vorlesung_1

November 7, 2024

[16]:
```
from IPython.core.display import display, HTML

display(HTML('''
<style>
    /* Globale Stildefinitionen für RISE */
     /* Schriftgröße für den Code in Codezellen erhöhen */
    .reveal .input_area p {
        font-size: 3.5em !important; /* Ändert die Schriftgröße für den Code */
    }
    .reveal .output_area {
        font-size: 3.5em !important; /* Ändert die Schriftgröße für den Output␣
 ↪*/
    }
    /* Optional: Padding und Margins anpassen */
    .reveal input_area, .reveal output_area {
        padding: 10px; /* Fügt mehr Innenabstand hinzu */
        margin: 10px; /* Fügt mehr Außenabstand hinzu */
    }
    .reveal p {
    font-size: 1.5em; /* Ändert die Schriftgröße für Fließtext */
}
 /* Globale Stildefinitionen für RISE */
    .reveal table {
        width: 100%; /* Ändert die Breite der Tabelle auf 100% */
        font-size: 2em !important; /* Ändert die Schriftgröße innerhalb der␣
 ↪Tabelle */
    }
    .reveal th, .reveal td {
        padding: 10px; /* Fügt Innenabstand zu den Zellen hinzu */
        text-align: left; /* Ausrichtung des Textes in den Zellen */
    }
</style>
'''))
```

```
/tmp/ipykernel_79532/189057892.py:1: DeprecationWarning: Importing display from
IPython.core.display is deprecated since IPython 7.14, please import from
IPython.display
  from IPython.core.display import display, HTML
```

```
<IPython.core.display.HTML object>
```

1 Einführung in Datenauswertung mit Python

Vorlesung Theoretische Chemie (WiSe 2024/2025)

2 Motivation

3 Motivation: Automatische Visualisierung

3.0.1 Beispiel: IR-Spektrum

3.0.2 Vorteile des Programmierens für Abbildungen

- **Reproduzierbar:** Code bleibt bestehen - der Aufwand weitere Abbildungen im selben Stil zu erzeugen beträgt praktisch 0.
- **Wiederverwendbar:** Kleine Änderung lassen sich leicht und schnell durch Anpassungen an bestehendem Code durchführen - Copy und Paste funktioniert oft.
- **Flexibel:** Grafiken können beliebig verändert und angepasst werden. Das beinhaltet das Erstellen von mehreren Grafiken, das Hinzufügen weiterer Objekte (Pfeile, Linien, Kreise, …) und viel mehr.
- **Hohe Qualität:** Konventionen wie Schrifgröße, Achsenbeschrifung etc. können leicht automatisiert werden und ermöglichen es schnell sehr hochwertige Grafiken zu erstellen.

4 Motivation: Automatische Datenauswertung

4.0.1 Auswertung von vielen, ähnlichen Daten: Beispiel IR-Spektrum

4.0.2 Vorteile des Programmierens für Datenauswertung

- **Reproduzierbar:** Code bleibt bestehen - die Analyse bleibt nachvollziehbar und wiederholbar, auch wenn die Daten sich ändern.
- **Automatisierung:** Die selbe Analyse lässt sich mit nahezu beliebig vielen Daten durchführen - ohne nennenswerten zusätzlichen Arbeitsaufwand.
- **Auswertung:** Die Datenauswertung kann beliebig komplex sein - eine “general purpose” Programmiersprache kann so ziemlich jede Art von Algorithmus ausführen
- **Zeitaufwand:** Nach der Implementierung laufen Skripte gewöhnlich in **Sekunden** - selbst wenn sie hunderte von Dateien auswerten.

5 Motivation: Animationen

Animationen können sehr hilfreich sein um komplexe Prozesse abzubilden. Das Programmieren bietet den Vorteil das sowohl die Daten als auch die Animation selber in einem Schritt erzeugt werden können.

Diffusion Monte Carlo Simulation eines harmonischen Oszillators

Simulation der Bewegung der Erde um die Sonne

6 Motivation: Physikalisches Verständnis von Vorgängen

Physik und Chemie sind experimentelle Wissenschaften - oft ist es einfacher zu sehen was passiert als einfach nur eine Gleichung anzuschauen. Durch **Simulationen** kann man oft intuitiver verstehen was passiert.

Atomorbital des Wasserstoff Atoms

Quantenmechanisches Doppelmuldenpotential

7 Beispiel: Quantenmechanischer Tunneleffekt

7.0.1 1D - Doppelmuldenpotential:

Hamiltonian: $\hat{H} = \frac{\hat{p}^2}{2} + x^4 - x^2 - 0.05 * x$ - **Klassische Zeitwentwicklung:** Teilchen fällt in eine Mulde wenn die kinetische Energie nicht reicht um die Barriere zu überwinden - **Quantenmechanische Zeitentwicklung:** Das Teilchen kann tunneln und hat in beiden Mulden eine nicht verschwindende Wahrscheinlichkeit

8 Warum Python?

8.0.1 Ranking der beliebtesten Programmiersprachen

Quelle: Hier

- **Vielseitigkeit:** Python kann für eine Vielzahl von Anwendungen verwendet werden, darunter Webentwicklung, Datenanalyse, maschinelles Lernen, wissenschaftliche Berechnungen, Systemskripting, Automatisierung und vieles mehr.
- **Wir beschränken uns auf eine spezielle Anwendung (Datenverarbeitung) und nutzen dafür nur wenige, grundlegende Funktionalitäten!** Notwendigerweise werden dadurch viele Themen nur oberflächlich und/oder unvollständig behandelt.

