CHEM2000 – a Python primer

Paolo Raiteri

February 17, 2026

Contents

1. Introduction to Python

Python is a versatile, high-level programming language known for its simplicity and readability. Created by Guido van Rossum and first released in 1991, Python has grown to become one of the most popular programming languages in the world. Its philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. This document is a very short introduction to Python, and it is designed to be a quick introduction to Python for CHEM2000 students, with the aim of providing a summary of the key concepts that you'll need to solve the numerical laboratories that are part of the CHEM2000 unit using Python, but it is far from comprehensive.

1.1 Why Learn Python?

Python's popularity stems from several key features:

- **Easy to learn:** Python has a simple, clear syntax that is ideal for beginners.
- **Versatility:** It can be used for web development, data analysis, artificial intelligence, scientific computing, and more.
- **Large community:** Python has a vast, supportive community and a wealth of libraries and frameworks.
- **Career opportunities:** Python skills are in high demand across various industries.

1.2 Other useful resources

Simple python online courses for people with little or no previous programming experience can be accessed through the Software Carpentry :

- Plotting and Programming in Python
- Programming with Python
- The Unix Shell

Some of those courses are also run in person at Curtin through the Curtin Institute for Data Science, but they can be easily done independently and they are designed to be completed in a day (about 7h30m for the full python courses).

These are some other examples of (free) python online courses or books.

- Python programming for data science
- Automate the boring stuff
- A Python Book: Beginning Python, Advanced Python, and Python Exercises
- Think Pyhon
- Non-Programmer's Tutorial for Python
- Introduction to Python programming

1.3 Jupyter Notebooks in Google Colab

This document and the examples therein have been developed and tested using **Jupyter Notebook**, and may need some adjustment if run on the command line. Jupyter Notebooks are interactive computing environments that allow you to create and share documents containing live code, equations, visualizations, and explanatory text. They are particularly useful for data analysis, scientific computing, and programming education. In a Jupyter Notebook, you can write and execute code in various programming languages (such as Python, R, or Julia) alongside rich text explanations, making it easy to document your work step-by-step. This format enables you to combine computational output with narrative descriptions, creating a comprehensive and reproducible record of your analysis or experiment. Jupyter Notebooks are widely used in academic and professional settings for tasks like data cleaning, statistical modeling, machine learning, and creating reports with embedded visualizations. In particular we would recommend using Jupyter Notebooks through Google Colab.

Google Colab (short for Colaboratory) is a free, cloud-based platform that allows you to write and execute Python code through your browser. It provides a Jupyter Notebook environment with free access to GPU resources, making it ideal for machine learning and data science projects. Colab integrates seamlessly with Google Drive for easy file storage and sharing. It comes pre-installed with many popular data science libraries, eliminating the need for local setup and configuration. This platform is especially useful for students and researchers who need computational power without access to high-performance hardware.

A Jupyter notebook consists of a series of cells that can be either *code* or *text*, which can then be *executed* by pressing the **play** button or *Shift+Return* (if you want to look like a pro). For example, we can write the classic first program in a code cell

Python Code

```
print(f"Hello World")
```

and after it is run produced the expected output (Figure 1.1)

Output

```
Hello World
```

We can then add a *text* cell to include comments or background for our code. Text cells accept *Markdown* syntax and once executed will render the text. Markdown provides a simple way of producing formatted text and it accepts *LaTeX* commands for equations, which look much nicer that those made with the equation editor in MS Word (Figures 1.2). For a Markdown tutorial see for example here

1.4 Coding help

You can ask any python questions on Campuswire, but you would get a much quicker answer using google. In fact, chances are that someone has already asked your question on stackoverflow, *e.g.* you can search for "How to print formatted string in Python3?" or "Adding a legend to a plot"

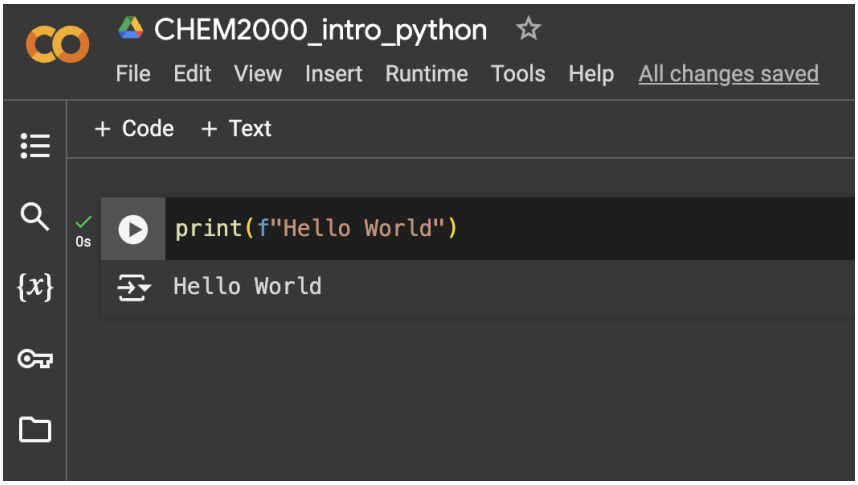

Figure 1.1: Screen capture of the "Hello World" program from Google Colab

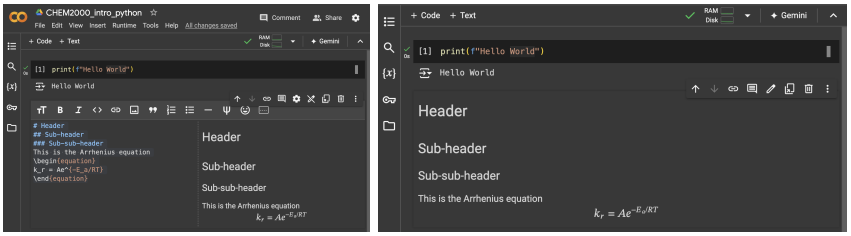

Figure 1.2: Example of markdown cell in Google Colab, as we type (left) and the final product (right)

.

If you are using Google Colab, you can use Gemini, the artificial intelligence (AI) system created by Google (Figure 1.3. Google Colab does offer free access to some Gemini models, but the specifics can change over time. At the time of writing, Google Colab offers free Access to Gemini Pro, which allows users to experiment with and use the model without cost. There are usage limits but Google doesn't always publicly specify what this are. This would be more than enough for the purposes of this laboratories. Alternative, other large language models can provide coding assistance.

