

PR Grundlagen der Datenverarbeitung mit Python

Tag 1

(mit Lösungen der Aufgaben)

Gabriel Schöpfer, Michael Gatt, Michael Hütter, Milan Ončák

1.	Einführung in Python (lokale Installation, Visual Studio Code, Jupyter Notebook).....	2
2.	Python Basics: Variablen, Listen, Schleifen, Funktionen und Bedingungen.....	4
3.	Grundlagen von NumPy (Arrays, Rechnen mit Vektoren und Matrizen).....	9
4.	Datenimport und -verarbeitung mit Pandas	13
5.	Visualisierung von Daten mit Matplotlib.....	16
6.	Pandas: Etwas fortgeschritten.....	23
7.	NumPy: Etwas fortgeschritten	27

Hilfreiche Dokumentationen

NumPy: <https://numpy.org/doc/stable/reference/index.html>

Matplotlib: Beispielgalerie <https://matplotlib.org/stable/gallery/index>, Cheatsheets
<https://matplotlib.org/cheatsheets/>

Pandas: <https://pandas.pydata.org/docs/reference/frame.html>

SciPy: <https://docs.scipy.org/doc/scipy/reference/index.html>

sowie Google + Stackoverflow 😊

1. Einführung in Python (lokale Installation, Visual Studio Code, Jupyter Notebook)

- a) Installation von Visual Studio Code (VS Code)
<https://code.visualstudio.com/download>

Einfach je nach Betriebssystem, auf den großen Button klicken, herunterladen, und durch den Installationswizard durchklicken. Die voreingestellten Werte können alle übernommen werden.

- b) Installation von Python

(Wenn ihr bereits eine funktionierende Python-Version installiert habt, könnt ihr diese auch gerne nutzen. Ihr braucht nicht die neueste Version.)

<https://www.python.org/downloads/>

Oben auf den großen gelben Button „Download Python 3.13.7“ klicken, damit Python herunterladen. Dann durch den Installationswizard klicken.

Haken setzen bei „Add python.exe to PATH“. Dann auf „Customize installation“ klicken. Alle voreingestellten Werte können übernommen werden. Im letzten Schritt den Pfad unter „Customize install location“ am besten unverändert lassen, aber den Pfad (z.B. C:\Users\c7441332\AppData\Local\Programs\Python\Python313) irgendwo zwischenspeichern, da man diesen später in VS Code angeben muss.

- c) Python in VS Code: erste Schritte

VS Code öffnen und links in der Symbolleiste auf „Extensions“ klicken, dort nach „Python“ suchen. Die Python-Erweiterung installieren. (Die richtige Erweiterung hat circa 180 Millionen Downloads.)

- d) Python in VS Code nutzen:

Klicke in der Symbolleiste links auf das Explorersymbol (sieht aus wie zwei Papierblätter, die übereinander liegen) und klicke „Open Folder“. Wähle dann einen Ordner aus. Erstelle z.B. einen neuen Ordner „Python_Vorkurs“ und wähle diesen aus. Mit Doppelklick irgendwo auf das große leere Fenster erstellst du eine neue Datei. Schreibe in dieser Datei

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

Und speichere die Datei als hello_world.py (Achtung: Die Endung „.py“ ist wichtig!)

Dann sollte VS Code automatisch erkennen, dass es sich um eine Python-Datei handelt, und rechts unten „Python“ anzeigen. Ganz rechts unten kann man dann den Python Interpreter auswählen. Dazu klicken, und im sich öffnenden Menü auf „Enter interpreter path...“ klicken. Dann den vorher zwischengespeicherten Pfad reinkopieren. (Alternativ kann man auch auf „Find“ klicken und den Ordner mit dem Python-Executable manuell suchen und dann auf „Select Interpreter“ klicken.) Die installierte Python-Version sollte damit von VS Code erkannt werden.

Nun kann die Datei hello_world.py ausgeführt werden. Dazu oben rechts auf „Run Python File“ klicken (Strg+F5). In der Kommandozeile unten sollte nun der Output

```
Hello World
```

erscheinen. Herzlichen Glückwunsch! Damit hast du einen funktionierende Python-Umgebung geschaffen!

Für Datenanalyse/Plotten sind sogenannte IPython Notebooks (Dateiendung .ipynb) oft angenehmer zum Arbeiten. Während man alle Aufgaben dieses Kurses und auch des Physik-Praktikums mit normalen „.py“-Pythondateien bearbeiten kann, zeigen wir hier auch die Möglichkeit von .ipynb, da es gewisse Vorteile mit sich bringt.

e) Nutzen von .ipynb-Dateien mit VS Code

Installiere zunächst die Erweiterung „Jupyter“ in VS Code.

Erstelle eine neue Jupyter-Notebook Datei: „File“ --> „New File“ --> „Jupyter Notebook“ und schreibe dort:

```
print("Hello World from .ipynb")
```

Speichere die Datei z.B. als „hello_world_ipynb.ipynb“.

Führe die Zelle mit „Umschalt+Enter“ aus. Dann die installierte Python-Umgebung (z.B. 3.13.7) auswählen. Dann muss bestätigt werden, dass man den ipykernel installieren will. Die Ausgabe „Hello World from ipynb“ erscheint jetzt direkt unterhalb der Zelle, und nicht wie vorher in der Kommandozeile.

Super! Damit kannst du jetzt auch iPython Notebooks in VS Code verwenden.

Anmerkung: Falls Probleme mit den iPython/Jupyter Notebooks auftreten, ist es völlig ok, alles mit „normalen“ „.py“ Dateien zu machen.

Alternative:

Während die vorgestellte Lösung unsere Empfehlung darstellt, da sie langfristig die besten Möglichkeiten bietet, kann man Python auch online verwenden:

<https://jupyter.org/try-jupyter/lab/>

Dann unter „Notebook“ auf „Python 3.13“ klicken und los geht's. Tippe einfach

```
print("Hello World from online Jupyter Notebook")
```

und führe den Code mit „Strg+Enter“ aus.

2. Python Basics: Variablen, Listen, Schleifen, Funktionen und Bedingungen

Nun wollen wir erste Rechnungen mit Python durchführen. Erstelle dazu eine neue Datei: „File“ --> „New File“ --> „Jupyter Notebook“ und benenne sie z.B. „python_basics.ipynb“. Tippe z.B.

1+1

und drücke „Strg+Enter“ um den Code in der Zelle auszuführen. Wähle den installierten Python-Kernel „z.B. Python 3.13.7“ und das Ergebnis (2) sollte direkt unter der Zelle angezeigt werden.

Mit „Shift+Enter“ wird der Code ausgeführt und direkt eine neue Zelle erzeugt.

Spiele damit am besten ein bisschen rum. Die normalen Rechenoperationen (+, -, *, /, **) sollten alle problemlos funktionieren.

