vorlesung_1

November 5, 2024

1 Einführung in (gute) Programmierung mit Python

Vorlesung Theoretische Chemie (WiSe 2024/2025)

2 Motivation

- coole Plots
- schnelle Datenevaluation
- Beispiele Matplotlib (Animation)
- Teilchen im 2D/3D-Kasten

3 Motivation: Visualisierung

3.0.1 Beispiel: 3D-Plots

3.0.2 Beispiel: Animationen

4 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.

4.0.1 Beispiel: Summiere die Zahlen von 1 bis 10

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

print(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.

5 Warum Python?

- **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.
- Für komplexe Aufgaben ist es oft **einfacher und sinnvoller** nach dem passenden Paket zu suchen anstatt selbst sämtliche Funktionalitäten zu programmieren.

6 Warum Python?

- **Einfach und leicht zu lesen:** Der Code sieht oft wie „pseudocode“ aus, 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!")
```

7 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

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

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

8 Warum Python?

- **Bibliotheken und Frameworks:** 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.

8.0.1 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 Sinuxfunktion
# 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)
```

8.0.2 Einfachheit > Performance

Die Grundlage des Programmierens ist der Programmcode der von Entwickler*innen geschrieben wird. Guter Code folgt einfachen Prinzipien: - Lesbarkeit - Verständlichkeit - Konfigurierbarkeit - Don't repeat yourself*

8.0.3 Beispiel: Fläche eines Kreises

$A = \pi * r^2$

[ ]:
```
# Aufgabe: Berechne die Fläche zweier Kreise A = pi * r^2
print(f"Die Fläche von Kreis 1 beträgt {3.1415 * (2 ** 2)}")
print(f"Die Fläche von Kreis 2 beträgt {3.1415 * (4 ** 2)}")
```

[ ]:
```
# 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)
```

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

9 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.

[ ]:
```
# 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)
```

10 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

[ ]:
```
# Variablen dienen der Datenspeicherung!
my_integer = 10
my_integer - 3
print(my_integer)
```

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
```

11 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))
```

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

12 Basics: Zahlen

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

[ ]:
```python
# 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.

[ ]:
```python
# 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.

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

13 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
# multiplication
mul = a * b
# division
div = a / b
# modulo
mod = a % b
# exponent
power = a ** b

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

```
<h3>Wichtig</h3>
<p>Die Reihenfolge der mathematischen Operationen folgt der Standardreihenfloge mathematischer
```

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

[ ]:
```

```

14 Basics: 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
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

```
<h3>Wichtig</h3>
<p>
    Funktionen wie <code>print()</code> benutzen immer(!) Klammern, die bestimmen, mit welchen
    <a href="https://docs.python.org/3/" target="_blank" style="color: darkblue; text-decoratio
        Python-Dokumentation
    </a>
    der Funktionen!
</p>
```

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

```

```python
# 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)
```

[ ]:

```python

```

15 Basics: Strings

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

[ ]:

```python
my_text = "Hello World"
print(my_text)
```

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

[ ]:

```python
my_text = "10"
my_int = 2
print(my_text + my_int)
```

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	`"eins zwei drei".split()`	`['eins', 'zwei', 'drei']`

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

[ ]:

```python
# 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))
```

16 Basics: Bool'sche Logik

Bool'sche Werte können nur zwei Werte annehen: 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)
```

17 Basics: 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)
```

18 Basics: 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]`.

[ ]:
```
# Listen Basics
my_list = [3, "PC"]
print(my_list[1], my_list[0])
```

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

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)`

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)
```

19 Basics: 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` und folgt der allgemeinen Syntax `for [name] in [iterable]:`

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

Ein **For-Loop** ist eine **Kontrollstruktur**, innerhalb derer der darauffolgende Code nach bestimmten Regeln ausgeführt wird. Der Geltungsbereich aller **Kontrollstrukturen**, also auch von For-Loops, wird bestimmt durch die **Einrückung** (Intendation), die üblicherweise 4 Leerzeichen beträgt.

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.

[ ]:
```
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:

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

20 Basics: Kontrollstrukturen

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.

[ ]:
```
# If-else
my_list = [x for x in range(5)]
for element in my_list:
    if element > 2:
        print(element)
    else:
        print("Uninteressante Zahl kleiner 3!")
```

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.

[ ]:
```
# If-else
my_list = [x for x in range(5)]
for element in my_list:
    if element > 2 and element < 4:
        print(element)
    elif element == 0:
        print("Wir starten mit 0!")
    elif element == 4:
        print("Und enden mit 4!")
    else:
        print("Uninteressante Zahl kleiner 3!")
```

If Statements können auch in List Comprehensions genutzt werden.

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