1.5 Using Jupyter Notebooks for your research

Jupyter notebooks are powerful tools for advanced scientific calculations, data analysis, and visualization. They combine the functionality of a scientific calculator with the flexibility of a programming environment, allowing users to write and execute code in discrete cells, view results immediately, and iterate quickly, as well as document their work with markdown text and LaTeX equations.

Jupyter supports multiple programming languages, with Python being particularly popular for its rich ecosystem of scientific libraries. It allows for using powerful libraries like NumPy for numerical computations, Pandas for data manipulation, and Matplotlib's pyplot for creating high-quality figures. Users can write and execute Python code in discrete cells, performing complex mathematical operations and data analysis with immediate feedback. The notebook environment allows for rapid iteration and experimentation, making it ideal for exploratory data analysis. Scientists can generate publication-ready plots using pyplot, customizing every aspect from color schemes to font sizes. The ability

Figure 1.3: Example of coding assistance provided by Gemini.

to mix code with markdown text and LaTeX equations enables clear documentation of methods and results. This integrated approach streamlines the scientific workflow from initial calculations to final visualizations, enhancing reproducibility and facilitating the communication of findings.

Typically we would use the first cell as a text (Markdown) cell to add a title and some basic info about the content of the notebook. After that we would have a code cell, where we import the python packages that we require for the work.

Python Code

```
# python packages
import pandas as pd # Dataframes and reading CSV files
import numpy as np # Numerical libraries
import matplotlib.pyplot as plt # Plotting library
```

We can use the hash (`#`) symbol to add comment lines into a code cell.

Note that not every python package is available in google Colab, and you may have to install them manually. For example, if we try to load the `lmfit` packaged, which is used to do curve fitting, we would get an error.

Python Code

```
import lmfit
```

Output

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-2-42f2a570039d> in <cell line: 1>()
----> 1 import lmfit

ModuleNotFoundError: No module named 'lmfit'

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

This can be fixed by installing the required package by using the "magic function" (`%pip`).

Python Code

```
%pip install lmfit
import lmfit
```

Note that this need to be done only once per session, and after `lmfit` has been installed, it can be commented out.

2. Python basics

blurb

2.1 Importance of Indentation in Python

In Python, indentation is not merely a matter of style; it is a fundamental aspect of the language's syntax. Unlike many other programming languages that use braces or keywords to delineate code blocks, Python uses indentation to define the structure and scope of code. Consistent indentation is crucial for:

- **Block definition:** Indentation determines which statements are part of a particular code block, such as the body of a function, loop, or conditional statement.
- **Readability:** Proper indentation enhances code readability by visually representing the hierarchical structure of the program.
- **Error prevention:** Incorrect indentation can lead to logical errors or syntax errors, as the Python interpreter relies on indentation to understand the code's structure.
- **Consistency:** Maintaining consistent indentation (typically 4 spaces per level) is a best practice that facilitates collaboration and code maintenance.

Therefore, mastering indentation is essential for writing correct, readable, and maintainable Python code.

2.2 Formatted printing

Python offers several powerful methods for formatted printing, enabling developers to create well-structured and readable output. The most versatile approach is the f-string (formatted string literal), introduced in Python 3.6. F-strings are prefixed with f and allow embedding expressions inside curly braces . For example, `f"The result is {2 + 2}"` would print "The result is 4". Another method is the `.format()` function, which replaces placeholders in a string with specified values: `"The result is {}".format(2 + 2)`. These formatting techniques enable precise control over output appearance, including alignment , padding, decimal precision, and even calling functions within the formatted string (`f"{func(x)}"`). By mastering these tools, programmers can generate clear, customized text output for various applications, from simple console messages to complex report generation. The `:` used to separate the quantity to be printed out and the formatting.

Python Code

```
print(f"{22888.6641:.2f}") # Float with 2 decimal places
print(f"{22888.6641:>10.2f}") # 10 digits float with 2 decimal places (right aligned)
print(f"{22888.6641:<10.2f}") # 10 digits float with 2 decimal places (left aligned)
print(f"{22888.6641:<10.2g}") # Exponential notation
print(f"{22888.6641:<10.2g} or {22888.6641:<10.2f}")
```

Output

```
22888.66
  22888.66
22888.66
2.3e+04
2.3e+04    or 22888.66
```

Note that you can add as many `{}` as you like, and you can use variables instead of the actual value to be printed.

2.3 Variables

In Python, variables are used to store data that can be referenced and manipulated throughout a program. Variables can hold different types of data, such as numbers, strings, lists, and more. Unlike some other programming languages, Python does not require you to explicitly declare the type of a variable when you create it. The type is inferred from the value assigned to the variable.

2.3.1 Variable Names

Variable names in Python must adhere to the following rules:

- They must start with a letter (a-z, A-Z) or an underscore (_).
- The rest of the variable name can contain letters, numbers (0-9), or underscores.
- Variable names are case-sensitive, which means `Variable` and `variable` would be considered different variables.

Here are some examples of valid and invalid variable names:

- Valid: `my_var`, `a123`, `_temp`
- Invalid: `2cool4school`, `my-var`, `is student`

2.3.2 Types of Variables

Python supports several data types, including but not limited to:

- `int` (Integer): Whole numbers, e.g., `10`, `-3`
- `float` (Floating Point): Decimal numbers, e.g., `3.14`, `-0.001`
- `str` (String): Sequence of characters, e.g., `"Hello"`, `'Python'`
- `bool` (Boolean): Represents `True` or `False`
- `list` (List): Ordered, mutable collection of items, e.g., `[1, 2, 3]`, `["apple", "banana"]`, or mixed types `["apple", 1.2, True]`
- `dict` (Dictionary): Unordered collection of key-value pairs, e.g., {`"name": "Alice", "age": 25`}

2.3.3 Creating Variables

To create a variable in Python, you simply assign a value to a name using the equals sign (=). For example:

Python Code

```
x = 10
name = "Alice"
is_student = True
my_list = [1,2,3]
my_dict = {"temperature": 300, "pressure": 1}
```

In this example, `x` is an integer variable, `name` is a string variable, and `is_student` is a boolean variable. Variables can also be created from the output of a function, as we will see later.