Die absolute Basis von allem sind Variablen. Eine Variable definiert man über

```
a = 1
```

Um etwas auf der Kommandozeile auszugeben, verwendet man die print()-Funktion:

```
print(a)
```

Definieren wir zusätzlich die Variable „b“:

```
b = 2
```

Damit kann man dann rechnen, z.B.

```
c = a + b  
print(c)
```

In Listen kann man mehrere Informationen gesammelt speichern, z.B.

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

und dann über die Position darauf zugreifen, z.B.

```
first_entry = my_list[0]  
print(first_entry)
```

oder aber auch

```
print(my_list[2] + 5)
```

was 8 ausgibt. Hierbei ist zu beachten, dass man in Python bei 0 zu zählen beginnt.

Eine der größten Stärken von PCs ist das wiederholte Ausführen der gleichen Tätigkeit (ohne sich zu beschweren ;)). Um den PC dazu zu zwingen, eignen sich am besten sogenannten Schleifen (engl. Loops). Es gibt zwei wesentliche Arten von Schleifen in Python. Die erste ist die for-Schleife

```
1. for i in range(10):  
2.     print(f"Interaction {i}, still motivated")  
3.
```

Die Variable i läuft hier von 0 bis 9 (https://www.w3schools.com/python/ref_func_range.asp), und in jeder Iteration wird etwas geprintet. In der print-Funktion wird hier ein sogenannter f-String

verwendet. Diese sind sehr praktisch, da man durch {} eine direkte Ersetzung im String erreichen kann.

Die zweite Schleife ist eine while-Schleife.

```
1. i = 0
2. while i < 10:
3.     print(f"Interaction {i}, still motivated")
4.     i = i + 1
5.
```

Hier initialisieren wir zunächst die Variable *i* mit dem Wert 0. Solange die Bedingung *i* < 10 erfüllt ist, wird das print-Statement ausgeführt. Am Ende der Iteration wird der Wert der Variable *i* um 1 erhöht, sodass *i* < 10 irgendwann nicht mehr gilt, und die Schleife abbricht.

Aufgabe:

1) Nutze eine for-Schleife, um alle Zahlen von 1 bis 100 aufzusummieren.

Lösung:

```
1. sum = 0
2. for i in range(1, 101):
3.     sum += i
4. print(f"sum = {sum}")
5.
```

2) Wiederhole das gleiche mit einer while-Schleife

Lösung:

```
1. sum = 0
2. i = 1
3. while i <= 100:
4.     sum += i
5.     i += 1
6. print(f"sum = {sum}")
7.
```

Ein anderer Grundbaustein von Python sind Funktionen. Diese definiert man folgendermaßen:

```
1. def f(x):
2.     return x**2
3.
```

Aufgerufen wird die Funktion dann z.B. als *f(2)*. Um das Ergebnis zu printen schreiben wir also:

```
1. print(f(x=2))
2.
```

Aufgabe:

3) Definiere die Funktion $f(x) = x^3 + 7$ in Python und Werte sie mithilfe eines Loops für x zwischen 0 und 10 aus. Schreibe das Ergebnis als Output.

Lösung:

```
1. def f(x):
2.     return x**3 + 7
3.
4. for x in range(10):
5.     print(f(x))
6.
```

Man kann in Python auch Funktionen definieren, die nicht unbedingt einer klassischen mathematischen Funktion entsprechen. Die folgende Funktion gibt das dritte Element einer übergebenen Liste zurück, in diesem Fall „huhn“.

```
def get_third_entry(some_list):
    return some_list[2]

tiere = ["hase", "igel", "huhn", "ameise"]

print(get_third_entry(tiere))
```

Man kann auch etwas komplexere Funktionen definieren z.B.:

```
def count_total_number_of_letters(list_with_words):
    num_letters = 0
    for word in list_with_words:
        for letter in word:
            num_letters += 1
    return num_letters
```

Aufgabe:

4) Was macht die Funktion `count_total_number_of_letters()`?

Lösung:

Sie zählt die gesamte Anzahl an Buchstaben aller Wörter, die in der übergebenen Liste sind.

Aufgabe:

5) Was gibt folgender Code-Abschnitt zurück?

```
tiere = ["hase", "igel", "huhn", "ameise"]
print(count_total_number_of_letters(tiere))
```

Lösung:

18

Aufgabe:

6) Schreibe eine Funktion, die eine Liste an Wörtern (strings) bekommt, und eine Liste retourniert, die genauso lang ist wie die übergebene Liste, und jeder Eintrag der Liste entspricht der Anzahl an Buchstaben des Worts an dieser Stelle, z.B.

```
tiere = ["hase", "igel", "huhn", "ameise"]
```

```
print(count_letters(tiere))
```

Dies soll folgende Liste zurückgeben:

[4, 4, 4, 6]

Hinweis: Dazu sind folgende zwei Python-Befehle notwendig. Mit

```
some_new_list = []
```

kann man eine leere Liste definieren.

Mit `append()` kann man einer Liste ein weiteres Objekt hinzufügen, z.B.

```
some_list.append(5)
```

fügt der Liste `some_list` an ihrem Ende die Zahl 5 hinzu.

Lösung:

```
def count_letters(list_with_words):
    num_letters_list = []
    for word in list_with_words:
        num_letters_this_word = 0
        for letter in word:
            num_letters_this_word += 1
        num_letters_list.append(num_letters_this_word)
    return num_letters_list

print(count_letters(tiere))
```

Sehr oft will man einen Teil des Codes nur unter einer bestimmten Bedingung ausführen. Das geht z.B. so:

```
def is_large_word(word):  
    num_letters = len(word)  
    print(f"num_letters = {num_letters}")  
    if num_letters > 10:  
        return True  
    else:  
        return False  
  
print(is_large_word("Elefant"))  
print(is_large_word("Donaudampfschiff"))  
print(is_large_word("Obstkuchen"))  
print(is_large_word("100"))
```

Aufgabe:

7) Was macht die Funktion „is_large_word()“? Was geben die vier print()-Kommandos aus? Was passiert, wenn man anstatt des strings „100“ den Integer 100 übergibt?

Lösung:

An die Funktion is_large_word() muss ein string übergeben werden. Die Funktion gibt „True“ zurück, wenn der string aus mehr als 10 Buchstaben besteht, ansonsten wird „False“ zurückgegeben. Wenn man den Integer 100 übergibt, bekommt man eine Fehlermeldung.

3. Grundlagen von NumPy (Arrays, Rechnen mit Vektoren und Matrizen)

a) Installation von NumPy

Anstatt mit Skalaren wollen wir nun mit Vektoren und Matrizen rechnen. Dazu ist „NumPy“ die absolute Standardbibliothek in Python. Tippe dazu in einer neuen Datei numpy_playground.ipynb

```
import numpy as np
```

Das wird einen Fehler erzeugen „No module named 'numpy'“. Bislang haben wir zwar Python installiert, alle Bibliotheken (Libraries bzw. sogenannte „Module“) müssen aber zusätzlich installiert werden. Um numpy zu installieren, tippe unten in der Kommandozeile (öffnet sich mit „Strg+ä“ oder „Strg+ö“)

```
pip install numpy
```

Führe den Code von oben jetzt nochmal aus:

```
import numpy as np
```

und es sollte keine Fehler mehr geben.

