In the above code, you are opening a file, copying all of its content to a txt file and then applying "if statement". What I want to apply "if statement" on original file and then print all that data to a txt file as we used to do before**

###### ChatGPT said:

ChatGPT

Got it! You want the logic changed so the **if-statement filtering** is applied directly on the input file as it’s read line-by-line, without first copying the entire file content into memory with readlines(). This ensures the filtering and writing happen simultaneously, making the code more memory-efficient and closely resembling the way it was "done before."

Here’s the updated code to meet that requirement:

### Code: Filtering While Reading Line by Line

python

CopyEdit

import os

import argparse

def extract\_etot(mdout\_file\_path):

"""

Reads an mdout file line-by-line, extracts total energy values (Etot), and writes them to a .txt file.

"""

# Determine output file name based on the input file name

base\_name = os.path.basename(mdout\_file\_path).split('.')[0] # Get the base file name without extension

output\_file\_path = f"{base\_name}\_Etot.txt"

# Open the mdout file for reading and the output file for writing

with open(mdout\_file\_path, 'r') as infile, open(output\_file\_path, 'w') as outfile:

for line in infile:

# Check if the line contains "Etot"

if "Etot" in line:

# Extract the total energy value from the line

split\_line = line.split()

etot\_value = split\_line[2]

print(f"Found Etot: {etot\_value}")

outfile.write(f"{etot\_value}\n") # Write the value to the output file

print(f"Etot values written to {output\_file\_path}")

if \_\_name\_\_ == "\_\_main\_\_":

# Initialize argparse

parser = argparse.ArgumentParser(description="This script analyzes an mdout file and extracts Etot values.")

# Generate argument for the mdout file

parser.add\_argument("mdout\_file", help="The file path to the mdout file to analyze.")

args = parser.parse\_args()

# Call the function to extract Etot

extract\_etot(args.mdout\_file)

### Explanation of Changes

1. **Reading and Filtering Simultaneously**:
   * Instead of using data = infile.readlines(), the code now processes the file **line by line** directly using a for loop (for line in infile).
2. **Immediate Filtering**:
   * The if "Etot" in line check is applied as each line is read, ensuring only relevant lines are processed further.
3. **Writing While Reading**:
   * The filtered data (etot\_value) is directly written to the output file inside the same loop. This avoids holding unnecessary data in memory.
4. **Efficiency**:
   * This approach is more memory-efficient and aligns with how files are processed sequentially when working with large datasets.

### Advantages of This Approach

* **Memory-Efficient**: No need to load the entire file into memory.
* **Streamlined Process**: Filtering and writing happen in a single pass.
* **Simple and Clean**: The code is closer to the "write as you filter" approach you described.

### Usage Example

#### Input File: 03\_Prod.mdout

css

CopyEdit

NSTEP = 1 TIME(PS) = 0.000 Etot = -12345.678

NSTEP = 2 TIME(PS) = 0.002 Etot = -12340.000

NSTEP = 3 TIME(PS) = 0.004 Etot = -12350.321

#### Command to Run:

bash

CopyEdit

python analyze\_mdout.py "path/to/03\_Prod.mdout"

#### Output File: 03\_Prod\_Etot.txt

diff

CopyEdit

-12345.678

-12340.000

-12350.321

# Explanation of “analyze\_all\_mdout” file using parsing command

**### Running code from the Linux Command Line ###**

# For this homework assignment, we will use the code from "Processing Multiple Files and Writing Files"

# where we processed the file "03\_Prod.mdout". We will create another Jupyter "analyze\_mdout.ipynb".

# Refer to this file for further details about the assignment.

# We will modify this using argparse to run in terminal.

## My Code ##

# Let us do an assignment now to find the total energy from any user defined \*.mdout file (wild card).

import os

import glob # Import glob to handle wildcard patterns

import argparse

def open\_mdout(mdout\_file\_path): # "mdout\_file\_path" is just a placeholder to take the needed xyz file path

# Create a .txt file to store Etot data (to have a variable file name, we use basename function):

filename = os.path.basename(mdout\_file\_path)

split\_filename = filename.split('.')

