Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2024-05 Dozent: Karsten Flügge

Themen des Tages

- Operatoren
- Listen, Dictionaries, Tupel, Mengen
- list, dict, tuple, set

- Speichern und Lesen in / aus Datei

Operatoren

Vorrang von Operatoren, Gegebenenfalls Operationen Klammern oder in mehrere Zeilen teilen.

'Präzedenz' ≈ Bindung (Gruppierung)

```
(...)                # Klammern
x[i], x(...), x.y  # Zugriff, Aufruf  z.B. .real
**                   # Potenz
+x, -x, ~x         # Vorzeichen, bitweise Negation
*, /, //, %        # ÷ Punktrechnung   3 * True == 3
+, -                 # Strichrechnung
<<, >>               # Bitverschiebung
&                     # bitweises UND
^                     # bitweises XOR
|                    # bitweises ODER
== != < <= > >=          # Vergleiche
not                    # logisches NICHT  True, False
and                    # logisches UND
or                    # logisches ODER
=  :=  += ...       # Zuweisung; kein Ausdruck
```

Operatoren

Zuweisung = seit 3.8 :=

⚠️ Vorsicht, NIE Vergleich == mit Zuweisung = verwechseln!

```
>>> x = 1
>>> y = 2
>>> x == y
False / True  "Boolean" Wahrheitswert
>>> x = y
>>> x == y
True
```

`nil` ist null aus anderen Sprachen

gibt keine Pointer

Logisch: and, or, not, ^ (xor)

*= /= %= += -= # in place arithmetic / compound arithmetic zB compound addition

Operatoren

Mathematisch: + - / * % (modulo rest) ** (hoch, potenz, ^, ^^)
werden abgebildet auf Funktionen mit speziellem Namen
__add__

Vergleich: == , >, <
__eq__ __gt__ __lt__ equal greater / less than

Logisch: and, or, not, ^ (xor)

Operatoren

Operatoren + - / * % ... werden abgebildet auf Methoden mit speziellem Namen

```
'__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__',
'__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__',
'__getstate__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__',
'__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__',
'__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__',
'__xor__', 'as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'is_integer', 'numerator', 'real', 'to_bytes']
```

Ungleichheit

Ungleichheit

3 != 4 heisst "3 ungleich 4"

Hintergrund in c etc darf man !x schreiben fuer "nicht x"

⚠️ in python schreibt man **not x** für logische Negierung!

not x == y ist das selbe wie x != y

Syntaktischer Zucker

Oft möchte man eine Variable um eins erhöhen

x = x + 1

dafür gibt es spezielle Zuweisung

x += 1

Der 'Trick' geht mit verschiedenen Operatoren:

*= /= %= += -= # in place arithmetic / compound arithmetic operators zB compound addition

Aufgaben

Finde für je zwei Operatoren ein Beispiel, wo die Reihenfolge wichtig ist, also wo mit und ohne Klammer ein anderes Ergebnis herauskommt
z.B. 2 + 3 * 4 != (2 + 3) * 4

```
**                  # Potenz = hoch
+x, -x              # Vorzeichen
*, /, //, %         # Punktrechnung ÷    3 * True == 3
+, -                # Strichrechnung
== != < <= > >=         # Vergleiche
```

Operatoren Ausdruck

Die Anwendung von Operatoren sind ein Beispiel für einen sogenannten Ausdruck (expression)

"Alles was bei einer Variablen Zuweisung rechts stehen kann"

x = 3 * 3
x = print('hallo')

Ein Ausdruck welcher nicht ausgegeben oder zugewiesen wird macht normalerweise wenig Sinn!

```
len('hallo') # ⚠️ use it!?
print(len('hallo')) # ok, verwendet
laenge = len('hallo') # ok, verwendet (später…)
```

String Operatoren

```
"a" in "Hallo"   # __contains__

Modulo zum Werte in Text einsetzen:

"Hello %s, you are %d years old" % ("Pannous", 99)   # __mod__
# 'Hello Pannous, you are 99 years old'

Nutze besser format string f"Hello {name}"

plus ist klar
mal : Wiederholung
Vergleich: Lexikalische Sortierung "a" < "b"

Achtung: Groß vor klein!
>>> 'a' < 'B'
False

Vergleich klar
```

Vorschau Funktionen vs Methoden vs Operatoren

Führen Aktionen aus / holen Informationen
Rückgabe **Typ** kann unterschiedlich sein