Hinweis: bei der Installation über pip gibt es manchmal Probleme mit den verschiedenen Python-Versionen, wenn mehrere Python-Versionen installiert sind. Manchmal nützt es dann „python -m pip install numpy“ anstatt „pip install numpy“ auszuführen. Wenn das auch nicht funktioniert, kann man die gewünschte Pythonversion im Installationsverzeichnis umbenennen, z.B. „python313.exe“ anstatt „python.exe“, und dann kann man mit „python313 -m pip install numpy“ numpy installieren, und zwar in die Python3.13-Version. Wenn der interne VS Code Terminal nicht funktioniert vielleicht die Standard Windows PowerShell.

b) Vektoren

Jetzt wollen wir einen Vektor definieren:

```
u = np.array([1,2,3])
```

Zur Erklärung: [1,2,3] ist eine Liste und diese Liste übergeben wir als Argument an die Funktion np.array(). u ist dann eine 1D-Vektor mit den Einträgen der Liste, in diesem Fall 1,2 und 3.

Mit diesem Vektor können wir rechnen, wie wir es gewohnt sind, z.B.

```
v = 10*u  
print(v)
```

Wie wir sehen, macht Python das, was wir erwarten.

Oder auch

```
w = u/2  
print(w)
```

Lass uns jetzt einen weiteren Vektor definieren:

```
a = np.array([4,5,6])
```

Bei der Multiplikation müssen wir ein bisschen aufpassen. Was passiert, wenn wir

```
a*u
```

rechnen? Man sieht, Python interpretiert das zeilenweise, also quasi

$$\mathbf{a} \cdot \mathbf{u} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 4 \cdot 1 \\ 5 \cdot 2 \\ 6 \cdot 3 \end{pmatrix} = \begin{pmatrix} 4 \\ 10 \\ 18 \end{pmatrix}$$

Das ist nicht das, was man aus mathematischer Sicht erwarten würde! Um beispielweise ein Skalarprodukt zu berechnen, greifen wir am besten auf den „@“-Operator zurück. Der macht in der Regel das, was man aus mathematischer Sicht erwarten würde, z.B.

```
a @ u
```

Und, magischerweise, interpretiert Python das als Skalarprodukt und es kommt das Ergebnis 32 heraus. Als Ergebnis wird `np.int64(32)` geschrieben. Das bedeutet lediglich, dass der Datentyp `np.int64`, also ein numpy-Integer ist. Das braucht uns nicht weiter zu interessieren. Wir können damit ganz normal rechnen, z.B.

```
a @ u + 8
```

und es kommt 40 heraus. Anstatt des magischen „@“-Operators, kann man auch

```
np.dot(a, u)
```

verwenden. Das ist ein bisschen expliziter („dot“ meint das „dot product“, also Skalarprodukt), aber sieht in manchen Augen wahrscheinlich weniger schön aus. Was man verwendet ist absolut Geschmackssache.

Das coole ist, der „@“-Operator funktioniert genauso gut für Matrix-Vektor oder Matrix-Matrix Multiplikationen, z.B.

```
1. A = np.array([[2, 0, 0],
                [0, 1, 0],
                [0, 0, 1]])
2. A @ a
3.
```

also

$$\mathbf{A} \cdot \mathbf{a} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 8 \\ 5 \\ 6 \end{pmatrix}.$$

oder

```
1. B = np.array([[5, 0, 0],
                [0, 1, 0],
                [0, 0, 1]])
2. A @ B
3.
```

also

$$\mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 5 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 10 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Die Alternativen hier wären

```
np.matmul(A, a)
```

bzw.

```
np.matmul(A, B)
```

Aufgabe:

8) Definiere die Vektoren und Matrizen

$$\mathbf{a}_1 = \begin{pmatrix} 20 \\ 55 \\ 6.5 \end{pmatrix}, \quad \mathbf{a}_2 = \begin{pmatrix} 27.5 \\ 3 \\ 8 \end{pmatrix}, \quad \mathbf{A}_1 = \begin{pmatrix} 5 & 7 & 9 \\ 10 & 2 & 0 \\ 1 & 0 & 1 \end{pmatrix}, \quad \mathbf{A}_2 = \begin{pmatrix} 7.5 & 9 & 33 \\ 0 & 4 & 5 \\ 2 & 0 & 6 \end{pmatrix}$$

und berechne mithilfe von Numpy

- i) $\mathbf{a}_1 + \mathbf{a}_2$
- ii) $\mathbf{a}_1 \cdot \mathbf{a}_2$
- iii) $\mathbf{A}_1 + \mathbf{A}_2$
- iv) $\mathbf{A}_1 \cdot \mathbf{A}_2$
- v) $\mathbf{A}_1 \cdot \mathbf{a}_2$

Lösung:

- i) $\mathbf{a}_1 + \mathbf{a}_2 = \begin{pmatrix} 47.5 \\ 58 \\ 14.5 \end{pmatrix}$
- ii) $\mathbf{a}_1 \cdot \mathbf{a}_2 = 767$
- iii) $\mathbf{A}_1 + \mathbf{A}_2 = \begin{pmatrix} 12.5 & 16 & 42 \\ 10 & 6 & 5 \\ 3 & 0 & 7 \end{pmatrix}$
- iv) $\mathbf{A}_1 \cdot \mathbf{A}_2 = \begin{pmatrix} 55.5 & 73 & 254 \\ 75 & 98 & 340 \\ 9.5 & 9 & 39 \end{pmatrix}$
- v) $\mathbf{A}_1 \cdot \mathbf{a}_2 = \begin{pmatrix} 230.5 \\ 281 \\ 35.5 \end{pmatrix}$

c) Zufallszahlen

Oft benötigt man Zufallszahlen. Auch das geht problemlos mit numpy. Z.B

```
randnum = np.random.rand()  
print(randnum)
```

Dadurch wird der Variablen randnum eine Zufallszahl im Intervall [0,1] zugewiesen.

Aufgabe:

9) Führe den Code oben mehrmals aus und beobachte, was passiert.

Lösung:

Jedes Mal wird eine andere Zufallszahl im Intervall [0,1] erzeugt.

Wenn man ein Array von Zufallszahlen zwischen 0 und 1 benötigt, geht das so:

```
randnums = np.random.rand(3, 2)
```

```
print(randnums)
```

Dies erzeugt ein numpy Array von der Größe 3x2 mit Zufallszahlen im Intervall [0,1].

Aufgabe:

10) Schreibe eine Funktion. Diese soll ein 1D numpy array an Zufallszahlen als Argument bekommen, und wiederum ein numpy array dieser Zahlen zurückgeben, allerdings nur die Zahlen, die größer sind als 0,5. Rufe die Funktion dann mit einem 1D numpy array mit 10 Zufallszahlen zwischen 0 und 1 auf.