mdout\_filename = split\_filename[0]

with open (F'\*{mdout\_filename}\_Etot\_file.txt' , 'w+') as Etot\_txt\_file: # This line create a txt file

# open the mdout file to get Etot data:

with open (mdout\_file\_path , 'r') as outfile: # "mdout\_file\_path" is user defined file path in terminal

data = outfile.readlines()

for line in data:

if "Etot" in line:

Etot\_line = line

#print(Etot\_line)

split\_Etot\_line = Etot\_line.split()

Etot\_value = float(split\_Etot\_line[2])

print(Etot\_value)

Etot\_txt\_file.write(F'The total energy value is {Etot\_value} \n')

if \_\_name\_\_ == "\_\_main\_\_":

# Initialize argparse

parser = argparse.ArgumentParser(description = "The script analyzes a user given mdout file and outputs Etot.")

# Generate argument for mdout file (Replace argparse setup with the following to allow multiple files)

parser.add\_argument("all\_mdout\_files", nargs="+", help="The file paths to multiple mdout files")

args = parser.parse\_args()

all\_mdout\_files = args.all\_mdout\_files

print(all\_mdout\_files) # all\_mdout\_files is a list containing multiple mdout files

# To call a function, I need input

for mdout\_file in all\_mdout\_files:

open\_mdout(mdout\_file)

print(mdout\_file)

## To use this code on terminal ##

# Convert ipynb file to py using: jupyter nbconvert --to script analyze\_all\_mdout.ipynb

# Use this command to execute: python analyze\_all\_mdout.py "CMS Workshp (MolSSI data)"/\*.mdout

# Use this command to enter the directory: cd /Users/areejnad/Documents/Areej\ Academic\ Data/My\ Research/CCR\ and\ Simulation\ Learning/Learning\ Python/Learning\ Data/

# Use this command to check the files: ls "CMS Workshp (MolSSI data)"/\*.mdout (notice the quotes location here)

# Or use this command to execute: python analyze\_all\_mdout.py CMS\ Workshp\ \(MolSSI\ data\)/\*.mdout

### Code Before if \_\_name\_\_ == "\_\_main\_\_":

This part defines the logic for reading and processing \*.mdout files to extract Etot values and save them to a .txt file.

* **Imports:**

python

CopyEdit

import os

import glob

import argparse

* + **os**: Used to interact with the operating system, such as managing file paths.
  + **glob**: Used for handling wildcard patterns (e.g., \*.mdout).
  + **argparse**: Used to handle command-line arguments provided by the user.
* **Function open\_mdout:** This function processes an individual mdout file to extract Etot values and writes them to a corresponding .txt file.

python

CopyEdit

def open\_mdout(mdout\_file\_path):

* + Takes the path to an mdout file as input.

python

CopyEdit

filename = os.path.basename(mdout\_file\_path)

split\_filename = filename.split('.')

mdout\_filename = split\_filename[0]

* + Extracts the base filename (e.g., from /path/to/file.mdout, extracts file.mdout) and then splits it into file and mdout using . as the delimiter.

python

CopyEdit

with open (F'\*{mdout\_filename}\_Etot\_file.txt' , 'w+') as Etot\_txt\_file:

* + Creates or opens a text file named \*\_Etot\_file.txt to store the extracted Etot values.

python

CopyEdit

with open (mdout\_file\_path , 'r') as outfile:

data = outfile.readlines()

* + Opens the input mdout file and reads all its lines into a list called data.

python

CopyEdit

for line in data:

if "Etot" in line:

Etot\_line = line

split\_Etot\_line = Etot\_line.split()

Etot\_value = float(split\_Etot\_line[2])

print(Etot\_value)

Etot\_txt\_file.write(F'The total energy value is {Etot\_value} \n')

* + Loops through each line in the data list.
  + If the line contains "Etot", it:
    1. Splits the line into components.
    2. Extracts the third element (at index 2), converts it to a float (this is the Etot value).
    3. Prints the Etot value.
    4. Writes the value to the output .txt file.

### Code After if \_\_name\_\_ == "\_\_main\_\_":

This part sets up the script to run directly from the command line and handles multiple mdout files.

