2. Daten- und Programmstruktur in Python

Markus Reinert[1]

Leibniz-Institut für Ostseeforschung Warnemünde (IOW)

23. April 2021

[1] ✉ markus.reinert@io-warnemuende.de

Python-Kurs 2

- Erinnerung: Listen und Tuples
- Listen und Tuples: Zugriff auf die Daten
- *Dictionaries*: Strukturierte Datensammlungen
- *For-* und *While*-Schleifen
- Bedingte Anweisungen mit *if*
- *In-line* Bedingungen und Schleifen
- Funktionen 1
- Funktionen 2
- Bibliotheken (auch im Lockdown geöffnet)
- Stilvoll programmieren

Erinnerung: Listen und Tuples

Liste	**Tuple**
Sammlung beliebiger Daten	unveränderbare Liste
`temp = [20, 22, 20.5, "error", 21]`	`dimensionen = (128, 256, 32)`
meist für *unbekannte* Anzahl von Daten *desselben* Typs verwendet	meist für *bekannte* Anzahl von Daten *unterschiedlichen* Typs verwendet

Named Tuple – oft besser geeignet als ein einfaches Tuple:

```
# zu Beginn des Programms:
from collections import namedtuple
# bevor das Tuple verwendet wird:
Dimension = namedtuple("Dimension", ("x", "y", "z"))
# Definition des NamedTuples:
dim = Dimension(128, 256, 32)     # oder: Dimension(x=128, y=256, z=32)
# Verwendung:
dim.x     # ergibt 128
dim.z     # ergibt 32
```

Listen und Tuples: Zugriff auf die Daten

- Anzahl der Elemente in einer Liste: `len(daten)`
- `daten[i]` gibt Element `i` aus Liste `daten` zurück
  (geht auch mit NamedTuples, sollte dort aber vermieden werden, siehe vorherige Folie)
- `daten[i] = x` überschreibt Element `i` der Liste `daten` mit dem Wert von `x`
  (geht nicht mit (Named)Tuples)
- **Wichtig:** Zählung beginnt bei `i = 0`
  Bsp.: hat eine Liste 10 Elemente, so können Indizes 0 bis 9 verwendet werden

Dictionaries: Strukturierte Datensammlungen

- `personal_data = {"name": "Max", "age": 23, "city": "Rostock"}`
- Zugriff auf ein Element: `personal_data["name"]`
- Zugriff auf alle Elemente: `personal_data.items()`

For- und *While-*Schleifen

For-Schleife:

feste Anzahl an Wiederholungen, z. B.:

- *n*-mal
- für jedes Element einer Datensammlung
- für jede Datei in einem Ordner

While-Schleife:

Anzahl der Wiederholungen geknüpft an eine Bedingung, z. B.:

- Konvergenz eines numerischen Verfahrens
- sequentielles Lesen von Daten

Wichtig:

Jede Anweisung, die zu einer Schleife gehört,
muss mit derselben Anzahl an Leerzeichen eingerückt sein!

Üblich sind 4 Leerzeichen oder ein Tab.

Ausnahme: einzelne Anweisungen können in derselben Zeile stehen.

Bedingte Anweisungen mit *if*

Vergleichsoperatoren:

- Gleichheit: `==`
- Ungleichheit: `!=`
- Vergleich: `<`, `>`, `<=`, `>=`
- Verknüpfung: `and`, `or`, `not`

Mehrere Vergleiche:

```
if x > 0:
  print("x ist positiv")
elif x == 0:
  print("x ist null")
else:
  print("x ist negativ")
```

Wichtig: richtige Einrückung beachten (wie bei Schleifen).

In-line Bedingungen und Schleifen

- können manchmal praktisch sein
- sollten Lesbarkeit verbessern
- können manchmal durch Funktionen ersetzt werden (z. B. `map`)

Beispiele:

- `half_data = [x / 2 for x in data]`
- `inverse_data = [1 / x if x != 0 else 0.0 for x in data]`
- ```
  print(
      "The influence is",
      "significant" if p <= alpha else "not significant",
  )
  ```
  ```

Funktionen 1

Anweisungen, die an mehr als einer Stelle verwendet werden
und eine bestimmte *Funktion* erfüllen, können als Funktion definiert werden.

Funktionen haben Eingabewerte (Argumente) und Ausgabewerte.

Arten von Funktionsargumenten:

Argumente ohne Namen

Beispiel:

```
print(3, "ist kleiner als", 4)
```

Optionale Namen

Beispiel:

```
np.zeros((3, 4))
# ist equivalent zu
np.zeros(shape=(3, 4))
```

Argumente mit Namen

Beispiel:

```
print("Dieser Text", end=" ")
# Argument ‘‘end’’ kann nicht
# ohne Namen verwendet werden
print("geht hier weiter.")
```

- **Verwende die Schreibweise mit Namen, wenn es die Lesbarkeit verbessert!**
- Argumente ohne Namen müssen immer vor Argumenten mit Namen kommen.
- Die Reihenfolge von Argumenten ohne Namen ist von Bedeutung.
- Die Reihenfolge von Argumenten mit Namen ist egal.

Funktionen 2

Einfachstmögliche Funktion:

```
def square(x):
    return x ** 2
```

Kompliziertere Funktion:

```
def gauss(x, mu=0.0, sigma=1.0):
    """Calculate the value of the Gauss distribution with mean mu and std.dev. sigma at x.

    x: float
    mu: mean of the distribution (default 0)
    sigma: standard deviation (default 1)
    """
    if sigma == 0:
        print("Error!")
        return np.NaN   # beendet den Funktionsaufruf
    z = (x-mu) / (2*sigma)
    return np.exp(- z ** 2) / np.sqrt(2*np.pi*sigma**2)
```

Sammlungen von Funktionen und Daten

Sammlung von (veränderlichen) Daten:	**Sammlung von Funktionen (und ggf. Konstanten):**	**Sammlung von Daten mit *zugehörigen* Funktionen:**
Listen, (Named) Tuples, Dictionaries, andere Datenstrukturen	Module (= Bibliotheken): Python-Dateien, die man mit `import` in ein Programm einbinden kann	Klassen

➔ Beispiele dafür werden wir in den nächsten Kursen kennenlernen

Programmierstil

"Programs are meant to be read by humans
and only incidentally for computers to execute."
(Harold und Sussman)

- → aussagekräftige Namen wählen
- → Leerzeichen einbauen
- → lange Zeilen umbrechen, am Besten innerhalb von Klammern
- → Einrückungen verwenden
- → Kommentare hinzufügen *# in Python geht das so*

Weitere Tipps:
`https://www.python.org/dev/peps/pep-0020/`
`https://google.github.io/styleguide/pyguide.html`