Lösung:

```
def filter_numbers(array_with_numbers):
    return_list = []
    for number in array_with_numbers:
        if number > 0.5:
            return_list.append(number)
    return np.array(return_list)

randnums_1d = np.random.rand(10)
print(randnums_1d)
filtered_numbers = filter_numbers(randnums_1d)
print(filtered_numbers)
```

4. Datenimport und -verarbeitung mit Pandas

Als nächstes wollen wir uns damit beschäftigen, wie wir Daten aus einer Datei importieren können. Dazu erstellen wir eine neue Datei, z.B. pandas_playground.ipynb

Um pandas zu nutzen, müssen wir *in der Kommandozeile* zunächst wieder

```
1. pip install pandas  
2.
```

ausführen, um pandas zu installieren.

Nun können wir in pandas_playground.ipynb

```
1. import pandas as pd  
2.
```

schreiben, um das pandas-Modul zu importieren. Nur so können wir es auch verwenden. Jetzt wollen wir eine Datei mithilfe von pandas importieren. Lade dazu die Datei „spectrum.csv“ aus Olat herunter und speichere sie in dem Ordner, in dem auch deine pandas_playground.ipynb-Datei ist. Der erste Versuch wäre z.B.

```
1. df = pd.read_csv("spectrum.csv")  
2. print(df)
```

Wie man sieht, wird die Datei importiert. Allerdings hat der sogenannte pandas-DataFrame nur 1 Spalte ([8001 rows x 1 columns]). Offensichtlich erkennt pandas nicht, dass wir eigentlich zwei Spalten haben. Es erwartet nämlich, dass die Daten mit Kommas separiert sind. In unserem Fall sind die Daten mit Leerzeichen separiert, deshalb müssen wir das zusätzliche Argument sep=r"\s+" angeben. Damit wird alles als Spalten-Separator gewertet, was irgendwie wie ein space ist (Leerzeichen, zwei Leerzeichen, Tab, ...), also:

```
1. df = pd.read_csv("spectrum.csv", sep=r"\s+")  
2. print(df)  
3.
```

Hinweis: das „r“ vor r"\s+" zeigt an, dass es sich um einen raw-string handelt. Der Backslash wird also nicht als escape-Code interpretiert. „s+“ bedeutet „one or many whitespaces“.

Indem wir auf „Open ‘df’ in Data Wrangler“ klicken, können wir uns den DataFrame recht angenehm anschauen und überprüfen, ob die Daten richtig importiert wurden. (Hinweis: Die „Data Wrangler“ Erweiterung muss eventuell zunächst noch in VS Code installiert werden.)

```
1. print(df)  
2.
```

gibt auch eine gute erste Übersicht. Auch

```
1. df.info()  
2.
```

ist oft hilfreich.

Je nachdem wie die Daten vorliegen, muss der pd.read_csv() Aufruf leicht modifiziert werden. Ein häufiger Fall ist beispielsweise, dass es keine Header-Zeile gibt, die Datei also direkt mit den Daten beginnt. Die Datei „spectrum_no_header.csv“ stellt so ein Beispiel dar. Versuchen wir zunächst, die Datei wie vorher einzulesen:

```
1. df2 = pd.read_csv("spectrum_no_header.csv", sep=r"\s+")  
2. print(df2)  
3.
```

Wie wir sehen, haben wir jetzt nur noch 8000 Zeilen (anstatt 8001). Die erste Zeile mit dem Datenpaar (0.0, 5.35758639e-34) wurde jetzt offensichtlich als Spaltenname interpretiert. Das sehen wir z.B. auch, wenn wir

```
1. df.info()  
2.
```

ausführen. Das wollen wir nicht. Stattdessen wollen wir pandas sagen, dass es direkt mit Daten losgeht. Das geht mit dem Argument header=None, also

```
1. df3 = pd.read_csv("spectrum_no_header.csv", sep=r"\s+", header=None)  
2. print(df3)  
3.
```

Jetzt haben wir wieder 8001 Zeilen, und das erste Datenpaar ist (0.0, 5.35758639e-34). Die Spaltennamen sind automatisch als „0“ und „1“ festgelegt worden. Damit wir später noch wissen was die Daten bedeuten, ist es oft hilfreich, aussagekräftige Spaltennamen zu vergeben, z.B.

```
1. df4 = pd.read_csv("spectrum_no_header.csv", sep=r"\s+", header=None, names=["wavenumber",  
"intensity"])  
2. print(df4)  
3.
```

Dies sollte das erwartete Ergebnis bringen.

Aufgabe:

11) Importiere die Datei „spectrum_no_header.csv“ und speichere sie als pandas DataFrame in der Variable df5. Benenne die erste Spalte „Wellenzahl“ und die zweite Spalte „Intensität“.

Lösung:

```
1. df5 = pd.read_csv("spectrum_no_header.csv", sep=r"\s+", header=None, names=["Wellenzahl",  
"Intensität"])  
2. print(df5)  
3.
```

Manchmal hat man Datensätze, von welchen man nur einen Teil importieren will. Dann ist der Parameter „usecols“ in der Funktion pd.read_csv() hilfreich, siehe z.B.

https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html.

Aufgabe:

12) Importiere die ersten zwei Spalten der Datei „spectrum_no_header_three_columns.csv“ als pandas DataFrame.

Lösung:

```
df6 = pd.read_csv("spectrum_no_header_three_columns.csv", sep=r"\s+", header=None,  
names=["Wellenzahl", "Intensität"], usecols=[0,1])  
  
print(df6)  
  
print(df6.info())
```

Bevor wir zu fortgeschrittenen Datenbearbeitung mit pandas kommen, wollen wir unser Ergebnis zunächst mithilfe von Matplotlib visualisieren.

5. Visualisierung von Daten mit Matplotlib

Zum Plotten brauchen wir zunächst wieder ein bestimmtes Python-Modul. Das *mit Abstand* verbreitetste ist *Matplotlib*. Also neue .ipynb Datei erstellen (z.B. matplotlib_playground.ipynb), über die Kommandozeile matplotlib installieren (pip install matplotlib), und in der neuen .ipynb-Datei matplotlib importieren (Hinweis: manchmal verschwindet unten im Fenster der Terminal. Dieser kann mit „Strg+ä“ oder „Strg+ö“ wieder geöffnet werden.)

```
1. import matplotlib.pyplot as plt  
2. import pandas as pd
```

Importieren wir zunächst wieder den Datensatz

```
1. df = pd.read_csv("spectrum_no_header.csv", sep=r"\s+", header=None, names=["Wellenzahl",  
"Intensität"])  
2. print(df)  
3.
```

Jetzt wollen wir die Daten plotten, nämlich die Intensität als Funktion der Wellenzahl.

```
1. plt.plot(df["Wellenzahl"], df["Intensität"])  
2. plt.show()  
3.
```

Hinweis: der zusätzlich Befehl „plt.show()“ ist nur in normalen .py-Dateien notwendig. In Jupyter Notebooks mit .ipynb-Dateiendung ist dieser Zusatz nicht notwendig. Der Plot wird sowieso gezeigt.