2.3.4 Changing Variable Values

Once a variable is created, you can change its value by assigning a new value to it. You can also use the value of the variable to evaluate a methematical operation and update the value of the variable itself. For example:

Python Code

```
x = 10
print(x)  # Output: 10
x = 20
print(x)  # Output: 20
x = x + 2
print(x)  # Output: 22
```

Output

```
10
20
22
```

In this example, the variable `x` is first assigned the value `10`, and then it is reassigned the value `20`.

It is important to remember that the notebook keeps in memory all the variables that we have defined since its inception, even if they have been later deleted. This could cause unpredictable behaviour if the same variable name has been used in multiple cells with for different purposes, or if the cells are run out of sequence. It is good practice to periodic restart the "kernel" of the Jupyter Notebook and rerun all the cells from the very beginning.

2.4 Lists

Even more useful than variables are lists and arrays, which allow us to store many values in one place. Lists can be created by hand or be the output of other Python functions. They can contain numbers, strings or other variables, or mixed types

Python Code

```
listOfNumbers = [300, 2, 3.2]
print("One dimensional list of numbers :",listOfNumbers)
listOfStrings = ["Temperature" , "Pressure" , "Volume"]
print("One dimensional list of strings :",listOfStrings)
mixedString = ["temperature" , 300]
print("The mixed list is :",mixedString)
```

Output

```
One dimensional list of numbers : [300, 2, 3.2]
One dimensional list of strings : ['Temperature', 'Pressure', 'Volume']
The mixed list is : ['temperature', 300]
```

2.4.1 List slicing

We can easily access one (or more) of the elements of the list, by specifying their location in the list. **Note that Python starts counting from zero!**

Python Code

```
print("Second element of the list of numbers :",listOfNumbers[1])
print("Second element of the list of strings :",listOfStrings[1])
```

Output

```
Second element of the list of numbers : 2
Second element of the list of strings : Pressure
```

Remember that in python the first index is included while the second isn't. So if we want the last to elements of the list we can use

Python Code

```
print("Last two elements of the list of numbers :",listOfNumbers[1:3])
```

Output

```
print("Last two elements of the list of numbers :",listOfNumbers[1:3])
```

2.5 Dictionaries

Python also has another class of data, called dictionaries, which shares many similarities with lists. The main difference is that dictionaries are not "ordered" which means that any loop over their elements may produce different outputs after any execution. Each entry of a dictionary is defined by its label, and it can be of any kind, string, number, list, array, function... The label can be used to access the values, similarly to the index for a list.

Python Code

```
student = {
"Name" : "John",
"Age" : "23",
"Mark" : 60
}
print(student)
print(f"{student['Name']}'s age is {student['Age']} ")
```

Output

```
{'Name': 'John', 'Age': '23', 'Mark': 60}
John's age is 23
```

Dictionaries also have built-in functions that make it easy to loop over their elements, suck as `.items()`

Python Code

```
for key, value in student.items():
  print(f"{key} : {value}")
```

Output

```
Name : John
Age : 23
Mark : 60
```

2.6 Functions

Functions can be used to perform (complicated) operations on the input variables, and return a result that can be stored in a variable where the function was called. Python executes commands sequentially, therefore functions must be defined before they are used The function below illustrates how to define a function that computes the sum of two numbers.

Python Code

```
def sumAB(a,b):
total = a + b
return total
a = 10
b = 21
c = sumAB(a,b)
print(c)
```

Output

```
31
```

Note again how the indentation is used to define what code is inside the function and what is outside.

Functions can take different variable type as input, and the returned value is automatically adjusted to match the input type (at least for the simple functions that will be used in this unit). The code below illustrated how to define a simple function, which evaluates a mathematical expression, a parabola, for a single value or an array of values.

Python Code

```
def parabola(x,a,b,c):
  y = a*x**2 + b*x +c
  return y
a = 2.5
b = -0.1
c = .128

# Evaluate the parabola at one point
xValue = 1
yValue = parabola(xValue,a,b,c)
print(type(xValue),type(yValue))
print(xValue,yValue)
print("-"*30)

# Evaluate the parabala at an array of points
xArray = np.arange(0,5,0.1)
yArray = parabola(xArray,a,b,c)
print(type(xArray),type(yArray))
print(xArray)
print(yArray)
```

Output

```
<class 'int'> <class 'float'>
1 2.528
------------------------------
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7
 1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5
 3.6 3.7 3.8 3.9 4.  4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9]
[ 0.128  0.143  0.208  0.323  0.488  0.703  0.968  1.283  1.648  2.063
  2.528  3.043  3.608  4.223  4.888  5.603  6.368  7.183  8.048  8.963
  9.928 10.943 12.008 13.123 14.288 15.503 16.768 18.083 19.448 20.863
 22.328 23.843 25.408 27.023 28.688 30.403 32.168 33.983 35.848 37.763
 39.728 41.743 43.808 45.923 48.088 50.303 52.568 54.883 57.248 59.663]
```

3. Using Python as a scientific calculator

Jupyter Notebooks can be used as a powerful scientific calculator, offering a wide range of mathematical capabilities through its built-in functions and libraries. With Jupyter Notebooks, you can perform basic arithmetic operations, as well as more complex calculations involving trigonometry, logarithms, and exponentials. The Python math module provides access to advanced mathematical functions, while libraries like NumPy extend Jupyter Notebooks's capabilities to include array operations, linear algebra, and statistical functions. Jupyter Notebooks's interactive mode allows for quick calculations, making it ideal for on-the-fly problem-solving. Its ability to handle variables, define custom functions, and create plots using libraries like Matplotlib further enhances its utility as a scientific calculator. This versatility makes Jupyter Notebooks an excellent tool for students and researchers across various scientific disciplines, from physics and engineering to data analysis and computational biology.

3.1 Mathematical operation in Python

The basic mathematical operations that are directly available in Python are:

+ Addition

- Subtraction

* Multiplication

/ Division

** Exponentiation

% Modulus

// Integer division (floor)

Python Code

```
# Basic mathematical operations in Python
a = 5 + 3 # Addition
print(f"a = {a}")

b = 10 - 4 # Subtraction
print(f"b = {b}")

c = 2 * 6 # Multiplication
print(f"c = {c}")

d = 14 / 3 # Division
print(f"d = {d}")

e = 17 // 3 # Integer division (floor division)
print(f"e = {e}")

f = 17 % 3 # Modulus (remainder)
print(f"f = {f}")
```

```python
g = 2 ** 3 # Exponentiation
print(f"g = {g}")

h = 2 ** (1/2) # Square root (slow)
print(f"h = {h}")
```

Output

```
a = 8
b = 6
c = 12
d = 4.666666666666667
e = 5
f = 2
g = 8
h = 1.4142135623730951
```

Note how the division between two integers produces a floating point number. Also note how through the exponentiation you can also access the square root (or any n-root), however, that operation is not-optimised for performance and could make your code very slow.