9 Nützliche Resourcen

9.0.1 Online Resourcen

- **Google**
- **Stack Overflow**
- Python Dokumentation
- Youtube Tutorials

9.0.2 Coding Hilfen

- **Syntax Highlighting** - Findet Syntax Fehler im Code und ist Standard bei IDE's wie Pycharm
- **AI Assistenten** - ChatGPT & Co können guten Code schreiben - wenn **genau beschrieben wird was gebraucht wird**

10 Warum Python?

- **Einfach und leicht zu lesen:** Der Code lässt sich oft intuitiv verstehen, was die Sprache ideal für Einsteiger macht.
- Funktionen und Operatoren orientieren sich an englischen Begriffen
- Python verwendet Einrückungen um Codeblöcke zu kennzeichnen. Das erleichtert die Strukturierung und Lesbarkeit.

[ ]:
```python
# Ziel: Gebe die Zahl aus, wenn sie größer/gleich als 10 ist
my_number = 9
if my_number >= 10:
    print(my_number)
else:
    print("Die Zahl ist kleiner als 10!")
```

11 Warum Python?

- **Interaktiv und interpretierbar:** Python ist eine interpretierte Sprache, was bedeutet, dass der Code direkt Zeile für Zeile ausgeführt wird. Das macht es einfach, Code direkt auszuprobieren und zu testen.
- Der Code wird von einem Python Interpreter von oben nach unten gelesen und ausgeführt

Quelle: Hier

[1]:
```python
# Ziel: Gebe die Zahl aus, wenn sie größer/gleich als 10 ist
if number >= 10:
    print(number)
else:
    print("Die Zahl ist kleiner als 10!")
# Macht es Sinn die Variable hier zu deklarieren?
number = 9
# Nein.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 2
      1 # Ziel: Gebe die Zahl aus, wenn sie größer/gleich als 10 ist
----> 2 if number >= 10:
      3     print(number)
      4 else:

NameError: name 'number' is not defined
```

Wenn eine Variable nicht definiert ist **bevor** sie aufgerufen wird, produziert das einen Fehler.

12 Fehlermeldungen: Computer sind dumm, aber Fehlermeldungen sind nützlich!

Wenn der Code nicht lesbar für den Compiler (Python) ist, wird eine Fehlermeldung produziert. **Fehlermeldungen in Python sind (meistens) sehr nützlich!** Sie enthalten wichtige Informationen um den Fehler zu finden und zu verbessern: - Fehlercode (NameError, IndexError, SyntaxError, …) - Wo tritt der Fehler auf? - Um was für eine Art von Fehler handelt es sich konkret? Fehlermeldungen sind essentiell um Fehlfunktionen des Programms zu verhindern - ein Programm sollte nicht funktionieren wenn nicht eindeutig ist was es tun soll.

12.0.1 Beispiel 1: Fehler finden

[ ]:
```
# Aufgabe: Berechne die Fläche eines Kreises A = pi * r^2
pi = 3.1415          # Kreiszahl Pi
r_1 =                # Radius Kreis 1
r_2 = 4              # Radius Kreis 2
A_1 = pi * (r_1 ** 2) # Fläche Kreis 1
A_2 = pi * (r_2 ** 2) # Fläche Kreis 2
# Textausgabe
print(f"Die Fläche von Kreis 1 beträgt ", A_1)
print(f"Die Fläche von Kreis 2 beträgt , A2)
```

12.0.2 Google Colab Link

13 Warum Python?

- **Module und Bibliotheken:** Python verfügt über eine Vielzahl von Bibliotheken und Frameworks, die die Entwicklung erheblich erleichtern. Beispiele sind:
  - NumPy, Pandas, Matplotlib für Datenwissenschaft und -analyse
  - Django, Flask für Webentwicklung
  - TensorFlow, PyTorch für maschinelles Lernen
- Für komplexe Aufgaben ist es oft **einfacher und sinnvoller** nach dem passenden Paket zu suchen anstatt selbst sämtliche Funktionalitäten zu programmieren.

Quelle: Hier

14 Module und Bibliotheken: Beispiel Sinusfunktion

Definition der Sinusfunktion: $Sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - ...$

[ ]:
```
# Ziel: Berechne die Sinusfunktion
def sinus(x, n):
# Taylornäherung der Sinusfunktion
# sin(x) = x - x^3/3! + x^5/5! - ...
# x: Punkt der Sinusfunktion
# n: Anzahl an Termen der Taylornäherung
    y = x # Output
```

```
    fac = 1 # Factorial
    for i in range(1, n):
        n_term = x ** (2 * i + 1)
        fac *= 2 * i * (2 * i + 1)
        sign = (-1) ** i
        y += sign * n_term / fac
    return y

x = 0.5
n = 3 # Länge der Taylorexpansion
sin_x = sinus(x, n)
print(sin_x)
```

[ ]:
```
# Ziel: Berechne die Sinusfunktion mit NumPy
import numpy as np # Importiere das NumPy (Numerisches Python) Paket
x = 0.5
sin_x = np.sin(x)
print(sin_x)
```

`NumPy` ist eine der wichtigsten Module für Python und wird in der nächsten Vorlesung ausführlicher vorgestellt.