#### Line-by-Line Breakdown

1. **if \_\_name\_\_ == "\_\_main\_\_":**
   * This condition ensures that the code inside it only runs when the script is executed directly (not when it is imported as a module in another script).
2. **parser = argparse.ArgumentParser(...)**

python

CopyEdit

parser = argparse.ArgumentParser(description="The script analyzes a user given mdout file and outputs Etot.")

* + Creates an ArgumentParser object, which allows you to define and parse command-line arguments.
  + The description parameter provides a description of the script's purpose.

1. **parser.add\_argument(...)**

python

CopyEdit

parser.add\_argument("all\_mdout\_files", nargs="+", help="The file paths to multiple mdout files")

* + Adds a positional argument all\_mdout\_files to the parser:
    - "nargs=+": Allows the user to provide one or more file paths (e.g., file1.mdout file2.mdout).
    - "help": A short description that is displayed if the user requests help using the -h flag.

1. **args = parser.parse\_args()**

python

CopyEdit

args = parser.parse\_args()

* + Parses the command-line arguments and stores them in the args object.

1. **all\_mdout\_files = args.all\_mdout\_files**

python

CopyEdit

all\_mdout\_files = args.all\_mdout\_files

* + Extracts the list of mdout file paths provided by the user from the args object.

1. **print(all\_mdout\_files)**

python

CopyEdit

print(all\_mdout\_files)

* + Prints the list of mdout file paths for debugging or confirmation purposes.

1. **Loop through each file:**

python

CopyEdit

for mdout\_file in all\_mdout\_files:

open\_mdout(mdout\_file)

print(mdout\_file)

* + Iterates over the list of file paths (all\_mdout\_files).
  + For each file:
    - Calls the open\_mdout function, passing the current file path as an argument. This processes the file and extracts Etot values.
    - Prints the name of the processed file.

### How the Script Works in Practice

1. **Run the Script:**

bash

CopyEdit

python script\_name.py file1.mdout file2.mdout

* + Replace script\_name.py with the name of your Python file.
  + Provide one or more \*.mdout file paths as arguments.

1. **What Happens:**
   * The argparse module collects the file paths into all\_mdout\_files.
   * For each file in the list:
     + The open\_mdout function processes it, extracts Etot values, and writes them to a corresponding text file.
     + The script prints the Etot value and the processed file name.

### Example Command

bash

CopyEdit

python analyze\_mdout.py file1.mdout file2.mdout

### Example Output

csharp

CopyEdit

['file1.mdout', 'file2.mdout']

-2123.34

The total energy value is -2123.34

file1.mdout

-2341.56

The total energy value is -2341.56

file2.mdout

This output confirms that both files were processed successfully, and their Etot values were printed and saved to the respective text files.

# Git and GitHub

### ****What is Git?****

* **Git** is a version control system.  
  It helps you track changes in your files (especially code) and manage your project's history.  
  Key features of Git:
  + **Version tracking:** Records changes made to files over time.
  + **Branching and merging:** Lets you create different branches for features or fixes and merge them back.
  + **Collaboration:** Allows multiple developers to work on the same project without overwriting each other's work.

Example:  
Imagine you're writing a novel. Git lets you save snapshots (versions) of your draft so you can go back to an earlier version if needed.

### ****What is GitHub?****

* **GitHub** is a cloud-based platform for hosting Git repositories.  
  Think of it as a place where you can store your Git projects online so others can access and collaborate on them.

Key features of GitHub:

* **Repository hosting:** Stores your projects in the cloud.
* **Collaboration tools:** Provides features like pull requests, issue tracking, and team discussions.
* **Portfolio:** Showcases your code/projects to potential employers or collaborators.

Example:  
You can use Git on your computer to track changes locally and then upload (push) those changes to GitHub to share them with others.

### ****What is Atom?****

* **Atom** is a free, open-source text editor developed by GitHub.  
  It's designed for coding and is highly customizable with themes and extensions.