In fact, for more complicated mathematical operation we want to use `NumPy` (or `math`), which also contains a some mathematical constants

Python Code

```python
print(f"The approximate value of pi is : {np.pi}")
print(f"The approximate value of the Euler constant (e) is : {np.e}")
print(f"The square root of two is : {np.sqrt(2)}")
print(f"The natural logarithm of two is : {np.log(2)}")
print(f"The logarithm base 10 of two is : {np.log10(2)}")
print(f"The sine of pi is : {np.sin(np.pi)}")
print(f"The cosine of pi is : {np.cos(np.pi)}")
```

Output

```
The square root of two is : 1.4142135623730951
The natural logarithm of two is : 0.6931471805599453
The logarithm base 10 of two is : 0.3010299956639812
The sine of pi is : 1.2246467991473532e-16
The cosine of pi is : -1.0
The approximate value of pi is : 3.141592653589793
The approximate value of the Euler constant (e) is : 2.718281828459045
```

4. NumPy

NumPy (Numerical Python) is a fundamental library for scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy's efficiency and ease of use make it an essential tool for data manipulation and scientific computing in Python.

4.1 Key Features of NumPy

- **ndarray:** The core of NumPy is the ndarray object, an efficient multi-dimensional array that allows for vectorized operations.
- **Mathematical operations:** NumPy provides a wide range of mathematical functions that operate on entire arrays without the need for explicit loops.
- **Broadcasting:** This feature allows operations on arrays of different shapes, making code more concise and efficient.
- **Integration:** NumPy integrates well with other scientific Python libraries, forming the foundation for many data science and machine learning tools.

4.2 Basic Usage

To use NumPy, you typically start by importing the package.

Python Code

```
import numpy as np
```

With NumPy, you can create arrays, perform mathematical operations, and manipulate data with ease:

In many respects, NumPy arrays behave similarly to lists, but they would allow us to do "array" operations, which are not possible with lists. Let's for example create

Python Code

```
# Create a numpy array from a list
my_list = [1, 2, 3, 4, 5]
my_array = np.array(my_list,dtype=float)
print(type(my_array))
print(my_array)
```

Output

```
<class 'numpy.ndarray'>
[1. 2. 3. 4. 5.]
```

4.3 Slicing

We can select elements from a NumPy array similarly to what we did with lists or by using a `mask`, *i.e.* using a list of booleans with the same length as the array itself.

Python Code

```
# Create a numpy array from a list
my_list = [1, 2, 3, 4, 5]
print(type(my_list))
print(f"List = {my_list}")

my_array = np.array(my_list,dtype=float)
print(my_array[2])
print(my_array[2:4])

# Select first and fourth elements
print(my_array[ [True,False,False,True,False] ])

# Select only the even numbers of the array
mask = list(my_array % 2 == 0)
print(mask)
print(my_array[mask])
```

Output

```
<class 'list'>
List = [1, 2, 3, 4, 5]
3.0
[3. 4.]
[1. 4.]
[False, True, False, True, False]
[2. 4.]
```

4.4 Array operations

This are operations done on every element of the array.

Python Code

```
# Create a numpy array from a list
my_list = [1, 2, 3, 4, 5]
print(type(my_list))
print(f"List = {my_list}")

my_array = np.array(my_list,dtype=float)
print(type(my_array))
print(f"First array  = {my_array}")

# Add 1 to each element in the array
new_array = my_array + 1
print(f"Second array = {new_array}")

# Multiply each element in the array by 2
new_array = my_array * 2
```

```
print(f"Third array  = {new_array}")

# Square all the elements
new_array = my_array ** 2
print(f"Fourth array = {new_array}")

# Add two arrays together
new_array = my_array + new_array
print(f"Fifth array  = {new_array}")
```

Output

```
<class 'list'>
List = [1, 2, 3, 4, 5]
<class 'numpy.ndarray'>
First array  = [1. 2. 3. 4. 5.]
Second array = [2. 3. 4. 5. 6.]
Third array  = [ 2.  4.  6.  8. 10.]
Fourth array = [ 1.  4.  9. 16. 25.]
Fifth array  = [ 2.  6. 12. 20. 30.]
```

On the other hand, the mathematical operations have different meanings when used on lists, or do not work, depending on the types involved in the operation.

Python Code

```
# Concatenate two lists
print(my_list + my_list)
# Triple the list
print(my_list * 3)
# No allowed
print(my_list + 1)
```

Output

```
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-54-67c64142f0ae> in <cell line: 6>()
      4 print(my_list * 3)
      5 # No allowed
----> 6 print(my_list + 1)

TypeError: can only concatenate list (not "int") to list
```

5. Loops and conditionals

Loops are fundamental constructs in programming, serving as the backbone for efficient and scalable code. They allow developers to execute a block of instructions repeatedly, automating repetitive tasks and processing large sets of data with minimal code duplication. Loops are essential for iterating through arrays, lists, and other data structures, enabling operations on each element without explicitly writing code for every item. They facilitate dynamic problem-solving, allowing programs to continue operations until specific conditions are met. Common types include 'for' loops for known iteration counts and 'while' loops for condition-based repetition. By reducing code redundancy and enhancing readability, loops contribute significantly to program efficiency and maintainability. Their versatility makes them indispensable in various scenarios, from simple counting tasks to complex algorithms, forming a critical component of computational thinking and software development.

There are many ways of creating loops in python, and here are some examples to loop over the elements of a list. It important to note here that the indentation of the code determines where the loop finishes.

Python Code

```
my_list = [1, 2, 3, 4, 5]

# Direct loop over the elements of a list
for x in my_list:
  print(x)
print("--- This is outside the loop ---")

# Use an iterator, an the len() function to get the list length
print(f"List length {len(my_list)}")
for i in range(0,len(my_list)):
  print(i, my_list[i])
```

Output

```
1
2
3
4
5
--- This is outside the loop ---
List length 5
0 1
1 2
2 3
3 4
4 5
```

Note that the upper bound of the iterator is not included.