Funktionen
stehen unabhängig von Objekten (ohne Punkt .)
print(x)
help()
dir(x) **#** listet alle Methoden vom Objekt x
len(x) # int

Methoden
hängen an Objekten (Dinge mit Eigenschaften), werden mit '.' Punkt aufgerufen, weiter mit **()**
z.B.
"HELLO"**.**lower() => "hello"
Alle Methoden zwei Unterstrichen rufen wir NICHT direkt auf, sondern:

Operatoren
hängen an Objekten aber werden ohne . aufgerufen
"a" + "b" und werden **intern** auf Methoden abgebildet z.B. __add__ "häßlich by design"

Listen

Listen sind container (≈Arrays, Felder) zum sammeln von verschiedensten Objekten

list eckige klammer und komma

```
wochentage = ["Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag", "Sonntag"]

for i in wochentage:
  print(i)

print(wochentage[0]) # "Montag"

so wie bei Zeichen in Text fangen wir bei 0 an zu zählen!
```

Listen

```
wochentage = ["Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag", "Sonntag"]
```

erweitern

```
wochentage + ["Feiertag"]
```

löschen

```
wochentage.remove("Feiertag")
del wochentage[-1]  # löscht letztes
```

abfragen

```
wochentage[0] # "Montag"
```

setzen

```
wochentage[0]="Monday" #jetzt Englisch!
```

prüfen

```
"Dienstag" in wochentage
```

Listen

Vergleich

```
>>> [1,2,3]==[1,2,3]
True
>>> [1,2,3]==[1,3,2] Reihenfolge ist wichtig!
False
>>> [1,2,3]==[1,2,3.0] Darstellung kann abweichen
True
>>>
```

solange für jedes Element a_i == b_i gilt

Listen

Listen können verschiedene Dinge enthalten:

```
mischSack = ["Montag", 1, 4.4]

for i in mischSack:
  print(i)
```

Zugriff aufs n+1te Element über list[n] **index** :

```
print(mischSack[2]) # 4.4
```

Zugriff aufs **letzte Element** über list[**-1**] index :

```
print(mischSack[-1]) # 4.4
```

Arbeiten mit Listen

- **__contains__** Liste enthält… ?

```
1 in [1,2,3] # True   x in xs == "x ∈ xs"
```

- *__add__ Liste* ***erweitern***

```
>>> [1,2] + 3
TypeError: can only concatenate list (not "int") to list  … AHA:

>>> [1,2] + [3]
[1, 2, 3]
```

- len __len__ Länge

```
len(1,2,3) # 3
```

- **remove** Eintrag entfernen:

```
>>> [1,2,3]-[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'list' and 'list'

>>> x=[1,2,3]
>>> x.remove(2) # ⚠️ EIN objekt, was gelöscht werden soll, nicht der index
>>> x
[1, 3]

del x[1]  # löschen an index !
```

Arbeiten mit Listen

⚠️ **Modifizierende** funktionen (vs pure)

• **pop() letztes Element herausploppen (lesen und entfernen)**

```
>>> x = [1,2,3]
>>> y = x.pop()
3
>>> x
[1,2]
```

pop(n) zum aktiven entfernen des n+1 ten Elementes

externer ***operator del*** *bildet auf* ***__delitem__*** *ab*
del *a[1]*

• ***.append()*** *Liste erweitern (vs __add__ !)*

```
>>> x = [1,2,3]
>>> x.append(4)
>>> x
[1, 2, 3, 4]  == [1,2,3] + [4]
```

• *reverse() umdrehen*

```
>>> x = [1,2,3]
>>> x.reverse()
>>> x
[3, 2, 1]
```

dir(list) überschaubar

Arbeiten mit Listen

⚠ **Modifizierende** funktion (vs pure)
⚠ anders als bei str kann man in einer Liste elemente direkt **setzen / überschreiben**

```
>>> x = [1,2,3]
>>> x[1]=5
>>> x
[1, 5, 3]
```

allgemeine Syntax liste[index-n]=wert

dir(list) überschaubar

Slicing

💡 wie str kann man in einer Liste Elemente-Bereiche (**Teil-Listen**) auslesen

```
>>> x=["a","b","c","d"]

>>> x[:2] # first 2 elements
['a', 'b']

>>> x[2:] # after first 2 elements
['c', 'd']

>>> x[2:4] # from index 2 to index 4 exklusive ⚠️
['c', 'd']

x[2:-1] # von 2 bis zum letzten Element exklusive

allgemein
liste[start:ende]   start=0 end-index exklusive

Aufgabe jetzt: mit Listen experimentieren
gib mir ["b","c"] via slicing

default ende = len(x)
```