Damit haben wir unseren ersten Plot erstellt. Lass uns nun den Plot ein bisschen verschönern.

```
1. plt.plot(df["Wellenzahl"], df["Intensität"], color="red")  
2. plt.xlabel(r"Wellenzahl / cm$^{-1}$")  
3. plt.ylabel("Intensität / a.u.")  
4. plt.title(r"Schwingungsspektrum von C$_{120}^{}$")  
5. plt.xlim((0,2000))  
6.
```

Die Kombination von „r“ vor dem String und \$...\$ im String lässt python den Teil als Latex-Code interpretieren, sodass man dort Formeln etc. schreiben kann, so eben z.B. auch das tiefgestellte „120“ und das hochgestellte Minus.

Jetzt wollen wir ein zweites Schwingungsspektrum importieren

```
1. df2 = pd.read_csv("spectrum_no_header_2.csv", sep=r"\s+", header=None, names=["Wellenzahl",  
"Intensität"])  
2. print(df2)  
3.
```

und dieses in den gleichen Plot plotten:

```
1. plt.plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1")  
2. plt.plot(df2["Wellenzahl"], df2["Intensität"], color="blue", label="Spektrum 2")  
3. plt.xlabel(r"Wellenzahl / cm$^{-1}$")  
4. plt.ylabel("Intensität / a.u.")  
5. plt.title(r"Schwingungsspektrum von C$_{120}^{}$")  
6. plt.xlim((0,2000))  
7. plt.legend(loc="upper left")  
8.
```

Dabei haben wir noch Legendeneinträge für die beiden Datensätze erstellt mittelt label=“Spektrum XY“ und plt.legend(loc=“upper left“). Das Ergebnis ist schon mal relativ schön. Allerdings hat Spektrum 1 eine viel niedrigere Intensität als Spektrum 2. Zur besseren Sichtbarkeit, wollen wir Spektrum 1 skalieren, sodass beide Spektren gut sichtbar sind. Dazu brauchen wir wieder pandas. Direkt nachdem wir den ersten Datensatz importieren, multiplizieren wir die Intensität mit 20:

```
1. df = pd.read_csv("spectrum_no_header.csv", sep=r"\s+", header=None, names=["Wellenzahl",  
"Intensität"])
```

```

2. print(df)
3. df["Intensität"] = df["Intensität"] * 20
4.

```

Danach machen wir wieder den Plot, wie vorher:

```

1. plt.plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1 x20")
2. plt.plot(df2["Wellenzahl"], df2["Intensität"], color="blue", label="Spektrum 2")
3. plt.xlabel(r"Wellenzahl / cm$^{-1}$")
4. plt.ylabel("Intensität / a.u.")
5. plt.title(r"Schwingungsspektrum von C$_{120}^-$")
6. plt.xlim(0,2000)
7. plt.legend(loc="upper left")
8.

```

Und wir sehen, dass die Intensität angepasst wurde. Außerdem haben wir die Legende angepasst zu „Spektrum 1 x20“ um zu zeigen, dass die Intensität mit 20 multipliziert wurde.

Aufgabe:

13) Lese die beiden Datensätze spectrum3.csv und spectrum4.csv ein, plotte sie in einen Plot mit den Farben grün und orange, erzeuge aussagekräftige Achsenbeschriftungen, Legende und Titel. Passe die Intensität der Daten falls notwendig an, sodass beide Spektren gut sichtbar sind.

Lösung:

```

1. df3 = pd.read_csv("spectrum3.csv", sep=r"\s+", header=None, names=["Wellenzahl",
   "Intensität"])
2. df4 = pd.read_csv("spectrum4.csv", sep=r"\s+", header=None, names=["Wellenzahl",
   "Intensität"])
3. plt.plot(df3["Wellenzahl"], df3["Intensität"], color="green", label="Spektrum 3")
4. plt.plot(df4["Wellenzahl"], df4["Intensität"], color="orange", label="Spektrum 4")
5. plt.xlabel(r"Wellenzahl / cm$^{-1}$")
6. plt.ylabel("Intensität / a.u.")
7. plt.title(r"Schwingungsspektrum von C$_{120}^-$")
8. plt.xlim(0,2000)
9. plt.legend(loc="upper left")
10.

```

Damit haben wir die absoluten Basics von Datenimport und Plotten abgedeckt. Oft will man aber etwas kompliziertere Plots erstellen.

Manchmal will man z.B. mehrere Plots untereinander zeigen. Vor allem dann, wenn man sehr viele Daten hat, und diese in einem Plot nicht mehr übersichtlich dargestellt werden können. Dazu verwendet man den plt.subplots() Befehl. Dieser erzeugt mehrere sogenannte „Axes“ Objekte, die man mithilfe von Indexierung erreichen kann, also z.B. axs[0] oder axs[1]. fig ist dann die ganze Figure an sich, die meisten Befehle beziehen sich aber auf die Axes-Objekte.

```

1. fig, axs = plt.subplots(nrows=2, ncols=1, sharex=True)
2. axs[0].plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1 x20")
3. axs[1].plot(df2["Wellenzahl"], df2["Intensität"], color="blue", label="Spektrum 2")
4.

```

Auch hier können wir wieder die Plots mit Achsenbeschriftungen etc. verbessern. Die Befehle dafür sind etwas anders (leider), aber sehr ähnlich.

```

1. fig, axs = plt.subplots(nrows=2, ncols=1, sharex=True)
2. axs[0].plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1 x20")
3. axs[1].plot(df2["Wellenzahl"], df2["Intensität"], color="blue", label="Spektrum 2")
4. axs[1].set_xlabel(r"Wellenzahl / cm$^{-1}$")

```

```

5. axs[0].set_ylabel("Intensität / a.u.")
6. axs[1].set_ylabel("Intensität / a.u.")
7. axs[1].set_xlim(0,2000)
8. axs[0].legend()
9. axs[1].legend()
10.

```

Aufgabe:

14) Erstelle einen Plot mit den vier Spektrum als Subplots:

- a) Alle vier Plots untereinander angeordnet.
- b) Alle Plots im gleichen Feld.
- c) Die Plots in der Form 2x2 angeordnet sind.

Lösung a)

```

1. fig, axs = plt.subplots(nrows=4, ncols=1, sharex=True, figsize=(6.4, 10))
2. axs[0].plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1 x20")
3. axs[1].plot(df2["Wellenzahl"], df2["Intensität"], color="blue", label="Spektrum 2")
4. axs[2].plot(df3["Wellenzahl"], df3["Intensität"], color="green", label="Spektrum 3")
5. axs[3].plot(df4["Wellenzahl"], df4["Intensität"], color="orange", label="Spektrum 4")
6. axs[3].set_xlabel(r"Wellenzahl / cm$^{-1}$")
7. axs[3].set_xlim(0,2000)
8. axs[0].set_ylabel("Intensität / a.u.")
9. axs[1].set_ylabel("Intensität / a.u.")
10. axs[2].set_ylabel("Intensität / a.u.")
11. axs[3].set_ylabel("Intensität / a.u.")
12. axs[0].legend()
13. axs[1].legend()
14. axs[2].legend()
15. axs[3].legend()
16.