14.0.1 Einfachheit > Performance

Die Grundlage des Programmierens ist der Programmcode der von Entwickler*innen geschrieben wird. Guter Code folgt einfachen Prinzipien: - **Lesbarkeit** - Ordentlicher Code nutzt **Konventionen** für Variablen, Einrückungen, Kommentare - **Verständlichkeit** - Verständlicher Code ist **kommentiert** und **strukturiert** - **Konfigurierbarkeit** - Kleine Änderungen an der Aufgabe sollten zu kleinen Änderungen am Code führen - **Don't repeat yourself** - Wiederkehrende Aufgaben sollten zusammengefasst und automatisiert werden

14.0.2 Beispiel Lesbarkeit & Verständlichkeit: Fläche eines Kreises

$A = \pi * r^2$

```
<h2>Bad practice</h2>
```

[28]:
```
print(f"{3.1415 * (2 ** 2)}")
print(f"{3.1415 * (4 ** 2)}")
```

```
12.566
50.264
```

```
<h2>Good practice</h2>
```

[27]:
```
# Aufgabe: Berechne die Fläche eines Kreises A = pi * r^2
pi = 3.1415          # Kreiszahl Pi
r_1 = 2              # Radius Kreis 1
r_2 = 4              # Radius Kreis 2
A_1 = pi * (r_1 ** 2) # Fläche Kreis 1
```

```
A_2 = pi * (r_2 ** 2) # Fläche Kreis 2
# Textausgabe
print(f"Die Fläche von Kreis 1 beträgt ", A_1)
print(f"Die Fläche von Kreis 2 beträgt ", A_2)
```

```
Die Fläche von Kreis 1 beträgt  12.566
Die Fläche von Kreis 2 beträgt  50.264
```

Das Ergebnis ist zwar das selbe, aber die zweite Version ist **klarer und verständlicher**.

15 Basics: Was ist ein Programm?

Ein Programm besteht aus einer Reihe von Anweisungen, die **nacheinander** ausgeführt werden. In einem Programm werden verschiedene Objekte, z.B. Zahlen, Listen oder Text, durch Operationen manipuliert, um ein gewünschtes Ergebnis zu erhalten.

15.0.1 Beispiel: Summiere die Zahlen von 1 bis 10

[ ]:
```
# Summiere die Zahlen von 1 bis 10
n = 10
my_number = 0
for i in range(1, n + 1):
    my_number = my_number + i

print(my_number)
```

Ein Python Programm ist eine **Textdatei**, die üblicherweise mit `.py` endet und Befehle in der Python Sprache enthält. Die Textdatei wird zu einem Python Programm, wenn sie durch einen **Python Intepreter** gelesen wird - üblicherweise durch einen Befehl in der **Kommandozeile**.

```
$ python my_script.py
```

Alternativ kann ein IDE (Integrated Development Environment) wie **PyCharm** genutzt werden, die sowohl das Programmieren in der Textdatei als auch die Ausführung des Programms in einer Entwicklungsumgebung vereint. Zusätzlich bieten Entwicklungsumgebungen nützliche Hilfestellungen, z.B. durch Syntax-Highlighting und Fehlererkennungen.

16 Basics: Variablen

Variablen bilden die Grundlage eines Programms und dienen der **Datenspeicherung**. Eine Variable ist definiert durch drei Dinge:

- **Bezeichner:** Der Name der Variable, z.B. `my_number`
- **Datentyp:** Die Art der Daten die in der Variable gespeichert werden, bspw. Zahlen, Text, eine Liste von Zahlen oder ein Plot
- **Wert:** Der spezifische Wert der Variable, bspw. eine spezifische Zahl oder eine Grafik

[12]:
```
# Variablen dienen der Datenspeicherung!
my_integer = 10
```

```
my_integer - 3
print(my_integer)
```

```
10
```

Die **Bezeichner** einer Variable können frei gewählt werden, innerhalb der von Python gesetzten Regeln für Variablen: - **Python ist case-sensitive:** Groß und Kleinschreibung (`omega`, `Omega`, …) spielt bei der Bennenung von Variablen eine Rolle - **Erlaubte symbole:** `a ... z`, `0 ... 9` und `_` (**nicht:** `-`) - **Geschützte Namen:** Namen von speziellen Funktionen und Befehlen sind geschützt und können nicht überschrieben werden

```
<h3>Wichtig</h3>
<p>Variablen sollten deskriptiv sein, d.h. der Name einer Variable sollte darauf schließen las
```

17 Basics: Variablen

Es gibt 3 verschiedene Arten von Datentypen, die eine Variable mit einem einzelnen Objekt haben kann:

- **Zahlen:** Entweder Ganze Zahlen (**integers**) oder Kommazahlen (**floats**), die mit einer bestimmten Präzision gespeichert werden
- **Text:** Text wird in Python durch Anführungszeichen `""` oder `''` markiert um von Variablen unterschieden zu werden.
- **Booleans:** Bool'sche Werte können nur zwei Werte annehmen: `True` und `False`, oder alternativ `1` und `0`

Variablen dienen zunächst der Speicherung. Um den Wert einer Variable zu überprüfen, kann sie ausgegeben werden mit der `print()` Funktion, bei der alles innerhalb der Klammern ausgegeben wird. Verschiedene Daten werden dabei durch Kommas separiert.

[ ]:
```
my_integer = 10
my_float = 10.0
my_text = "zehn"
my_bool = True

print(my_integer, type(my_integer))
print(my_float, type(my_float))
print(my_text, type(my_text))
print(my_bool, type(my_float))
```

Wenn ein Wert **nicht** in einer Variable gespeichert wird, verfällt er einfach.