Key features of Atom:

* **Git and GitHub integration:** You can view, edit, and commit changes directly from Atom.
* **Cross-platform:** Works on Windows, macOS, and Linux.
* **Customization:** Add plugins or write your own to tailor it to your needs.

Example:  
If you're editing Python scripts or writing Markdown files, Atom provides syntax highlighting and easy integration with Git.

### ****The Command:****

bash

CopyEdit

git config --global core.editor "atom --wait"

**What it does:**

* This sets Atom as the default text editor for Git.
* When Git needs you to edit something (e.g., writing a commit message), it will open Atom.

**What --wait means:**

* Git will "wait" until you close the Atom window before continuing.  
  This ensures that Git doesn't proceed until you're done editing.

### ****Example Workflow with Git, GitHub, and Atom:****

1. **Write code in Atom:**  
   You create or edit files for your project using Atom.
2. **Track changes with Git:**  
   You use Git to stage and commit your changes.
3. **Push changes to GitHub:**  
   You upload your Git history to GitHub, where it’s stored and shared.

### ****Step 1: Navigate to Your Project Folder****

To start tracking files in a Git repository, you need to navigate to the location where your project files are stored. For your specific case, run:

bash

CopyEdit

cd "Documents/Areej Academic Data/My Research/CCR and Simulation Learning/Learning Python/Learning Data/"

This command uses the cd (change directory) command to move into your project folder.

### ****Step 2: Check if the Folder is a Git Repository****

You can check whether the folder is already a Git repository by running:

bash

CopyEdit

git status

* **Output if it’s not a Git repository:**

arduino

CopyEdit

fatal: not a git repository (or any of the parent directories): .git

* This means you need to initialize a new repository in this folder.

### ****Step 3: Initialize a Git Repository****

To create a new Git repository in this location, run:

bash

CopyEdit

git init

* **What happens:**  
  This command creates a hidden .git folder in your project directory. This .git folder contains all the information Git needs to track changes to your files.

### ****Step 4: Check Repository Status****

After initializing the repository, you can check its status again:

bash

CopyEdit

git status

* **Output if no files are being tracked:**

vbnet

CopyEdit

On branch master

No commits yet

Untracked files:

(use "git add <file>..." to include in what will be committed)

<file list>

The list of **untracked files** indicates files that Git sees in the folder but is not currently tracking.

### ****Step 5: Add Files to the Staging Area****

Before Git can track your files, you need to add them to the **staging area**. To add all files in the current directory, run:

bash

CopyEdit

git add .

* **What this does:**  
  This tells Git to stage all untracked files in the directory and its subdirectories.

### ****Step 6: Commit the Files****

To create a "checkpoint" (commit) for your project files, run:

bash

CopyEdit

git commit -m "add initial project files"

* **What this does:**
  + Creates a snapshot of your current project state.
  + The -m flag allows you to include a commit message describing what this commit represents.

**Example commit message:**  
Good: "Add initial project files"  
Bad: "Files" (not descriptive enough).

The "commit" command is used to save your changes to the local repository. Note that you have to explicitly tell Git which changes you want to include in a commit before running the "git commit" command. This means that a file won't be automatically included in the next commit just because it was changed.

### ****Step 7: Check the Clean State****

After committing, run:

bash

CopyEdit

git status

* **Output if no changes have been made since the commit:**

vbnet

CopyEdit

On branch master

nothing to commit, working tree clean

This means everything in your folder is up to date, and Git is tracking all changes.

### ****Step 8: Modify a File****

When you make changes to any file, Git will detect this. For example, if you add a **docstring** to geometry\_analysis.py:

python

CopyEdit

"""

This module has functions associated with analyzing the geometry of a molecule.

When run as a script and given an xyz file, this script will print out the bonds. Run

$ python geometry\_analysis.py --help

to see input options.

"""

Save your changes.