```

Lösung b)

```

1. fig, ax = plt.subplots()
2. ax.plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1 x20")
3. ax.plot(df2["Wellenzahl"], df2["Intensität"], color="blue", label="Spektrum 2")
4. ax.plot(df3["Wellenzahl"], df3["Intensität"], color="green", label="Spektrum 3")
5. ax.plot(df4["Wellenzahl"], df4["Intensität"], color="orange", label="Spektrum 4")
6. ax.set_xlabel(r"Wellenzahl / cm$^{-1}$")
7. ax.set_ylabel("Intensität / a.u.")
8. ax.set_xlim(0,2000)
9. ax.legend()
10. plt.savefig("spectra_all_four.pdf")
11.

```

Hinweis: hier haben wir den Befehlt `plt.savefig("spectra_all_four.pdf")` eingeführt, um den Plot als pdf zu speichern. Genauso kann man z.B. auch als svg oder png speichern:
`plt.savefig("spectra_all_four.svg")` oder

`plt.savefig("spectra_all_four.png")`

.pdf und .svg haben die beste Auflösung.

Lösung c)

```

1. fig, axs = plt.subplots(nrows=2, ncols=2, sharex=True, sharey=True, figsize=(12.8, 10))
2. axs[0][0].plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1 x20")

```

```

3. axs[1][0].plot(df2["Wellenzahl"], df2["Intensität"], color="blue", label="Spektrum 2")
4. axs[0][1].plot(df3["Wellenzahl"], df3["Intensität"], color="green", label="Spektrum 3")
5. axs[1][1].plot(df4["Wellenzahl"], df4["Intensität"], color="orange", label="Spektrum 4")
6. axs[1][0].set_xlabel(r"Wellenzahl / cm$^{-1}$")
7. axs[1][1].set_xlabel(r"Wellenzahl / cm$^{-1}$")
8. axs[1][0].set_xlim((0,2000))
9. axs[1][1].set_xlim((0,2000))
10. axs[0][0].set_ylabel("Intensität / a.u.")
11. axs[1][0].set_ylabel("Intensität / a.u.")
12. axs[0][0].legend()
13. axs[0][1].legend()
14. axs[1][0].legend()
15. axs[1][1].legend()
16.

```

Häufig ist es hilfreich, neben den Major Ticks auch Minor Ticks anzuzeigen. Dazu verwenden wir am besten die Klasse MultipleLocator

```

1. from matplotlib.ticker import MultipleLocator
2.

```

und können dann auf der x-Achse und auf der y-Achse die Minor Ticks hinzufügen:

```

1. ax.xaxis.set_minor_locator(MultipleLocator(50))
2. ax.yaxis.set_minor_locator(MultipleLocator(1e-18))
3.

```

MultipleLocator(50) bedeutet z.B., dass die Striche in einem Abstand von 50 platziert werden. Analog kann man auch den Major Locator anpassen. Dieser bezieht sich auf die Achsenpositionen, wo nicht nur Striche, sondern auch Zahlen dabeistehen.

Aufgabe:

15) Plotte ein Spektrum mit Zahlen bei 0, 500, 1000, 1500, 2000 auf der x-Achse und kleinen Markern (Minor Ticks) alle 100 Schritte. Auf der y-Achse sollen die Major Ticks einen Abstand von 1e-17 und die Minor Ticks einen Abstand von 2e-18 haben.

Lösung:

```

1. fig, ax = plt.subplots()
2. ax.plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1 x20")
3. ax.plot(df2["Wellenzahl"], df2["Intensität"], color="blue", label="Spektrum 2")
4. ax.plot(df3["Wellenzahl"], df3["Intensität"], color="green", label="Spektrum 3")
5. ax.plot(df4["Wellenzahl"], df4["Intensität"], color="orange", label="Spektrum 4")
6. ax.set_xlabel(r"Wellenzahl / cm$^{-1}$")
7. ax.set_ylabel("Intensität / a.u.")
8. ax.set_xlim((0,2000))
9. ax.legend()
10. ax.xaxis.set_major_locator(MultipleLocator(500))
11. ax.xaxis.set_minor_locator(MultipleLocator(100))
12. ax.yaxis.set_major_locator(MultipleLocator(1e-17))
13. ax.yaxis.set_minor_locator(MultipleLocator(2e-18))
14.

```

Manchmal will man die Schriftgröße anpassen. Während es möglich ist, die Schriftgröße für jedes einzelne Element (Legende, Achsenbeschriftung, Achsenlabels, ...) einzeln anzupassen, liefert eine globale Anpassung der Schriftgröße fast immer das gewünschte Resultat. Dies geht mittels

```
1. plt.rcParams.update({"font.size": 15})
```

Mithilfe von

```
1. print(plt.rcParams)
2.
```

erhält man eine Liste von globalen Parametern, die man ändern kann. Ein Auszug daraus ist nachfolgend vorhanden (gegeben sind die Defaultwerte):

```
1. plt.rcParams.update(
2.     {
3.         "font.size": 10,
4.         "lines.linewidth": 1.5,
5.         "xtick.major.size": 3.5,
6.         "xtick.minor.size": 2.0,
7.         "ytick.major.size": 3.5,
8.         "ytick.minor.size": 2.0,
9.         "axes.linewidth": 0.8,
10.    }
11. )
12.
```

Aufgabe:

16) Spiele mit den Werten herum und beobachte die Auswirkungen der verschiedenen Parameter.
(Hinweis: oft muss man die Zellen in VS Code Jupyter Notebooks zweimal ausführen, damit die Änderung aktiv wird.) Was verändern die verschiedenen Parameter?

Lösung:

alle Änderungen haben globale Auswirkungen

"font.size": Schriftgröße

"lines.linewidth": Liniendicke der geplotteten Daten

"xtick.major.size": Länge der Major Ticks auf der x-Achse

"xtick.minor.size": Länge der Minor Ticks auf der x-Achse

"ytick.major.size": Länge der Major Ticks auf der y-Achse

"ytick.minor.size": Länge der Minor Ticks auf der y-Achse

"axes.linewidth": Liniendicke der Achsen

Code:

```
1. fig, ax = plt.subplots()
2. ax.plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1 x20")
3. ax.plot(df2["Wellenzahl"], df2["Intensität"], color="blue", label="Spektrum 2")
4. ax.plot(df3["Wellenzahl"], df3["Intensität"], color="green", label="Spektrum 3")
5. ax.plot(df4["Wellenzahl"], df4["Intensität"], color="orange", label="Spektrum 4")
```

```

6. ax.set_xlabel(r"Wellenzahl / cm$^{-1}$")
7. ax.set_ylabel("Intensität / a.u.")
8. ax.set_xlim((0,2000))
9. ax.legend()
10. ax.xaxis.set_major_locator(MultipleLocator(500))
11. ax.xaxis.set_minor_locator(MultipleLocator(100))
12. ax.yaxis.set_major_locator(MultipleLocator(1e-17))
13. ax.yaxis.set_minor_locator(MultipleLocator(2e-18))
14. plt.rcParams.update(
15.     {
16.         "font.size": 12,
17.         "lines.linewidth": 3.5,
18.         "lines.markersize": 0.10,
19.         "xtick.major.size": 3.5,
20.         "xtick.minor.size": 2.0,
21.         "ytick.major.size": 3.5,
22.         "ytick.minor.size": 2.0,
23.         "axes.linewidth": 2,
24.     }
25. )
26.
27. plt.tight_layout()
28. plt.savefig("output.various_plotting_params.pdf")
29.