NumPy also has a built-in iterator, which allows for iterating over floating point numbers with arbitrary stride

Python Code

```
for i in np.arange(1,4,0.5):
  print(i)
```

Output

```
1.0
1.5
2.0
2.5
3.0
3.5
```

5.1 Use of variables, lists and loops

As an illustrative example of using lists and variables we can now compute the sum and average of all the elements in an list of numbers. In order to do that we initialise a variable to zero, and progressively sum the list elements to it. Note how we can increment the value of the variable using the `+=` operator; `a = a + 1` is equivalent to writing `a += 1`

Python Code

```
listOfNumbers = [1,2,3,4,5,6,7,8,9,10]
summ = 0.
print("The initial value of summ is {}",(summ))
for i in range(0,len(listOfNumbers)):
  value = listOfNumbers[i]
  summ += value
  print("Iteration {}, list element {}, value of summ {}". \
        format(i+1,value,summ))
print("The final value of summ is {}".format(summ))
average = summ / len(listOfNumbers)
print("The average is {}".format(average))
```

Output

```
The initial value of summ is {} 0.0
Iteration 1, list element 1, value of summ 1.0
Iteration 2, list element 2, value of summ 3.0
Iteration 3, list element 3, value of summ 6.0
Iteration 4, list element 4, value of summ 10.0
Iteration 5, list element 5, value of summ 15.0
Iteration 6, list element 6, value of summ 21.0
Iteration 7, list element 7, value of summ 28.0
Iteration 8, list element 8, value of summ 36.0
Iteration 9, list element 9, value of summ 45.0
Iteration 10, list element 10, value of summ 55.0
The final value of summ is 55.0
The average is 5.5
```

Alternatively, a much more efficient way of doing the same procedure is to use some of the NumPy built-in functions to summ the elements of an array (or a list).

Python Code

```
listOfNumbers = [1,2,3,4,5,6,7,8,9,10]
summ = np.sum(listOfNumbers)
average = summ / len(listOfNumbers)
print("The averge is {}".format(average))

average = np.mean(listOfNumbers)
print("The averge is {}".format(average))
```

Output

```
The averge is 5.5
The averge is 5.5
```

6. Conditional statements

Conditional statements are essential control structures in programming that allow code to make decisions and execute different actions based on specific conditions. Conditionals can evaluate comparisons, logical operations, or function results, providing flexibility in defining execution paths. These statements enable programs to exhibit dynamic behavior, adapting to various inputs and states. They are fundamental to creating responsive and intelligent programs, from simple user input validation to complex algorithmic decision trees, as well as handling errors and invalid data. The most common form is the "if-else" statement, where code checks a boolean condition and executes one block of code if true and another if false. More complex decision-making can be achieved with "else if" clauses or switch/case statements in some languages. Similarly to loops, Python uses the indentation to define what is inside a conditional clause

```
if ... :
    # do something
elif ... :
    # do something
else:
    # do something
```

Note that you must add a `:` after the condition and that you can have as many `elif` statements as you need. Remember that after a condition is found to the `True` and the corresponding code is executed, the code jumps at the end of the conditional block and the remaining conditions are not evaluated, *i.e* only the first `True` condition is used. Python has a number of operators that can be used in conditional statements, see Table 6.1.

Operator	**Description**
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
is	Object identity
is not	Negated object identity
in	Membership
not in	Negated membership
and	Logical AND
or	Logical OR
not	Logical NOT

Table 6.1: Conditional Operators in Python

Let's make an example to show the what is the the syntax in Python

Python Code

```
listOfNumbers = [1, 2, 3, 4, 7.6, 3.9]
for x in listOfNumbers:
  if x % 2 == 0:
    print(f"{x} is even")
  elif x % 2 == 1:
    print(f"{x} is odd")
  else:
    print(f"{x} is not an integer")
```

Output

```
1 is odd
2 is even
3 is odd
4 is even
7.6 is not an integer
3.9 is not an integer
```

6.1 While loops

We can also combine loops and conditionals to execute a code until a given condition is met. Care has to be paid when using `while` loops, as if the condition is never met due to coding errors the program will run forever, or until it crashes or it is stopped.

Python Code

```
x = 0
while x < 10:
  print(x)
  x += 1
```

Output

```
0
1
2
3
4
5
6
7
8
9
```

Continue and break

It is also possible to control what gets executed in a loop using `continue` and `break`.

Python Code

```
for i in range(10):
  if i % 2 == 0:
    continue # go to the next iteration
  print(i)

print("-"*30)
for i in range(10):
  if i == 4:
    break # leave the loop
  print(i)
```

Output

```
1
3
5
7
9
------------------------------
0
1
2
3
```

7. Plotting using matplotlib and pyplot

Pyplot, a module within the Matplotlib library in Python, provides a convenient interface for creating high-quality figures and plots. After having imported `pyplot`, the basic workflow involves calling plt.figure() to create a new figure, then using various plotting functions like `plt.plot()` for line graphs, `plt.scatter()` for scatter plots, or `plt.bar()` for bar charts. These functions accept data in the form of arrays or lists. Customization options are extensive: `plt.xlabel()` and `plt.ylabel()` set axis labels, `plt.title()` adds a title, and `plt.legend()` includes a legend. Color, line style, marker type, and other visual elements can be adjusted using additional parameters. Multiple plots can be combined on a single figure using `plt.subplot()`. Once the plot is configured, `plt.show()` displays it, or `plt.savefig()` saves it to a file. This flexible and intuitive approach allows users to create a wide range of visualizations, from simple graphs to complex, publication-ready figures.

Python Code

```python
# Import the matplotlib library for plotting
import matplotlib.pyplot as plt
# Import the numpy library for numerical operations
import numpy as np

# Define the parabola function
# with default values for the parameters
def parabola(x,a=2.5 ,b=-0.1, c=0.128):
  y = a*x**2 + b*x +c
  return y

# Create an array of x values
x = np.linspace(-3, 7, 20)
# Calculate the corresponding y values
y = parabola(x,a,b,c)

# Create a figure
figure = plt.figure
# Plot the parabola as a line
plt.plot(x,y,label="line",color='blue')
# Plot the parabola as scatter points
plt.scatter(x,y,marker='o',label="data",color='red')
# Add a horizontal line at y=0
plt.axhline(0, color='black',linestyle="-.")
# Add a vertical line at x=0
plt.axvline(0, color='black',linestyle="--")
# Add a label to the x-axis
plt.xlabel('x label [units]')
# Add a label to the y-axis
plt.ylabel('y label [units]')
# Add a title to the plot
plt.title('Parabola')
# Add a legend
plt.legend()
# Save the figure to an image file
plt.savefig('parabola.png')
# Display the plot
```

```
plt.show()
```

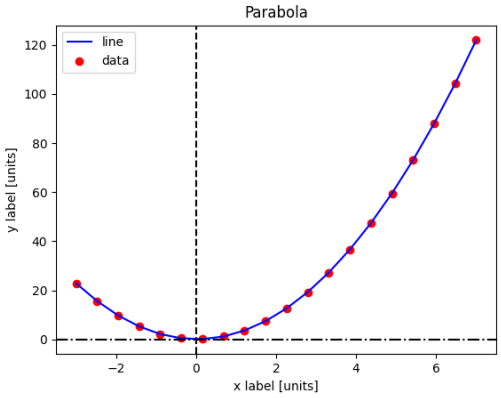

Figure 7.1: Caption

Python Code

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

# Generate some data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create a figure with two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot sin(x) on the first subplot
ax1.plot(x, y1, 'b-', label='sin(x)')
ax1.set_title('Sine Function')
ax1.set_xlabel('x')
ax1.set_ylabel('sin(x)')
ax1.legend()
ax1.grid(True)

# Plot cos(x) on the second subplot
ax2.plot(x, y2, 'r-', label='cos(x)')
ax2.set_title('Cosine Function')
ax2.set_xlabel('x')
ax2.set_ylabel('xos(x)')
ax2.legend()
ax2.grid(True)

# Adjust the layout and display the plot
plt.tight_layout()
plt.show()
```

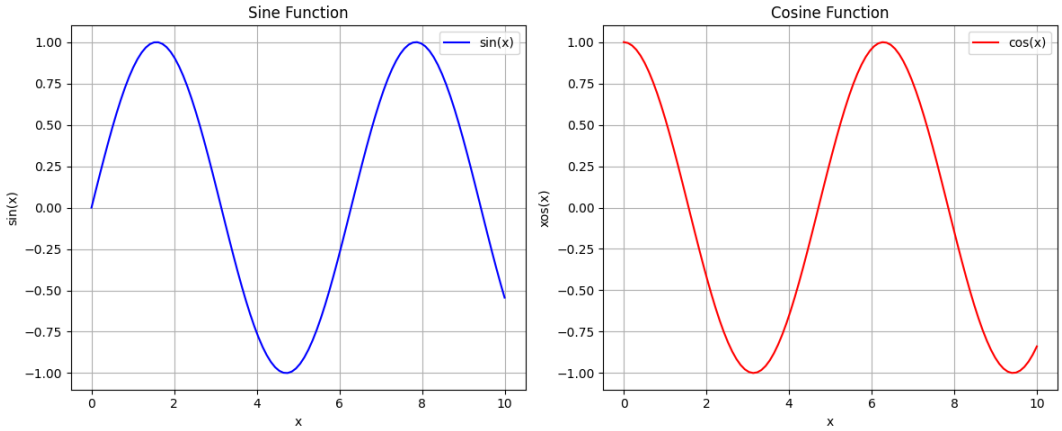

Figure 7.2: Caption

8. Reading data from file

Reading files into Python code is a fundamental skill that greatly enhances the versatility and power of your programs. This ability allows you to work with external data sources, making your code more flexible and applicable to real-world scenarios. By importing data from files, you can process large datasets, analyze experimental results, or load configuration settings without hard-coding values. This skill is crucial in scientific computing, data analysis, and many other fields where data is often stored in various file formats like CSV, TXT, or JSON. Reading files enables you to separate your data from your code, promoting better organization and reusability. It also allows your programs to interact with outputs from other software or instruments, facilitating interdisciplinary work and data exchange. As you progress in your programming journey, mastering file input will open up new possibilities for data manipulation, visualization, and complex analysis, making your code more robust and adaptable to different research or project needs.

NumPy's `genfromtxt` function offers a versatile method for importing CSV files into NumPy arrays. To use it, first import NumPy with `import numpy as np`. Then, call `np.genfromtxt()` with the CSV filename as the first argument. Key parameters include `delimiter` to specify the character separating values (typically ',' for CSV), `skip_header` to ignore a certain number of initial rows, and `usecols` to select specific columns. Set `names=True` to use the first row as column names, creating a structured array. For missing data, use `filling_values` to replace them with specified values. Genfromtxt automatically handles various data types, converting strings to floats or integers as appropriate. After importing, the resulting NumPy array can be manipulated using NumPy's powerful array operations, making it ideal for numerical analysis and scientific computing tasks.

Let's start by generating a CSV file with two columns X and Y.

Python Code

```
import numpy as np

# Define the data type for the structured array
dt = np.dtype([('X', np.float64), ('Y', np.float64)])

ndata = 5
x = np.random.uniform(0,1,ndata)
y = np.random.uniform(5,10,ndata)

array = np.zeros(shape=ndata, dtype=dt)
for i in range(ndata):
    array[i] = (x[i],y[i])

# Save the structured array as a CSV file
np.savetxt('results1.csv', array, delimiter=',',
           fmt="%f,%f", header='X,Y', comments='')
```

We can know write a code that reads the file and writes the values in the two columns.

Python Code

```
# Import the numpy library for numerical operations
import numpy as np

# Read the CSV file 'results1.csv', using the first line as column names
# The 'delimiter' parameter specifies that values are separated by commas
# 'names=True' tells numpy to use the first row as column names
data = np.genfromtxt('results1.csv', delimiter=',', names=True, comment="#)

# Access the column names from the data's dtype (data type) attribute
# For structured arrays, dtype.names contains the field names
column_names = data.dtype.names

# Print the column names to see the structure of our data
print("Column names:", column_names)

# Print the total number of elements in the array
# For a structured array, this is the number of rows
print(f"Number of points = {data.size}")

# Access data from the first column using its name
# column_names[0] gives us the name of the first column
x = data[column_names[0]]
print(f"X values = {x}")

# column_names[1] gives us the name of the second column
y = data[column_names[1]]
print(f"Y values = {y}")

# The structured numpy array can also be converted
# into a normal numpy array, i.e. without columns
data = data.view((data.dtype[0], len(data.dtype.names)))
```

Output

```
Column names: ('X', 'Y')
Number of points = 5
X values = [0.797733 0.627922 0.038332 0.546479 0.861912]
Y values = [7.837871 5.879141 7.551882 8.784729 5.550526]
```

8.1 Processing multiple files

One of the most common use of Python, and other programming languages is to automate repetitive tasks on different input data sets. Let's for example imagine that we want to compute the averages of 3 data sets, each which is stored in a separate file. First of all we need to generate some data and store them in files

Python Code

```
import numpy as np

# Initialise the random number generator for reproducibility
np.random.seed(123)

# Initialise the number of data sets and ...
numberOfDataSets = 3
```

```python
# ... number of values per data set
numberOfValues = 100

# Generate 3 random integers between 0 and 100
random_integers = np.random.randint(0, 101, numberOfDataSets)
print(f"Random Integers: {random_integers}")
print("-"*40)

# Generate normally distributed numbers with random means
for i in range(numberOfDataSets):
    x = random_integers[i]
    values = np.random.normal(x, 2, size=numberOfValues)
    # Save the array to a text file named 'data_set_.txt'
    filename = f"data_set_{i}.txt"
    np.savetxt(filename, values)
```

We can now read the files and process them and compute the averages of the values in each file.

Python Code

```python
import numpy as np

numberOfDataSets = 3

# Read the data into an numpy structured array
for i in range(numberOfDataSets):
    filename = f"data_set_{i}.txt"
    print(f"Opening file: {filename}")
    data = np.genfromtxt(filename)
    print(f"Number of values in file: {len(data)}")
    print(f"Mean: {np.mean(data)}") # Compute average
    print(f"Standard deviation: {np.std(data)/np.sqrt(len(data))}") compute StDev
    print(f"Expectation value: {random_integers[i]}") # Expected mean
    print("-"*40)
```

Output

```
Random Integers: [66 92 98]
----------------------------------------
Opening file: data_set_0.txt
Number of values in file: 100
Mean: 66.06244289275632
Standard deviation: 0.21377370231682927
Expectation value: 66
----------------------------------------
Opening file: data_set_1.txt
Number of values in file: 100
Mean: 91.90184372214968
Standard deviation: 0.19435581036223387
Expectation value: 92
----------------------------------------
Opening file: data_set_2.txt
Number of values in file: 100
Mean: 97.72519773730632
Standard deviation: 0.17700701681100545
Expectation value: 98
----------------------------------------
```

Alternatively, we can store the data sets and the averages in lists of further processing or plotting. Note that Python lists can contain any type of data, including other lists, structured arrays, ...

Python Code

```
# Empty list to store the datasets
listOfData = []

# Read the data into an numpy array
for i in range(numberOfDataSets):
    data = np.genfromtxt( f"data_set_{i}.txt" )
    # Append the data to the list
    listOfData.append(data)

# Empty list to store the averages
listOfAverages = []
for i in range(numberOfDataSets):
    # Get the data set from the list
    data = listOfData[i]
    # Append the average the the list
    listOfAverages.append(np.mean(data))
    print(f"Average from data set {i}: {listOfAverages[i]}")
```

Output

```
Average from data set 0: 66.06244289275632
Average from data set 1: 91.90184372214968
Average from data set 2: 97.72519773730632
```

9. Solving a kinetics problem with Python

The decomposition of n-propyl chloride:

$$CH_3CH_2CH_2Cl(g) \rightarrow CH_3CH = CH_2(g) + HCl(g)$$

has been studied by measurement of the pressure increase in a constant volume system. Starting with pure n-propyl chloride at an initial pressure of 112 Torr and 713 K, the following pressures, p, were observed at the specified times, t: Determine the order of the

t/min	15	30	45	60	75
$p/Torr$	136	155	170	182	191

reaction and calculate the rate constant.

Solution: First of all we recognise that we need the pressure of the reactant to determine the order of the reaction. Let's write the reaction as

$$A \rightarrow B + C$$

and build the ICE table. Here we use the equilibrium as the condition at any time after the reaction has started.

	A	B	C
I	p_0	0	0
C	$-x$	$+x$	$+x$
E	$p_0 - x$	$+x$	$+x$

We can then write the total pressure as, using Dalton's law

$$p_{tot} = p_0 - x + x + x = p_0 + x$$

Hence the pressure of the reactant becomes

$$p_A = p_0 - x = p_0 - (p_{tot} - p_0) = 2p_0 - p_{tot}$$

Python Code

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

# Input data
time = np.array([0, 15, 30, 45, 60, 75])
pres_total = np.array([112, 136, 155, 170, 182, 191])

# Partial pressure of the reactant
pres = 2*pres_total[0] - pres_total

# Graphical method to determine the order of the reaction
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 5))
```

```
ax1.set_title('zeroth order')
ax1.plot(time, pres, marker='o')
ax1.set_xlabel('Time')
ax1.set_ylabel('Pressure')

ax2.set_title('first order')
ax2.plot(time, np.log(pres), marker='o')
ax2.set_xlabel('Time')
ax2.set_ylabel('ln(Pressure)')

ax3.set_title('second order')
ax3.plot(time, 1./pres, marker='o')
ax3.set_xlabel('Time')
ax3.set_ylabel('1/Pressure')
plt.savefig('order.png')
plt.show()
```

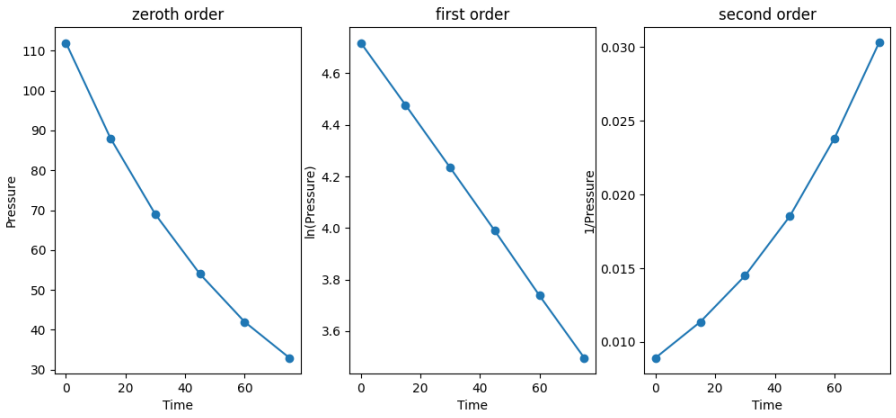

Figure 9.1: Graphical method to check the order of a reaction

Now that we know that the reacion if first order we can use the corresponding intrgrated rate law

$$\ln p_A(t) = \ln p_0 - k_r t$$

to determine the rate constant (k_r) by doing a linear fit.

Python Code

```
import lmfit
import numpy as np

def line(x,a=1,b=1):
    return a*x+b

# The reaction is first order
# Hence we fit log(p) vs time with a line
model = lmfit.Model(line)
params = model.make_params()
result = model.fit(np.log(pres), params, x=time)
print(result.fit_report())
result.plot()
plt.savefig('rate_constant.png')
plt.xlabel('Time')
```

```
plt.ylabel('ln(Pressure)')
plt.show()

# The rate constant is the negative of the slope
print(f"Rate constant = {-result.values['a']}")
```

Output

```
    [[Model]]
    Model(line)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 7
    # data points      = 6
    # variables        = 2
    chi-square         = 3.7197e-05
    reduced chi-square = 9.2993e-06
    Akaike info crit   = -67.9462170
    Bayesian info crit = -68.3626980
    R-squared          = 0.99996458
[[Variables]]
    a: -0.01633158 +/- 4.8598e-05 (0.30%) (init = 1)
    b:  4.72128490 +/- 0.00220705 (0.05%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(a, b) = -0.8257
--------------------------------------------------------------------------------
Rate constant = 0.016331582087159662
```

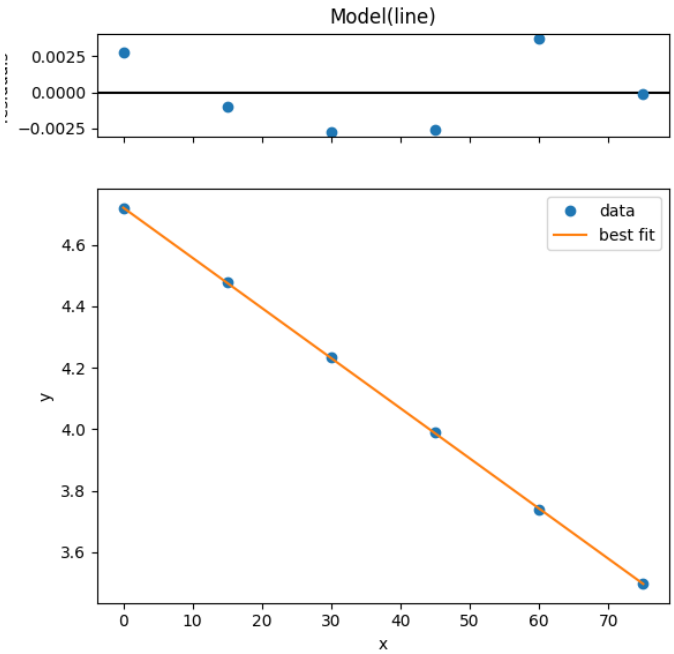

Figure 9.2: Linear fit to determine the rate constant of a chemical reaction

10. Plotting ideal gas isotherms

Let's write a code to plot selected ideal gas isotherms (p *vs* V) using the ideal gas law

$$pV = nRT$$

In this code we will use `numpy structured arrays`, to store the data and `pyplot` to make the final image.

Python Code

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

# Define the structure of our array
# 'f8' means 64-bit float
dt = np.dtype([('Temperature (K)', 'f8'),
               ('Volume (L)', 'f8'),
               ('Pressure (bar)', 'f8')])

# Create the temperature and pressure arrays
# 5 values linearly spaced in the range [100:300] - both included
arrayOfTemperatures = np.linspace(100, 300, 5)
# values in the range [0.1:1[ with spacing 0.02
arrayOfpressures = np.arange(0.1, 1, 0.02)

# Calculate the total size of our final array
total_size = len(arrayOfTemperatures) * len(arrayOfpressures)

# Create an empty structured array
data = np.empty(total_size, dtype=dt)

# Function to calculate the volume of an ideal gas
def idealVolume(T, p):
    R = 8.314  # Ideal gas constant in J/mol/K
    n = 1  # Number of moles
    conversionFactor = 0.01  # Conversion factor between J/bar to litre
    return (n * R * T / p) * conversionFactor

# Fill the structured array
index = 0
for T in arrayOfTemperatures:
    volumes = idealVolume(T, arrayOfpressures)
    for V, P in zip(volumes, arrayOfpressures):
        data[index] = (T, V, P)
        index += 1

# Create the figure
plt.figure()

# Plot the pressure and volume for each temperature
for temperature in arrayOfTemperatures:
    # Create a mask to extract the pressure and volume for the current temperature
    mask = data['Temperature (K)'] == temperature
    # Extract the pressure
    press = data[mask]['Pressure (bar)']
    # Extract the volume
```

```
    vol = data[mask]['Volume (L)']
    # Plot the pressure and volume for the current temperature
    plt.plot(vol, press, label=f'T = {temperature} K')

# Add labels
plt.ylabel('Pressure (bar)')
plt.xlabel('Volume (L)')

# Add a legend
plt.legend()

# Save the figure
plt.savefig('idealGas.png')

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

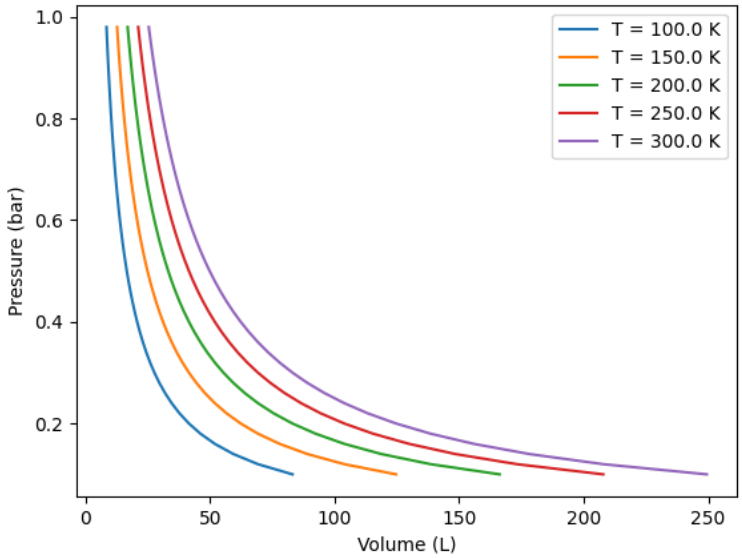

Figure 10.1: PV isotherms for an ideal gas.