Fortgeschrittenes Slicing

Slicing ist eine Methode in Python, um Elemente aus Sequenzen wie **Listen**, **Tupeln** und **Strings** herauszugreifen. Fortgeschrittenes Slicing erweitert die Grundkonzepte und bietet mehr Flexibilität beim Zugriff auf Teile einer Sequenz.

```
liste[start:end:step]
```

```
liste = [1, 2, 3, 4, 5, 6, 7, 8, 9]
jedes_zweite = liste[::2] # "Schrittlänge 2"
umgekehrt = liste[::-1] # "Schrittlänge -1 : gehe rückwärts"
```

Kombination von Slicing-Parametern

- Kombination von Start-, End- und Schrittindizes zur präzisen Kontrolle über die Auswahl.
- Beispiel: Auswahl jedes dritten Elements in einem Teilstück der Liste:

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
teilstueck = liste[1:8:3]
[1, 4, 7]
```

Aufgabe jetzt: mit Listen [::] slicing experimentieren
gib mir [2,4,6,8] via slicing
gib mir [2,5,8] via slicing
gib mir [8,6,4,2] via slicing # "Schrittlänge -2"

Sortieren

⚠️ *Modifizierend!*

```
>>> [1,3,2].sort()
>>>
>>> x
[1, 2, 3]
```

Später fortgeschrittene Sortierkriterien z.B.

nach Länge:

```
woerter = ["Banane", "Apfel", "Kirsche"]
woerter.sort(key=len)
print(woerter)  # ['Apfel', 'Banane', 'Kirsche']
```

Umdrehen

⚠️ *Nicht Modifizierend!*

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> umgedreht = reversed(liste)  # bildet ab auf liste.__reversed__()
>>> list(umgedreht)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

⚠️ *Modifizierend!*

```
>>> liste.reversed()
>>>    # kein Rückgabewert!
>>> liste
>>> [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
# liste[1,7]
```

PRO: "flachmachen" von verschachtelten Listen

```
sum kann anscheinend auch flaten:
doublelist =[[1, 2, 3], [5, 3]]
sum(doublelist,[])
```

Aufgaben für Listen

- Aufgaben für Listen

- a) Erstelle Liste mit 3 Namen, füge einen weiteren hinzu und lösche den zweiten
- b) Verdoppele alle Namen indem die Liste an sich selbst angefügt wird
- c) lösche letzten Namen und zeige ihn gleichzeitig an
- d) sortiere liste und zeige sie an
- e) drehe sie um
- Zusatzaufgaben wenn jemand vorzeitig fertig ist:
- f) prüfe ob "Bob" in der Liste vorkommt
- g) an welcher Stelle kommt "Alf" vor?
- h) lösche "Alf". Was passiert wenn er nicht existiert hat?
- i) zähle, wie oft "Ute" vorkommt
- j) was ist die Länge der finalen Liste?

Extra Aufgaben für Listen

- Erstelle eine Funktion, die zwei Listen erhält und prüft, ob in beiden die Zahl 42 vorkommt
- Erstelle eine Funktion, die prüft, ob alle Elemente einer Liste in einer anderen Liste **vorhanden** sind. (nicht notwendig an der selben Position)
- gemeinsamen Elemente zweier Listen:
  - Schreibe eine Funktion, die die Schnittmenge zweier Listen zurückgibt.
- Definiere **pure** sortier Funktion sort([2,3,1]) oder sort(3,1,2) => [1,2,3]
- Entferne Duplikate aus einer Liste (ohne Set-Verwendung, modifizierend oder per Konstruktion)
- Identifiziere das Element, das in einer Liste am häufigsten vorkommt. (später: via dict)
- Funktion Permutation / Rotieren von Elementen in einer Liste z.B. [1,2,3,4] => [2,3,4,1]
- IM KOPF: welche Elemente werden hier ausgewählt:

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
teilstueck = liste[2:6:3]
```

Pause

x

- **Dictionary**

  - **dict Dictionary "Wörterbuch" ≈**
  - **map "Abbildung" ≈**
  - **key-value store Schlüssel-Wert-Paare**

Eine **map** dient zur **Zuordnung** verschiedenster Daten(paare).