[8]:
```
a = 1
b = 2
# Ohne Zuweisung einer Variablen passiert nichts
a + b
print(a, b)
```

```
1 2
```

Variablen können nach Belieben überschrieben werden, wobei der ursprüngliche Wert gelöscht wird.

```
<h3>Wichtig</h3>
<p>Die <code>print()</code> Funktion ist die einfachste Möglichkeit um Variablen auszulesen, d
```

18 Basics: Funktionen

Funktionen sind der zweite grundlegende Bestandteil eines Programms. Eine Funktion ist eine festgegte Abfolge von **Operationen**, die mit einer Reihe von Variablen, den **Argumenten**, ausgeführt wird. Die Argumente werden durch **runde Klammern** `()` festgelegt, die Allgemein das Merkmal von Funktionen sind. Ein Beispiel dafür ist die `print()` Funktion, die alles innerhalb der Klammern ausgibt.

[14]:
```
a = 1
b = 2
print("Die Zahl", 1, "ist kleiner als", 2)
```

```
Die Zahl 1 ist kleiner als 2
```

In Python gibt es eine Reihe von **Built-In Funktionen**, die wesentliche Funktionalitäten zur Verfügung stellen. In der Regel ist es aber notwendig, zusätzlich eigene Funktionen zu nutzen. Der Beginn einer Definition für eine Funktion wird durch das **def Keyword** signalisiert. Die Allgemeine Syntax ist

```
def name(argument1, argument2, ...):
    #Code
    return output1, output2, ...
```

[16]:
```
def quadrat(x):
    # Bildet das Quadrat einer Zahl
    # x: Zahl
    y = x ** 2
    return y

a = 5
b = quadrat(a)
print(a, b)
```

```
5 25
```

```
<h3>Wichtig</h3>
<p>Die Anzahl der Input und Output Argumente muss exakt eingehalten werden! Eine Funktion kann
        Python-Dokumentation
    </a> zur Verfügung.</p>
```

19 Beispiel 2: Don't repeat yourself!

Eins der grundlegenden Prinzipien für guten Code ist die Vermeidung von sich wiederholenden Codeabschnitten. Funktionen sind dafür wesentliche Hilfsmittel. ### Beispiel: Konzentrationen berechnen Der folgende Code kann durch eine Funktion stark gekürzt werden, indem wiederholdende Abschnitte durch eine **Funktion** erledigt werden.

[18]:
```python
# Input Daten - Nothing to be done here
m1 = 10    # Masse Probe 1
m2 = 3     # Masse Probe 2
m3 = 12    # Masse Probe 3
m4 = 15    # Masse Probe 4

V1 = 102   # Volumen Lösung 1
V2 = 99    # Volumen Lösung 2
V3 = 109   # Volumen Lösung 3
V4 = 103   # Volumen Lösung 4

molmasse = 99   # g / mol

# Konzentrationsberechnung
def konzentration(gewicht, molmasse, volumen):
    # Berechnet die Konzentration in mol/l
    # gewicht: Stoffmenge in Gramm
    # molmasse: Molekulare Masse in g / mol
    # volumen: Lösungsmittel in ml
    """
    Berechne die Konzentration in mol / l hier
    """
# Berechnung der Stoffmengen
n1 = m1 / molmasse
n2 = m2 / molmasse
n3 = m3 / molmasse
n4 = m4 / molmasse

# Volumen in Litern
V1 = V1 * 1000
V2 = V2 * 1000
V3 = V3 * 1000
V4 = V4 * 1000

# Berechnung der Konzentrationen
c1 = n1 / V1
c2 = n2 / V2
c3 = n3 / V3
c4 = n4 / V4

# Output - Nothing to be done here!
print("Konzentration 1 beträgt", c1)
print("Konzentration 1 beträgt", c2)
print("Konzentration 1 beträgt", c3)
print("Konzentration 1 beträgt", c4)
```

```
Konzentration 1 beträgt 9.902951079421668e-07
Konzentration 1 beträgt 3.060912151821243e-07
```

```
Konzentration 1 beträgt 1.1120378092855157e-06
Konzentration 1 beträgt 1.4710208884966166e-06
```

19.0.1 Google Colab Link

20 Datentypen: Zahlen

Es gibt zwei unterschiedliche Arten von Zahlen: Ganze Zahlen (**integers**) oder Gleitkommazahlen (**floats**).

[ ]:
```
# integer (Ganzzahl)
a = 1
b = -5
c = 0
print(a, b, c)

d = 0.1 + 0.2
e = 0.3
f = d - e
print(d, e, f)
```

Floats können auch in wissenschaftlicher Notation mit einem Exponenten als `1000 = 10e3` und `0.001 = 10e-3` angegeben werden. Während **Integers** in Python eine beliebige Größe haben können sind **Floats** auf Zahlen zwischen $1.7 * 10^{-308}$ bis $1.7 * 10^{308}$ begrenzt, da jede Zahl lediglich 64 Bits speicher zur Verfügung steht. Höhere Zahlen sind nicht speicherbar.

[ ]:
```
# Overflow
a = 1e307
b = 1e308
print(a, b)
```

Floats beschreiben nur mit einer begrenzten Genauigkeit das Koninuum der reelen Zahlen, welche als **machine precision** bezeichnet wird.

[ ]:
```
a = 0.1 + 0.2
b = 0.3
print(b, f"{b:.20f}")
print(b - a)
```

```
<h3>Wichtig</h3>
<p>In Python werden Floats mit einem Punkt <code>.</code> geschrieben, nicht mit einem Komma wi
```

21 Basics: Operationen mit Zahlen

Variablen die **Integers** oder **Floats** enthalten, können mit grundlegenden mathematischen Operatoren genutzt werden.