### ****Step 9: Stage the Modified File****

Run git status again to see the changes:

bash

CopyEdit

git status

* **Output for a modified file:**

rust

CopyEdit

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

modified: geometry\_analysis.py

Now, stage the modified file:

bash

CopyEdit

git add geometry\_analysis.py

### ****Step 10: Commit the Change****

After staging, create another commit with a descriptive message:

bash

CopyEdit

git commit -m "add docstring to geometry\_analysis.py"

### ****Key Concepts from the Instructions****

1. **Repository:**  
   A Git repository is a folder where Git tracks changes to the files.
2. **Untracked Files:**  
   These are files Git sees but is not tracking yet. You need to add them to the staging area with git add.
3. **Staging Area:**  
   A place where you prepare files for committing. Adding files to the staging area doesn’t commit them yet.
4. **Commit:**  
   A snapshot of the changes in the repository. Commits help you keep track of different versions of your project.

### ****Summary of Commands****

| **Command** | **Description** |
| --- | --- |
| cd <path> | Navigate to the project folder. |
| git init | Initialize a new Git repository. |
| git status | Check the status of the repository. |
| git add <file> or git add . | Add files to the staging area (. adds all files). |
| git commit -m "message" | Create a commit with a descriptive message (creating snapshot). |

### ****Understanding the Output****

From the git status command, Git is telling you:

1. **Files with changes that are not staged for commit:**

makefile

CopyEdit

modified: .ipynb\_checkpoints/geom\_analysis-checkpoint.ipynb

modified: Python Learning.docx

* + These files have changes, but you haven’t added them to the staging area. Git won’t include them in the commit unless you explicitly stage them.

1. **The action you took:** You staged and committed only geom\_analysis.ipynb:

bash

CopyEdit

git add geom\_analysis.ipynb

git commit -m "added docstring to geometry\_analysis.ipynb"

* + This successfully committed the changes to geom\_analysis.ipynb.

1. **The remaining issue:** After your commit, the other two files (.ipynb\_checkpoints/geom\_analysis-checkpoint.ipynb and Python Learning.docx) are still modified but not staged. That’s why git status continues to list them.

### ****Steps to Resolve****

If you want to include all the remaining changes in your repository, you have two options:

#### ****Option 1: Stage and Commit Remaining Files****

1. Stage all modified files:

bash

CopyEdit

git add .

* + This stages all changes in the current directory (including subdirectories).

1. Commit the staged changes:

bash

CopyEdit

git commit -m "Add updates to checkpoint and docx file"

* + This creates a commit for the remaining files.

#### ****Option 2: Selectively Stage Files****

If you only want to commit specific files, do the following:

1. Stage a specific file (e.g., Python Learning.docx):

bash

CopyEdit

git add "Python Learning.docx"

1. Commit the change:

bash

CopyEdit

git commit -m "Updated Python Learning.docx"

1. If you don’t want to commit .ipynb\_checkpoints/geom\_analysis-checkpoint.ipynb, you can ignore it using a **.gitignore** file (see below).

### ****Handling the**** .ipynb\_checkpoints ****Folder****

The .ipynb\_checkpoints folder is automatically created by Jupyter Notebook to store temporary checkpoints. These files are generally not needed in a Git repository.

1. **Add .ipynb\_checkpoints to .gitignore:** Create or edit a .gitignore file in your repository root and add:

CopyEdit

.ipynb\_checkpoints/

This will tell Git to ignore the .ipynb\_checkpoints folder in future commits.

1. **Remove .ipynb\_checkpoints from the repository if already tracked:** If the .ipynb\_checkpoints folder is already being tracked by Git, remove it using:

bash

CopyEdit

git rm -r --cached .ipynb\_checkpoints/

Then commit the change:

bash

CopyEdit

git commit -m "Remove .ipynb\_checkpoints from the repository"

### ****Verify the Status****

After committing the changes, check the status again:

bash

CopyEdit

git status

* If everything is committed, the output should be:

vbnet

CopyEdit

On branch main

nothing to commit, working tree clean

### ****Summary of Commands****

bash

CopyEdit

# Stage all changes

git add .

# Commit changes with a message

git commit -m "Add remaining updates"

# Ignore unnecessary checkpoint files

echo ".ipynb\_checkpoints/" >> .gitignore

git rm -r --cached .ipynb\_checkpoints/

git commit -m "Remove and ignore .ipynb\_checkpoints folder"

Let me know if you need clarification or help!