Z.B.

```
translations = {"dog":"Hund", "cat":"Katze"}
flaggen = {"Deutschland":"🇩🇪","Frankreich":"🇫🇷","Spanien":"🇪🇸"};
einwohner={"Deutschland": 80_000_000, "Frankreich": 67_000_000, "Spanien": 47_000_000}
kontostand={32532523:235235, 2325523:2353523}
quadrat={1:1, 2:4, 3:9}
```

- **Dictionary**

  Erstellung:

  Ein Dictionary wird mit geschweiften Klammern {} erstellt. Innerhalb der Klammern werden Schlüssel und Werte durch Doppelpunkte getrennt, und die Paare durch Kommas.

```
x = {}   # leere map
x = { "key" : value }
```

dictionary[]

Lesen von Elementen

Zugriff: Auf die Werte kann zugegriffen werden, indem der Schlüssel in eckigen Klammern [] nach dem Dictionary-Namen verwendet wird.

über index[] aber mit key / Schlüssel statt Zahl:

```
>>> translations = {"dog":"Hund", "cat":"Katze"}
>>> translations["dog"]
'Hund'
```

```
Wenn der Schlüssel eine Zahl ist, ist die Abfrage ähnlich wie bei Listen (aber direkt, nicht n-1 )
>>> nums={1:100,2:200}
>>> nums[1]
100
```

```
⚠️ Lesen von nicht existierenden Einträgen = Fehler !!
>>> nums={1:100,2:200}
>>> nums[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

Mit einer Map können Werte sofort gefunden werden:
flaggen["Deutschland"]
Keine Schleife zum Prüfen aller Länder notwendig

dictionary

Setzen von Elementen

Änderung: Werte können geändert werden, indem ein neuer Wert einem bestehenden Schlüssel zugewiesen wird.

```
translations = {"dog":"Hund", "cat":"Katze"}
Setzen / überschreiben eines Elementes:
>>> translations["dog"]="WauWau"
```

Hinzufügen: Neue Schlüssel-Wert-Paare können einfach durch Zuweisung eines Werts zu einem neuen Schlüssel hinzugefügt werden.

```
Setzen / Hinzufügen eines Elementes:
>>> translations["cheese"]="Kaese"
```

dictionary

Beispiel

```python
# Erstellung eines Dictionary
person = {"name": "Maria", "alter": 30, "beruf": "Ingenieurin"}

# Zugriff auf den Wert mit dem Schlüssel 'name'
print(person["name"])  # Ausgabe: Maria

# Ändern des Wertes mit dem Schlüssel 'alter'
person["alter"] = 31

# Hinzufügen eines neuen Schlüssel-Wert-Paares
person["wohnort"] = "Berlin"

print(person)
# Ausgabe: {'name': 'Maria', 'alter': 31, 'beruf': 'Ingenieurin', 'wohnort': 'Berlin'}
```

Aktuelle Aufgabe:
nachvollziehen (abtippen;) zum Vertraut machen mit der Syntax und kurz experimentieren. was fällt auf?

Was darf in ein dict ?

Während die **Werte beliebig** sein können, müssen die **Schlüssel** einige **spezifische Kriterien** erfüllen:

1. **Unveränderlichkeit**: Schlüssel in einem Dictionary müssen von einem unveränderlichen (immutable) Datentyp sein. Das bedeutet, dass Typen wie Strings, Integer, Floats, Tuples und Booleans als Schlüssel verwendet werden können, weil sie unveränderlich sind.
2. **Einzigartigkeit**: Jeder Schlüssel in einem Dictionary muss einzigartig sein. Wenn ein Schlüssel mehrfach verwendet wird, wird der ursprüngliche Eintrag durch den neuesten Wert überschrieben.

Was nicht erlaubt ist:

- **Listen und Dictionaries als Schlüssel**: Da Listen und Dictionaries veränderbare (mutable) Datentypen sind, können sie nicht als Schlüssel in einem Dictionary verwendet werden. Ein Versuch, sie als Schlüssel zu nutzen, führt zu einem TypeError.
- **Sets als Schlüssel**: Sets sind ebenfalls veränderbar und können nicht als Schlüssel verwendet werden.