Operation	Symbol	Beschreibung
Addition	`+`	Addiert zwei Werte
Subtraktion	`-`	Subtrahiert den zweiten Wert vom ersten
Multiplikation	`*`	Multipliziert zwei Werte
Division	`/`	Dividiert den ersten Wert durch den zweiten
Modulo (Rest)	`%`	Gibt den Rest einer Division zurück
Potenz	`**`	Potenziert den ersten Wert mit dem zweiten
Quadratwurzel	`** 0.5`	Zieht die Quadratwurzel des Wertes

[ ]:
```python
a = 2.0
b = 3

# Addition:
add = a + b
# Subtraktion:
sub = a - b
# Multiplikation
mul = a * b
# Division
div = a / b
# Modulo
mod = a % b
# Exponent
power = a ** b

print(add, sub, mul, div, mod, power)
```

Die Reihenfolge der mathematischen Operationen folgt der Standardreihenfloge mathematischer Operationen (BODMAS): **Klammern, Exponenten, Division, Multiplikation, Addition, Subtraktion** - Kurz gesagt: Punkt vor Strich

[ ]:
```python
# Klammern vor Exponenten, Exponenten vor Division
a = 9 ** 1 / 2
b = 9 ** (1 / 2)
print(a, b)
```

Klammern setzen ist sehr hilfreich um schwer zu entdeckende Fehler zu vermeiden. Sämtliche Klammern müssen allerdings geschlossen sein, da ansonsten ein Fehler produziert wird.

22 Operationen mit Zahlen

Zusätzlich zu den grundlegenden Operatoren gibt es eine Vielzahl an Funktionen, die auf Zahlen angewandt werden können.

Operation	Symbol	Beschreibung
Absolutwert	`abs()`	Gibt den Betrag einer Zahl zurück

Operation	Symbol	Beschreibung
Rundung	`round()`	Rundet eine Zahl auf eine bestimmte Dezimalstelle
Maximum	`max()`	Gibt den größten Wert aus einer Sequenz zurück
Minimum	`min()`	Gibt den kleinsten Wert aus einer Sequenz zurück

[ ]:
```
a = -2.21
b = 3

# Absolutwert
abs_val = abs(a)
# Rundung
round_val = round(a, 1)
# Maximalwert
max_val = max(a, b)
# Minimalwert
min_val = min(a, b)

# Ausgabe
print(abs_val, round_val, max_val, min_val)
```

Spezielle mathematische Funktionen, wie bspw. die Exponentialfunktion, werden in Python durch **Module** bereitgestellt.

23 Datentypen: Strings

Text, oder **Strings**, wird in Python durch Anführungszeichen `''` und `""` markiert, um von **Variablen** unterschieden zu werden.

[7]:
```
my_text = "Hello World"
print(my_text)
```

```
Hello World
```

Strings bestehen aus einzelnen Zeichen, **Characters**, die auch separat aufgerufen werden können mittels eckiger Klammern.

[9]:
```
my_text = "Hello World"
print(my_text[0], my_text[1], my_text[2], my_text[3], my_text[4])
```

```
H e l l o
```

Strings sind ein fundamental anderer Datentyp als **Floats** und **Integers** und können nur in Ausnahmefällen kombiniert werden.

[11]:
```
my_text = "10"
my_int = 2
print(my_text + my_int)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[11], line 3
      1 my_text = "10"
      2 my_int = 2
----> 3 print(my_text + my_int)

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

Strings die Zahlen enthalten müssen erst in Floats oder Integers umgewandelt werden bevor sie als solche genutzt werden können.

[15]:
```
my_text = "10.1"
my_int = 2
my_text = float(my_text)
print(my_text + my_int)
```

```
12.1
```

Strings können mit einer Vielzahl an Operationen manipuliert werden

Operation	**Beschreibung**	**Beispiel**	**Ausgabe**
Verkettung	Verbindet Strings mit `+`	`"Hallo, " + "Welt"`	`Hallo, Welt`
Länge bestimmen	Anzahl der Zeichen mit `len()`	`len("Python")`	`6`
Ersetzen	Ersetzt Substrings mit `replace()`	`"Ich liebe Python".replace("Python", "Coding")`	`Ich liebe Coding`
Aufteilen	Teilt String in eine Liste von Wörtern	`"eins zwei drei".split()`	`['eins', 'zwei', 'drei']`

Variablen können in Text mittels der `format()` Funktion oder als `f""` **String** eingefügt werden.

[ ]:
```
# Gibt eine berechnete Konzentration aus
concentration = 0.2   # mol / l
# f"" string method
print(f"Die finale Konzentration beträgt {concentration} mol / l")
# format function
print("Die finale Konzentration beträgt {} mol / l".format(concentration))
```

24 Datentypen: Bool Werte

Bool'sche Werte können nur zwei Werte annehmen: Wahr oder Falsch, bzw. in Python `True` und `False`. Sie spielen eine wichtige Rolle in der Strukturierung von Programmen, bspw. indem gewisse Operationen nur durchgeführt werden wenn ein Wert `True` ist.

Oft werden **Bool**'sche Werte durch **Vergleichsoperatoren** oder **logische Operationen** erzeugt. **Vergleichsoperatoren** testen eine Bedingung zwischen zwei einzelnen Datenobjekten, bspw. zwischen zwei Zahlen oder zwei **Strings**, und geben einen **Bool**'schen Wert zurück.

[ ]:
```
# Vergleichsoperatoren
a = 1
b = 2