```

Um Informationen kompakt darzustellen, sind Inset-Plots oft eine elegante Möglichkeit, also ein Plot im Plot. Im Folgenden wollen wir Spektrum 1 im Hauptplot darstellen, und die Spektren 2,3 und 4 in einem kleinen Inset-Plot innerhalb des Hauptplots. Plotte dazu zunächst nur Spektrum 1.

```

1. fig, ax = plt.subplots()
2. ax.plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1")
3. ax.set_xlabel(r"Wellenzahl / cm$^{-1}$")
4. ax.set_ylabel("Intensität / a.u.")
5. ax.set_xlim((0,2000))
6. ax.legend()
7. ax.xaxis.set_major_locator(MultipleLocator(500))
8. ax.xaxis.set_minor_locator(MultipleLocator(100))
9. ax.yaxis.set_minor_locator(MultipleLocator(1e-19))
10.

```

Jetzt wollen wir eine zusätzliche Achse hinzufügen, auf die wir dann das Inset plotten. Zunächst müssen wir die entsprechende Funktion importieren

```

1. from mpl_toolkits.axes_grid1.inset_locator import inset_axes
2.

```

Dann erzeugen wir das Axes-Objekt:

```

1. inset_ax = inset_axes(parent_axes=ax, width="40%", height="40%", loc="upper left",
borderpad=2.5)
2.

```

Aufgabe:

17) Spiele mit den Parametern der Funktion `inset_axes()` herum, um zu sehen, welche Auswirkungen dies hat.

Lösung:

width: Breite des Insets, relativ zur Größe der Parent-Axes

height: Höhe des Insets, relativ zur Größe der Parent-Axes

loc: Position des Insets („upper left“, „upper right“, „lower left“, „lower right“,...)

borderpad: Abstand vom Rand (in Einheiten von Schriftgröße; also borderpad=3.5 bei Schriftgröße=10 bedeutet einen Abstand von 35 points)

18) Plotte jetzt mit inset_ax.plot(...) auf die neu erzeugte Achse die drei Spektren. Passe auch Major Ticks, Minor Ticks, Legende, Achsenlimits etc. des Inset Plots so an, dass nichts überlappt und alles gut lesbar ist.

Lösung:

```
1. from mpl_toolkits.axes_grid1.inset_locator import inset_axes
2. fig, ax = plt.subplots()
3. ax.plot(df["Wellenzahl"], df["Intensität"], color="red", label="Spektrum 1")
4. ax.set_xlabel(r"Wellenzahl / cm$^{-1}$")
5. ax.set_ylabel("Intensität / a.u.")
6. ax.set_xlim((0,2000))
7. ax.legend()
8. ax.xaxis.set_major_locator(MultipleLocator(500))
9. ax.xaxis.set_minor_locator(MultipleLocator(100))
10. ax.yaxis.set_minor_locator(MultipleLocator(1e-19))
11.
12. inset_ax = inset_axes(parent_axes=ax, width="40%", height="40%", loc="upper left",
borderpad=2.5)
13.
14. inset_ax.plot(df2["Wellenzahl"], df2["Intensität"], color="blue", label="Spektrum 2", lw=1.5)
15. inset_ax.plot(df3["Wellenzahl"], df3["Intensität"], color="green", label="Spektrum 3",
lw=1.5)
16. inset_ax.plot(df4["Wellenzahl"], df4["Intensität"], color="orange", label="Spektrum 4",
lw=1.5)
17. inset_ax.set_xlim((0,2000))
18. inset_ax.xaxis.set_minor_locator(MultipleLocator(200))
19. inset_ax.yaxis.set_minor_locator(MultipleLocator(2e-18))
20. inset_ax.legend(fontsize=9)
21.
```

Super! Damit bist du jetzt schon quasi ein Profi im Plotten mit Python. Jetzt wollen wir noch ein paar etwas fortgeschrittenere Möglichkeiten mit Pandas und NumPy besprechen.

6. Pandas: Etwas fortgeschrittenener

Wir haben bereits kennengelernt, wie man einfache Datenmanipulationen mit Pandas vollziehen kann, z.B.

```
1. df["Intensität"] = df["Intensität"] * 20  
2.
```

Im Folgenden wollen wir ein paar etwas fortgeschrittenere Möglichkeiten kennenlernen.

Aufgabe:

19) Importiere zunächst wieder die Datei „spectrum_no_header.csv“ mithilfe von Pandas:

Lösung:

```
1. import pandas as pd  
2. df = pd.read_csv("spectrum_no_header.csv", sep=r"\s+", header=None, names=["Wellenzahl",  
   "Intensität"])  
3. print(df)  
4. print(df.info())  
5.
```

Es gibt verschiedene Möglichkeiten, auf den Inhalt des DataFrames zuzugreifen. Eine Möglichkeit ist mittels „iloc“. Damit kann man auf die Werte per Index zugreifen. Die Indexierung ist 0-basiert, also der erste Index ist die 0, dann kommt 1 usw. Zum Beispiel

```
1. df.iloc[0,1]  
2.
```

greift auf die 0te Zeile und die 1te Spalte zu (bzw. in deutscher Sprache würde man sagen, erste Zeile und zweite Spalte). Die erste Zahl bezieht sich also auf die Zeile, die zweite auf die Spalte. Es gilt also iloc[Zeilenindex, Spaltenindex] wobei die beiden Indices 0-basiert sind.

Aufgabe:

20) Welcher Wert wird in obigem Beispiel zurückgegeben und welche Bedeutung hat dieser?

Lösung:

5.35758639e-34, Bedeutung: es ist die Intensität bei der Wellenzahl 0.0

Aufgabe:

21) Finde durch Verwenden von iloc den Wert der Intensität in der 16ten Zeilen (index 16)

Lösung:

```
1. df.iloc[16, 1]  
2.
```

1.34636122766e-31

Durch sogenanntes Slicing kann man mehrere Werte auf einmal erreichen, z.B.:

```
1. df.iloc[1:3,0]  
2.
```

liefert die Zeilen mit den Indices 1,2 („1:3“ bedeutet alles von Index 1 (eingeschlossen) bis Index 3 (ausgeschlossen)) und die Spalte mit dem Index 0.

Aufgabe:

22) Mache slicing auf den DataFrame, sodass du alle Spalten bekommen, aber nur die Zeilen mit den Indices zwischen 50 und 60 (beides eingeschlossen).

Hinweis: Lässt man die Spaltenindexierung weg, bedeutet das immer „alle Spalten“.