```python
# Ein Dictionary, das unterschiedliche Datentypen speichert
komplexes_dict = {
    "string": "Hallo Welt",      # Ein String
    "integer": 42,               # Eine Ganzzahl
    "float": 3.14159,            # Eine Fließkommazahl
    "list": [1, 2, 3],           # Eine Liste
    "dict": {"a": 1, "b": 2},    # Ein weiteres Dictionary
    "bool": True,                # Ein Boolescher Wert
    "function": print            # Eine Funktion (print-Funktion)
}
```

Was darf in ein dict ?

Beispiele von erlaubten schlüsseln:

```
wahrheitswerte = {True: "Wahr", False: "Falsch"}

ziffern_zu_namen = {1: "eins", 2: "zwei", 3: "drei", 4: "vier", 5: "fünf"}

tuple_als_schluessel = {(1, 2): "Eins und Zwei", (3, 4): "Drei und Vier"}

x=5
variable_als_schluessel = {x: "Variable hatte Wert 5"}
print(variable_als_schluessel.keys())  # Ausgabe: dict_keys([5])


# Beispiele von verbotenen schlüsseln:

list_als_schluessel = {[1, 2]: "Eins und Zwei"}  # TypeError: unhashable type: 'list'

dict_als_schluessel = {{1: 2}: "Eins und Zwei"}  # TypeError: unhashable type: 'dict'
```

dictionary of dictionaries

Verschachtelte Daten

Da ein dict in python beliebige Werte enthalten kann, unter anderem auch wieder ein dict, lassen sich hervorragend hierarchische / verschachtelte Daten damit abbilden.

Beispiel:

```
# Ein Dictionary, das Informationen über eine Person enthält
person = {
   "vorname": "Max",
   "nachname": "Mustermann",
   "alter": 25,
   "wohnort": {
      "straße": "Musterstraße 42",
      "plz": 12345,
      "stadt": "Musterstadt"
   }
}
```

Es gibt kaum Beschränkungen für die Daten, die sich damit abbilden lassen,
daher nennt sich diese Art von Daten auch unstrukturiert
(anders als demnächst beim tuple und später bei Klassen, welche Struktur vorschreiben können)

dictionary of dictionaries

Aufgabe jetzt: mit dictionaries experimentieren

Aufgabe: fuege in eine liste drei personen mit name, alter und wohnort hinzu
Aufgabe: ^^ frage den Wohnort der zweiten Person mit [] index ab

Erkenntnisse:
Abfrage von 'position n' in dict nicht einfach, am besten man kennt den Schlüssel (hier: name)

Zusammenfügen von dictionaries
>>> {1:2} | {3:4}
{1: 2, 3: 4} # vereint! ⚠️ Achtung es können Elemente überschrieben werden

Unser del Operator funktioniert auch hier:
del person["vorname"]
person.**pop**("alter") **ist wieder destruktiv (modifizieren!)**

Vergleich mit json …

JSON ist ein Datenformat welches in JS entsprang aber heute weit verbreitet ist.
Es ist konzeptionell und syntaktisch sehr ähnlich (aber nicht identisch!) zu unserem dictionary:

```
// unser dict:
person = {
   "vorname": "Max",
   "nachname": "Mustermann",
   "alter": 25,
   "wohnort": {
      "straße": "Musterstraße 42",
      "plz": 12345,
      "stadt": "Musterstadt"
   }
}
// JSON:
{
  "vorname": "Max",
  "nachname": "Mustermann",
  "alter": 25,
  "wohnort": {
    "straße": "Musterstraße 42",
    "plz": 12345,
    "stadt": "Musterstadt"
  }
}
```

Vergleich mit json …

Datentypen und Struktur

1. **Datentypen**:
   - **Python-Dictionaries** können eine breite Palette von Datentypen als Schlüssel und Werte enthalten, einschließlich aller unveränderlichen und veränderlichen Datentypen, solange die Schlüssel unveränderlich sind.
   - **JSON** ist ein Datenformat, das eine begrenzte Anzahl von Datentypen unterstützt: strings, numbers, booleans, arrays (ähnlich wie Listen in Python), Objekte (ähnlich wie Dictionaries in Python, aber nur mit String-Schlüsseln) und null.
2. **Syntax**:
   - **Python-Dictionaries** werden mit geschweiften Klammern `{}` dargestellt, und Schlüssel-Wert-Paare werden durch Kommas getrennt. Schlüssel und Werte werden durch Doppelpunkte getrennt.
   - **JSON** hat eine ähnliche Syntax, jedoch müssen die Schlüssel immer in Anführungszeichen (meist doppelte) gesetzt werden, was in Python nicht immer notwendig ist.

Verwendungszweck

- **Python-Dictionaries** sind eine interne Datenstruktur in Python und werden direkt im Code verwendet, um Daten während der Laufzeit eines Programms zu speichern und zu verwalten.
- **JSON** wird hauptsächlich für die Datenübertragung zwischen einem Server und einem Webclient oder zwischen verschiedenen Systemen verwendet. Es ist sprachunabhängig und wird oft zum Speichern und Austausch von Daten in einem leicht lesbaren Format genutzt.

IN JSON SIND KEINE KOMMENTARE ERLAUBT! 🤯😡🤦‍♂️ Lösung … nutze Json5 !

Trees ≈ nested dictionaries

Speichern / Laden

Texte (strings) und allgemeine Daten lassen sich in Dateien schreiben und lesen
Dafür können wir die open() Funktion auf folgende Weise nutzen:

```
file_name = "test"

file_out = open(file_name, "w") # "w" ≈ "write"
file_out.write("Hallo Welt")
file_out.close() # ⚠️ nicht vergessen, nach dem Schreiben die Datei schliessen!

file_in = open(file_name)  # "r" ≈ "read" optional
zeilen = file_in.readlines()

print(zeilen)


Aufgabe: was passiert wenn ich ein dict in eine Datei speichern möchte?



# pro tip / später : "wb" für write binary
```

Serialisieren / Speichern von dict als json

Die Ähnlichkeit zu json lässt sich zunutze machen, um dictionaries leicht zu speichern und zu laden:

```
import json

obj = {"name": "Max", "alter": 25}
file_name = "data.json"

s = json.dumps(obj) # object als String serialisieren
obj = json.loads(s) # object aus String laden


json.dump(obj, open(file_name, "w"))  # object als Datei serialisieren
obj = json.load(open(file_name))      # object aus Datei laden

print(obj)
```

list comprehension

In Python, a list comprehension provides a concise way to create lists. It consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The expressions can be anything, meaning you can put all kinds of objects in lists.

The basic syntax is:

```
[expression for item in iterable if condition]
```

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

```
evens = [x for x in range(20) if x % 2 == 0]
```

dictionary comprehension

```
dict comprehension wie list comprehension:

letter_counts = {letter: 0 for letter in letters_to_count}
```

Aufgaben dictionary

- **Zusammenführen von zwei Dictionaries:** Schreibe eine Funktion, die zwei Dictionaries zusammenführt und dabei die Werte von übereinstimmenden Schlüsseln addiert
- **Umkehren der Schlüssel-Werte-Paare in einem Dictionary:** Erstelle eine Funktion, die die Schlüssel und Werte eines Dictionaries umkehrt.
- darf eine Datei (f = open(file_name)) als Schlüssel verwendet werden? warum(nicht)?
-

Pause

- x

tuple

Tupel sind wie **unveränderliche** Listen

```
leeres_tupel = ()
tupel = (1, 2, 3, 4)
```

Alles (bis auf Modifikationen!) was wir mit Listen können, können wir auch mit Tupeln

```
for i in tupel:
  print(i)

print(tupel[0])  # Ausgabe: 1
len(tupel)  # Ausgabe: 4
…
```

neue Tupel

Tupel sind wie **unveränderliche** Listen

```
leeres_tupel = ()  # leeres dict = {}  leere liste = []
tupel_mit_einem_element = (1, )
tupel = (1, 2, 3, 4)
```

Jedwelche Modifikation scheitert aber oder ist nur durch Kreation eines neuen Tupels erlaubt:

```
# tupel[0] = 5  # TypeError: 'tuple' object does not support item assignment
# tupel.append(5)  # AttributeError: 'tuple' object has no attribute 'append'

# Kreation eines neuen Tupels
neues_tupel = tupel + (5,)  # OK
tupel2 = (5, 6)
kombiniertes_tupel = tupel + tupel2  # (1, 2, 3, 4, 5, 6)
wiederholtes_tupel = tupel2 * 2  # (5, 6, 5, 6)
```

PS: anders als in anderen Sprachen hat python nativ keine benamten Tupel (x=1, y=2)
Wer struktur-sichere dictionaries möchte kann diese aber als modul importieren:

```
from collections import namedtuple
Person = namedtuple("Person", ["name", "age", "city"])
```

keine first class citizen

sets

Ein set ist wie eine Menge in der Mathematik, oder wie eine ungeordnete Liste in welcher garantiert keine Duplikate enthalten sind.

Der Begriff **`set`** in Python bezeichnet eine Sammlung, die keine doppelten Elemente enthält und deren Reihenfolge nicht definiert ist. Dies ist besonders nützlich, wenn man eine Gruppe von einzigartigen Elementen verwalten möchte, ohne sich um die Reihenfolge oder doppelte Werte kümmern zu müssen.

Hier sind einige grundlegende Operationen mit `set`:

1. **Erstellung eines Sets**: Ein Set wird mit geschweiften Klammern `{}` erstellt, wobei die Elemente durch Kommata getrennt sind. Ein leeres Set wird jedoch mit `set()` und nicht mit `{}` erstellt, da `{}` ein leeres Dictionary darstellt.