# Kleiner
lower = a < b
# Größer
greater = a > b
# Gleich
eq = a == b
# Ungleich
neq = a != b
# Größer Gleich
greatereq = a >= b
# Kleiner Gleich
lowereq = a <= b

print(lower, greater, eq, neq, greatereq, lowereq)
```

```
<h3>Wichtig</h3>
<p>
   Vergleiche zwischen <code>Floats</code> sind oft schwierig, da durch die begrenzte Maschine
</p>
```

[ ]:
```
a = 0.1 + 0.2
b = 0.3
eq = a == b

print(eq)
```

25 Logische Operatoren

26 Logische Operatoren

Logische operatoren kombinieren zwei **Bool**sche Werte und geben gemäß ihrer Logik eine neue **Bool** zurück. | **Operator** | **Beschreibung** | **Beispiel** | **Ergebnis** | |————|————————————————————|————————————|————-| | `and` | Gibt `True` zurück, wenn beide Operanden `True` sind. | `True and True` | `True` | | | | `True and False` | `False` | | `or` | Gibt `True` zurück, wenn mindestens einer der Operanden `True` ist. | `True or False` | `True` | | | | `False or False` | `False` | | `not` | Negiert den Booleschen Wert des Operanden. | `not True` | `False` | | | | `not False` | `True` |

```
[ ]: # Logische Operatoren
a = True
b = False
# Und
und = a and b
# Oder
oder = a or b
# Nicht
nicht = not a

print(und, oder, nicht)
```

Bool'sche Werte können für mathematische Operationen genutzt werden, wobei sie entweder die Werte `True = 1` oder `False = 0` annehmen.

```
[ ]: a = True
b = 2.7
mult = a * b
add = a + b
div = a / b

print(mult, add, div)
```

Umgekehrt können logische Operationen zwischen sämtlichen Objekten durchgeführt werden. Dabei sind sämtliche einzelnen Datenobjekte `True`. Die einzige Ausnahme bildet die **Integer** `0`, die `False` ist. Variablen, die zwar deklariert wurden aber keinen Wert haben, bspw. ein leerer **String** `""`, sind ebenfalls `False`.

```
[ ]: a = "Apfel"
b = 0

und = a and b
oder = a or b
nicht = not a

print(und, oder, nicht)
```

27 Datentypen: Container

Variablen können zusammengefasst werden in Containern, die Operationen auf Gruppen von Daten erlauben. Es gibt 4 verschiedene grundlegende Arten von Containern:

- **Listen:** Listen enthalten geordnete Daten, die überschrieben werden können
- **Tupel:** Tupels enthalten geordnete Daten, die **nicht** überschrieben werden können
- **Sets:** Sets enthalten **ungeordnete, einzigartige** Daten
- **Dictionaries:** Dictionaries enthalten ungeordnete Key-Value Paare, bei denen jedem Key ein Objekt zugeordnet ist

Für den Moment konzentrieren wir uns auf **Listen**. Eine **Liste** enthält eine Menge an einzelnen

Elementen (**Integers**, **Floats**, **Strings**, **Bools**), die an einer bestimmten Position, einem Index, gespeichert sind. Der Index zählt die Elemente in der Liste, beginnend bei 0. Auf einzelne Elemente der Liste kann über den Index zugegriffen werden mittels eckiger Klammern `liste[index]`. Wenn versucht wird auf ein Element zuzugreifen, dass nicht in dem Container enthalten ist, wird ein Fehler produziert. Auf mehrere Elemente kann durch **Slicing** `list[index1:index2]` zugegriffen werden, wodurch alle Elemente zwischen den Indices ausgegeben werden.

```
<h3>Wichtig</h3>
<p>
   In Python wird immer ab 0 gezählt.
</p>
```

[15]:
```
# Listen Basics
elements = ["Hydrogen", "Helium", "Lithium", "Berrylium", "Boron", "Carbon",␣
 ↪"Nitrogen"]
print(elements[0], elements[-1])
print(elements[0:4])
```

```
Hydrogen Nitrogen
['Hydrogen', 'Helium', 'Lithium', 'Berrylium']
```

Quelle: Hier

28 Listen: Operationen

Listen können mit einer Vielzahl an Funktionen manipuliert werden.

Operation	Beschreibung	Syntax
Hinzufügen	Fügt ein Element am Ende der Liste hinzu	`liste.append(element)`
Einfügen	Fügt ein Element an einem bestimmten Index ein	`liste.insert(index, element)`
Löschen	Löscht das Element am angegebenen Index	`del liste[index]`
Länge bestimmen	Gibt die Anzahl der Elemente in der Liste zurück	`len(liste)`
Sortieren	Sortiert die Liste in aufsteigender Reihenfolge	`liste.sort()`
Zählen	Zählt, wie oft ein Element in der Liste vorkommt	`liste.count(element)`
Finden	Gibt den Index des ersten Vorkommens eines Elements	`liste.index(element)`

[7]:
```
elements = ["Hydrogen", "Helium", "Lithium", "Berrylium", "Boron", "Carbon",␣
 ↪"Nitrogen"]
elements.append("Oxygen")
print(elements)
elements.index("Carbon")
```

```
['Hydrogen', 'Helium', 'Lithium', 'Berrylium', 'Boron', 'Carbon', 'Nitrogen',
'Oxygen']
```

[7]: 5

Listen sind keine Vektoren und sind, genau wie **Strings**, im Allgemeinen nicht kompatible mit mathematischen Operatoren.

[ ]:
```
my_list = [1, 2, 3, 4]
my_list = my_list * my_list
print(my_list)
```

29 Kontrollstrukturen

Quelle: Hier

30 Kontrollstrukturen: Iterieren & Loops

Anstatt einzeln auf die Elemente einer Liste zuzugreifen, kann eine Operation nacheinander auf alle Elemente einer Liste angewandt werden in einem sogenannten `for` loop. Dabei wird die Liste Index für Index durchgegangen (**iteriert**), bis alle Elemente einmal behandelt wurden. Der Loop wird eingeleitet durch das **Keyword** `for ... in` und folgt der allgemeinen Syntax `for [name] in [iterable]:`

[ ]:
```
my_list = [5, 3, 6, 2]
for element in my_list:
    print(element)
```

Quelle: Hier