Lösung:

```
1. df.iloc[50:61, :]  
2.
```

Alternativ zum indexbasierten Zugriff mittels „iloc“ kann man mittels „loc“ auch labelbasiert zugreifen. Hier gilt also loc[Zeilenlabel, Spaltenlabel]. Auch hier gilt wieder, lässt man die Spaltenindexierung weg, bedeutet das „alle Spalten“. Auf alle Spalten der Zeile mit Index 0 kann man also in diesem Fall zugreifen mit

```
1. df.loc[0]  
2.
```

oder aber auch mit

```
1. df.loc[0,:]  
2.
```

Um auf die Zeile mit Index 0 zuzugreifen, aber nur die Intensität zu erhalten, gibt man jetzt nicht den Index der Intensität an wie bei iloc (Index 1), sondern direkt das Label „Intensität“, also:

```
1. df.loc[0, "Intensität"]  
2.
```

Aufgabe:

23) Greife mithilfe von „loc“ auf alle Intensitäten zu, die Zeilenindices zwischen 0 (inklusive) und 2000 (inklusive) haben. Hinweis: Beim Slicing mit loc ist Start und Ende inklusive (anders als bei iloc oder sonst fast immer in Python!!!)

Lösung:

```
1. df.loc[0:2000, "Intensität"]  
2.
```

Als dritte Möglichkeit hat man den direkten Zugriff auf Spalten, z.B.

```
1. df["Wellenzahl"]  
2.
```

gibt alle Wellenzahlen zurück.

Aufgabe:

24) Greife auf alle Intensitäten zu. a) mit iloc, b) mit loc und c) mit direktem Spaltenzugriff.

Lösung:

1. `df.iloc[:,1]`
2. `df.loc[:, "Intensität"]`
3. `df["Intensität"]`
- 4.

Eine andere oft sehr hilfreiche Möglichkeit ist der Zugriff mittels Bedingung. Wollen wir zum Beispiel nur die Zeilen des DataFrames haben, bei denen die Intensität größer als 1e-20 ist, können wir folgendermaßen vorgehen:

1. `df.loc[df["Intensität"] > 1e-20, :]`
- 2.

Das sollte 988 Zeilen x 2 Spalten zurückgeben.

Erklärung:

Welche Zeilen wollen wir? --> Die Zeilen für die die Bedingung `df["Intensität"] > 1e-20` wahr ist

Welche Spalten wollen wir? --> alle Spalten (":")

Hinweis: wie wir vorher gelernt haben, funktioniert folgendes genauso gut:

1. `df.loc[df["Intensität"] > 1e-20]`
- 2.

Aufgabe:

25) Greife mithilfe bedingungsbasierten Zugriffs auf den Teil des DataFrames zu, bei dem die Wellenzahl kleiner gleich 2000.0 ist.

Lösung:

1. `df.loc[df["Wellenzahl"] <= 2000]`
- 2.

Bedingungen kann man innerhalb einer DataFrames mit „&“ verknüpfen. Damit können wir auf den Teil des DataFrames zugreifen, bei dem die Wellenzahl zwischen 1000 (eingeschlossen) und 2000 (eingeschlossen) ist:

1. `df.loc[(df["Wellenzahl"] >= 1000) & (df["Wellenzahl"] <= 2000)]`
- 2.

Aufgabe:

26) Speichere den Rückgabewert des obigen Slices in einer Variablen „df_slice“ und plotte diesen Teil des DataFrames.

Lösung:

```
1. import matplotlib.pyplot as plt
2. df_slice = df.loc[(df["Wellenzahl"] >= 1000) & (df["Wellenzahl"] <= 2000)]
3. plt.plot(df_slice["Wellenzahl"], df_slice["Intensität"])
4. plt.show()
5.
```

7. NumPy: Etwas fortgeschrittenes

Vor allem im naturwissenschaftlichen Bereich ist NumPy sehr weit verbreitet. Das liegt daran, dass lineare Algebra Operationen usw. zum großen Teil in Python auf NumPy basiert. Daher wollen wir jetzt noch zeigen, wie man aus einem pandas DataFrame eine NumPy Array erzeugt, und wie slicing auf NumPy Arrays funktioniert. Spoiler: Es ist genauso wie mit pandas „iloc“.

Lass uns dazu eine neue Datei „numpy_advanced_playground.ipynb“ erstellen. Importieren wir zunächst numpy, pandas und lesen den Datensatz ein:

```
1. import numpy as np
2. import pandas as pd
3. df = pd.read_csv("spectrum_no_header.csv", sep=r"\s+", header=None, names=["Wellenzahl",
   "Intensität"])
4. print(df)
5. print(df.info())
6.
```

Jetzt wollen wir aus dem pandas DataFrame ein numpy Array erzeugen. Das ist einfach:

```
1. array = np.array(df)
2.
```

Jetzt können wir wieder ganz normal slicen, z.B.

```
1. array[:10, :]
2.
```

gibt alle Zeilen bis ausschließlich der Zeile mit Index 10 zurück (also alle Zeilen mit Index 0 bis 9), und alle Spalten.

Aufgabe:

27) Verwende numpy slicing um alle Intensitäten der Zeilen zu bekommen, die Indices zwischen 10 (einschließlich) und 20 (einschließlich) haben.

Lösung:

```
1. array[10:21, 1]
2.
```

Aufgabe:

28) Verwende numpy slicing, um alle Intensitäten zu bekommen, die größer als 1e-20 sind.

Hinweis: mithilfe von len(some_array) kann man die Länge eines numpy arrays bestimmen. Das Ergebnisarray sollte eine Länge von 988 haben.

Lösung:

```
1. array[array[:,1] > 1e-20, 1]
2. len(array[array[:,1] > 1e-20, 1])
3.
```

Wer fertig ist, kann mit Aufgaben aus dem Kurs „Programming, Data Analysis, and Deep Learning in Python“ von Prof. Jörg Müller von der Universität Bayreuth weitermachen.

Übersicht der Aufgaben

Im Folgenden ist eine Übersicht über die Aufgaben dargestellt. Die von uns besonders empfohlenen Aufgaben sind fett geschrieben.

Blatt	Aufgabe	Schwierigkeit (1-5)	Relevanz (1-5)	Nähe zum behandelten Stoff (1-5)	Empfehlung (1-5)
1	1	1	2	1	2
1	2	2	2	2	3
1	3	2	2	2	3
1	4	2	4	4	5
2	5	3	2	3	3
2	6	4	2	2	2
2	7	2	4	4	5
2	8	3	3	2	3
3	9	1	5	4	5
3	10	3	4	4	4
3	11	2	3	2	3
3	12	5	3	3	3
5	16	3	5	5	5
5	17	3	5	5	4
5	18	4	2	2	2
6	19	4	4	4	4
6	20	5	2	2	2
6	21	3	5	5	4
6	22	4	4	4	4
7	23	5	4	4	3
8	26	5	4	3	3
8	27	5	4	3	3
8	28	5	4	3	3