```
zahlen = {1, 2, 3, 4}
leeres_set = set()  # leeres_dict = {}
```

Hinzufügen von Elementen: Mit der Methode `.add()` kann ein einzelnes Element zu einem Set hinzugefügt werden.

```
zahlen.add(5)
```

Entfernen von Elementen: Elemente können mit der Methode `.remove()` entfernt werden. Wenn das Element nicht im Set vorhanden ist, wird ein Fehler ausgelöst.

```
zahlen.remove(2)
```

Alternativ kann `.discard()` verwendet werden, das keinen Fehler auslöst, wenn das Element nicht existiert.

```
zahlen.discard(10)
```

Prüfung, ob ein Element im Set ist: Dies kann mit dem `in` Schlüsselwort überprüft werden.

```
if 1 in zahlen:
    print("1 ist im Set")
```

Set-Operationen: Sets unterstützen mathematische Mengenoperationen wie Vereinigung, Schnitt und Differenz. (leider nicht direkt bei Listen)

```
1.set1 = {1, 2, 3}
2.set2 = {3, 4, 5}

4. vereinigung = set1 | set2  # {1, 2, 3, 4, 5}    pipe operator ( bitwise-or für Zahlen )
5. schnitt = set1 & set2      # {3}                ampersand operator (bitwise-and für Zahlen )
6. differenz = set1 - set2    # {1, 2}             normalerweise set1 \ set2 in der Mathematik set1 "ohne" set2
```

Diese Operationen machen `set` zu einem mächtigen Werkzeug für viele Probleme, bei denen es um die Verwaltung einzigartiger Elemente geht.

sets

Aufgabe jetzt: experimentieren mit tuple / set

Aufgabe: set in list umwandeln

Kleiner Nachtrag: Umwandeln

Typen lassen sich oft in einander umwandeln indem man den Typ wie eine Funktion verwendet (nennt sich später constructor)

```
>>> int("3")
3
>>> str(3)
'3'
>>> list({'a','b'})
['b', 'a']
>>> set(["a","b","a"])
{'b', 'a'}
```

Ausdruck versus Anweisung

Ausdruck (Expression) : Wert oder Code Fragment oder Berechnung welches Wert Erzeugt
Anweisung (Statement) : Komplette Code Zeile welche eine Aktion Ausführt

```
sum = 3 + 1; # '3 + 1' ist ein Ausdruck, das ganze ist ein Statement ( Zuweisung )
```

Arithmetic Expression:

```
int sum = a + b; // sum is 8
int wert = ( 1 + 2 ) * 3
```

Relational Expression:

```
isEqual = (a == b); // isEqual is false
kleiner10 = ( x < 10 )
```

Logical Expression:

```
jaOderNein = true or false;
ungleichZehn = x < 10 and x > 10
isEitherEven = (a % 2 == 0) || (b % 2 == 0); // isEitherEven is false
```

Function Call (Expression):