```
<h3>Wichtig</h3>
<p>
Ein **For-Loop** ist eine **Kontrollstruktur**, innerhalb derer der darauffolgende Code nach be
```

31 For-Loops

Loop	Funktion	Syntax
Liste	Loop über Elemente einer einzelne Liste	`for el in list:`
Zahlen	Loop über Zahlen von `start` bis `end`	`for element in range(start, end):`
mehrere Listen	Loop über die Elemente mehrerer Listen	`for el1, el2, ... in zip(list1, list2, ...):`
Index und Liste	Loop über den Index und die Elemente einer Liste	`for idx, el in enumerate(list):`

Ein For-Loop muss nicht direkt über die Elemente einer Liste iterieren. Oft ist es leichter, über Zahlen, bspw. die Indices einer Liste, zu loopen. Dies wird mit der `range()` Funktion erreicht.

[ ]:
```python
my_list = [5, 3, 6, 2]
n = len(my_list)

for i in range(n):
    my_list[i] = my_list[i] + 1

print(my_list)
```

Der Code innerhalb einer Kontrollstruktur kann beliebig lang und komplex sein, und es ist auch möglich (und häufig) mehrere Loops ineinander zu schachteln.

Eine kompaktische und praktische Methode um kurze Operationen auf Listen auszuführen sind **List Comprehensions**. Diese folgen einer einfachen einzeiligen Syntax:

[ ]:
```python
squares = [x ** 2 for x in range(10)]
print(squares)
```

32 Loops über mehrere Variablen

Anstatt nur eine einzelne Variable in einem **For-Loop** zu nutzen, können mit der `zip` Funktion auch mehrere Variablen in einem einzelnen Loop genutzt werden wenn sie die gleiche Länge haben.

[3]:
```python
molecules = ["Mol1", "Mol2", "Mol3"]
weights = [109.3, 165.9, 93.5]
for mol, w in zip(molecules, weights):
    print(f"{mol} wiegt {w} g / mol")
```

```
Mol1 wiegt 109.3 g / mol
Mol2 wiegt 165.9 g / mol
Mol3 wiegt 93.5 g / mol
```

Um gleichzeitig über den Index und ein Element aus einem iterierbaren Objekt zu loopen, kann die `enumerate()` Funktion genutzt werden.

[4]:
```python
molecules = ["Mol1", "Mol2", "Mol3"]
weights = [109.3, 165.9, 93.5]
for i, mol in enumerate(molecules):
    w = weights[i]
    print(f"{i}. {mol} wiegt {w} g / mol")
```

```
0. Mol1 wiegt 109.3 g / mol
1. Mol2 wiegt 165.9 g / mol
2. Mol3 wiegt 93.5 g / mol
```

33 Kontrollstrukturen: Entscheidungen

Quelle: Hier

34 Kontrollstrukturen: Entscheidungen

Der For-Loop ist eine der grundlegenden Kontrollstrukturen in Python. Kontrollstrukturen ermöglichen es, dem Programm eine komplexe Struktur zu geben, bspw. durch Schleifen (**Loops**) sich wiederholenden Codes oder durch **Entscheidungsstrukturen**, welche Codeabschnitte nur in gewissen Fällen ausführen.

Entscheidungsstrukturen sind sogenannte **If/Else** Blocks. Sie folgen der Syntax `if Bedingung: ... else: ....` Dabei wird der erste Code Block ausgeführt wenn die Bedingung erfüllt ist, während der andere ausgeführt wird wenn sie **nicht** erfüllt ist. Genau wie beim For-Loop werden die Code-Blocks durch ihre Einrückung voneinander getrennt.

[11]:
```
# If-else
color = "rot"

if color == "rot":
    print("Lithium!")
else:
    print("Natrium!")
```

```
Lithium!
```

35 If - Elif - Else

Eine Bedingung kann durch einen **Bool** Wert ersetzt werden oder mehrere Bedingungen verknüpfen gemäß der **Bool** Logik. Mehrere Spezialfälle können mit durch das **elif** Keyword hinzugefügt werden.

[13]:
```
# If-elif-else
color = "blau"

if color == "rot":
    print("Lithium!")
elif color == "gelb":
    print("Natrium!")
else:
    print("Kalium")
```

```
Kalium
```

If Statements können auch in List Comprehensions genutzt werden.

[ ]:
```
my_list = [x ** 2 for x in range(10) if x > 5]
print(my_list)
```

36 Wie schreibe ich guten Code? Aufbau eines Programms

Beautiful is better than ugly

Eine Grundsatz des Programmierens lautet **Ein Programm wird häufiger gelesen als geschrieben**

Was bedeutet: **Lesbarkeit zählt!**

Daher gibt es einen grundlegenden Aufbau, dem ein vollständiges Programm folgen sollte: - Header - Modul Importe - Definitionen: Konstanten, Funktionen (& Klassen) - Code

36.0.1 Header

In der Kopfzeile werden Allgemeine Informationen über das Programm gegeben. Im Wesentlichen beinhaltet der Header **wer** dieses Programm **wann** zu **welchem Zweck** geschrieben hat, und eine kurze Anleitung. Da ein Header oft mehrere Zeilen umfasst, lohnt es sich ihn als **Docstring** bestehend aus je **3 Anführungszeichen** """ """ zu verfassen. Ein Docstring ist ein mehrzeiliger Kommentar, der nicht vom Interpreter gelesen wird.