```
maxVal = max(a, b); // maxVal is 5
```

Ausdruck versus Anweisung

Anweisung (statement)

Deklaration:
x:int;
def test():pass

Definition:
Kontroll Ausdruck:
if, if:else:, while, for(:), for(;;)
return Statement (in a function):

Compound Statement (block / body / Körper / Anweisungsliste)
if x: (…)

Ausdruck versus Anweisung

R-Value versus L-Value

not explicitly applicable because Python handles variables and memory management differently.

Ausdrücke (Expressions) teilen sich in R-Value ≠ L-Value
Namensgebung von right/left, aber Fehlbezeichnung “misnomer”
i = 3; # 3 ist ein r value 'rechts', i ist ein l value 'left'
j = i; # ⚠ i ist immer noch ein l value!
j = i + 1; # ist wieder ein R-Value

r-value:
Ein temporärer oder nicht-adressierbarer Wert, der nicht über den Ausdruck, der ihn verwendet, hinaus bestehen bleibt.

Merke: "Ein R value hat keinen Namen"

l-value:
Ein Ausdruck, der auf ein Objekt verweist, das einen dauerhaften Speicherort im Speicher hat und daher adressierbar ist.

Merke: "Ein L value hat einen Namen"

Begleitliteratur Tag 3

„Python Crashkurs“, 3. Auflage 2023, ISBN: 9783988901088

Tag 3 & 4:

- Kapitel 3: Listen
- Kapitel 4: Arbeiten mit Listen
- Kapitel 6: Dictionary

Aufgaben Tag 3

⚠️ **für Denkaufgaben (erstmal) kein GPT benutzen, nur für technische Fragen**

Aufgaben Dictionary: siehe Liste

Vokabular Tag 3

list,
dict (map)
tuple
set

Zustand (Menge aller Variablen und ihrer Werte)
Gültigkeitsbereich

>>> ctrl+l zum reinigen der Console (geht auch in System Console)

Vorschau Kontrollstrukturen

Kontrollstrukturen sind wichtige Bausteine in der Programmierung, insbesondere in Python. Sie ermöglichen es dem Programm, Entscheidungen zu treffen, Wiederholungen auszuführen und somit auf verschiedene Eingaben und Situationen flexibel zu reagieren. Hier sind die grundlegenden Kontrollstrukturen in Python:

Bedingte Ausführung "IF"

```
temperatur = 22
if temperatur > 25:
   print("Es ist warm")
```

"Wenn Temperatur größer 25 ist, sag es ist warm"

```
if Bedingung :
   Dann-führe-aus
```

Bedingte Ausführung "IF ELSE"

```
temperatur = get_sensor_data()
if temperatur > 25:
   print("Es ist warm")
else:
   print("Es ist kühl")
```

"Wenn Temperatur größer 25 ist, sag es ist warm, ANSONSTEN sag es ist kühl" **otherwise**

Vorschau Kontrollstrukturen

Kontrollstrukturen und Funktionen haben Programmcode in einem sogenannten Block

```
temperatur = 22
if temperatur > 25:
   print("Es ist warm")           # Block Befehl 1
   print("Es ist wirklich warm")  # Block Befehl 2
```

Dieser wird durch **Einrückung** von anderem Code abgegrenzt

```
temperatur = 22
if temperatur > 25:
   print("Es ist warm")           # Block Befehl 1
   print("Es ist wirklich warm")  # Block Befehl 2
print("nicht Teil des if Befehles")
```

Space oder Tab \t **Einrückung**

Vokabeln Tag 3

- map "Abbildung" ≈
- dict Dictionary "Wörterbuch" ≈
- key-value store Schlüssel-Wert-Paare

Vokabular Tag 3

list,
dict (map)
tuple
set

Zustand (Menge aller Variablen und ihrer Werte)
Gültigkeitsbereich

>>> ctrl+l zum reinigen der Console (geht auch in System Console)

Vorschau Kontrollstrukturen

Kontrollstrukturen sind wichtige Bausteine in der Programmierung, insbesondere in Python. Sie ermöglichen es dem Programm, Entscheidungen zu treffen, Wiederholungen auszuführen und somit auf verschiedene Eingaben und Situationen flexibel zu reagieren. Hier sind die grundlegenden Kontrollstrukturen in Python:

Bedingte Ausführung "IF"

```
temperatur = 22
if temperatur > 25:
   print("Es ist warm")
```

"Wenn Temperatur größer 25 ist, sag es ist warm"

```
if Bedingung :
   Dann-führe-aus
```

Bedingte Ausführung "IF ELSE"

```
temperatur = get_sensor_data()
if temperatur > 25:
   print("Es ist warm")
else:
   print("Es ist kühl")
```

"Wenn Temperatur größer 25 ist, sag es ist warm, ANSONSTEN sag es ist kühl" **otherwise**

Vorschau Kontrollstrukturen

Kontrollstrukturen und Funktionen haben Programmcode in einem sogenannten Block

```
temperatur = 22
if temperatur > 25:
   print("Es ist warm")           # Block Befehl 1
   print("Es ist wirklich warm")  # Block Befehl 2
```

Dieser wird durch **Einrückung** von anderem Code abgegrenzt

```
temperatur = 22
if temperatur > 25:
   print("Es ist warm")           # Block Befehl 1
   print("Es ist wirklich warm")  # Block Befehl 2
print("nicht Teil des if Befehles")
```

Space oder Tab \t **Einrückung**