```
"""
Author: Jannis Kockläuner
Datum: 08.11.2024
Nutzloses Programm für die Vorlesung Einführung in Python
"""
```

Danach beginnt das Programm mit den Modulimporten. “‘python import numpy as np import matplotlib.pyplot

Funktionen werden gesammelt am Anfang des Programms definiert.

[28]:
```
def function_1():
    """
    Diese Funktion tut nichts.
    """
    return "Ich tue nichts"

def function_2():
    """
    Diese Funktion tut ebenfalls nichts
    """
    return "Ich tue noch
```

Danach folgt der eigentliche Code Block.

[26]:
```
text_1 = function_1()
text_2 = function_2()

print(text_1)
print(text_2)
```

```
Ich tue nichts
Ich tue noch weniger
```

37 Wie schreibe ich guten Code? PEP8 Standards

PEP 8 (Python Enhancement Proposal) ist im Prinzip die offizielle Python Style Richtlinie. Nach dem Motto **Lesbarkeit zählt** setzt PEP8 Standards, um Code möglichst lesbar zu strukturieren. Das beinhaltet unter anderem: - Bennenung von Variablen und Funktionen - Einrückungen - Leerzeichen - Kommentare - ...

37.0.1 Benennung von Variablen und Funktionen

Explicit is better than implicit

Variablen und Funktionen sollten immer ihre Funktion beschreiben. Dabei gelten folgende Regeln: - Variablen und Funktionen werden kleingeschrieben - Wörter werden durch Unterstriche _ getrennt

[35]:
```
# Bad Praxis
def Superwichtigefunktion3000(a, b, c):
    """
    Diese Funktion setzt Prioritäten.
    """
    if c:
        print(a)
    else:
        print(b)


O = "Super wichtig!"
p = "Nicht so wichtig!"
Q = True

Superwichtigefunktion3000(O, p, Q)
```

```
Super wichtig!
```

[36]:
```
# Pep8
def super_wichtige_funktion(wichtig, unwichtig, ist_wichtig):
    """
    Diese Funktion setzt Prioritäten.
    """
    if ist_wichtig:
        print(wichtig)
    else:
        print(unwichtig)


wichtig_text = "Super wichtig!"
unwichtig_text = "Nicht so wichtig!"
ist_wichtig_flag = True

super_wichtige_funktion(wichtig_text, unwichtig_text,
```

```
                        ist_wichtig_flag)
```

```
Super wichtig!
```

38 Wie schreibe ich guten Code? PEP8 Standards

38.0.1 Einrückungen

Einrückungen bestimmen den Ablauf eines Python Programms. Die Struktur eines Codes sollte direkt klar machen, welche Codeblöcke zusammen ausgeführt werden. Es gelten folgende Konventionen: - 4 Leerzeichen Einrückung - Leerzeichen statt Tabs

[53]:
```
# Bad Practice
fruchtkorb_1 = ["Apfel", "Apfel"]
fruchtkorb_2 = ["Pflaume", "Birne"]
fruchtkorb_3 = ["Pflaume", "Pflaume"]

# Könnnen wir einen Fruchsalat mit 3 verschiedenen Früchten machen?
for fruit_1 in fruchtkorb_1:
  for fruit_2 in fruchtkorb_2:
     if fruit_1 != fruit_2:
      for fruit_3 in fruchtkorb_3:
           if fruit_2 != fruit_3 and fruit_1 != fruit_3:
            print(f"Es gibt Fruchtsalat mit {fruit_1}, {fruit_2} und {fruit_3}")
```

```
Es gibt Fruchtsalat mit Apfel, Birne und Pflaume
Es gibt Fruchtsalat mit Apfel, Birne und Pflaume
Es gibt Fruchtsalat mit Apfel, Birne und Pflaume
Es gibt Fruchtsalat mit Apfel, Birne und Pflaume
```

[48]:
```
# Pep8
fruchtkorb_1 = ["Apfel", "Birne"]
fruchtkorb_2 = ["Pflaume", "Birne"]
fruchtkorb_3 = ["Pflaume", "Apfel"]

# Könnnen wir einen Fruchsalat mit 3 verschiedenen Früchten machen?
for fruit_1 in fruchtkorb_1:
    for fruit_2 in fruchtkorb_2:
        if fruit_1 != fruit_2:
            for fruit_3 in fruchtkorb_3:
                if fruit_2 != fruit_3 and fruit_1 != fruit_3:
                    print(f"Es gibt Fruchtsalat mit {fruit_1}, {fruit_2} und␣
 ↪{fruit_3}")
```

```
Es gibt Fruchtsalat mit Apfel, Birne und Pflaume
Es gibt Fruchtsalat mit Birne, Pflaume und Apfel
```

38.0.2 Leerzeichen

Sparse is better than dense

Leerzeichen sind in den meisten Fällen optional, verbessern die Lesbarkeit des Codes jedoch stark. Im wesentlichen gilt: - Vor und nach `=` wird **je ein** Leerzeichen eingefügt - **nach** einem Komma `,` wird **ein** Leerzeichen eingefügt - Vor und nach mathematischen Operatoren wird **je ein** Leerzeichen eingefügt

[ ]: