Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2024-05 Dozent: Karsten Flügge

Themen des Tages

- **Dictionaries,** Mengen
- dict, set

- Speichern und Lesen in / aus Datei

Referenzen

Eine Variable ist eine Referenz auf ein Objekt

a = (1, 2, 3)

b = a # zeigt auf selbes Objekt (1, 2, 3)

a = "hallo" # das Tupel lebt jetzt in b weiter

```
>>> a
(1, 2, 3)
>>> b = a  # Referenz aufs selbe Objekt
(1, 2, 3)
>>> a == b
True
>>> a is b  # identisches Objekt!
True
>>> c = (1, 2, 3) # neues 'ähnliches' Objekt
>>> a == c  # selbe Einträge
True
>>> a is c  # nicht 'identisch'
False
```

Müllsammlung

Alte Objekte ohne Referenz werden gelegentlich vom Garbage Collection 'aufgeräumt'.

Automatisch! Keine Gedanken machen.

unveränderliche Objekte

Tuple (1,2,3) nicht veränderbar

Strings sind nicht veränderbar

Man kann das eigentliche Objekt nicht direkt modifizieren, aber man modifizierte Objekte daraus erzeugen.

Variablen kann man immer verändern.

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 kümmern zu müssen

Syntax: sets werden mit Geschweiften Klammern { } angelegt

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

Container

Liste [1,2,3] veränderbar

Tuple (1,2,3) nicht veränderbar

Set {1, 2, 3} keine Duplikate, *ungeordnet*, veränderbar

Dictionaries {a:b, c:d} veränderbar

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

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}    nicht + 😠 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.

Aufgaben

Aufgabe jetzt: experimentieren mit set

a) Erstelle zwei Sets von Zahlen und bilde die Vereinigung, Schnittmenge und Differenz

b) was ist Unterschied zwischen *difference* Methode und '-' ?

c) prüfe ob {1,2} in {2,3,4} enthalten ist (tip: dir() um passende Methode zu finden)

Extra:

d) Finde 4 Arten um aus einem Set ein neues zu erzeugen, in welchem ein Element fehlt

sets

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'}) # ungeordnet, keine Reihenfolge!
['b', 'a']
>>> set(["a","b","a"])
{'b', 'a'}
```

⚠️ **Achtung: Element ≠ Teilmenge**

a ∈ B ≠ a ⊂ B

a ∈ B <> a in B

a ⊂ B <> a.issubset(B)

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 vs set()
  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'
```

```
>>> translations["mouse"]
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    translations["mouse"]
    ~~~~~~~~~~~~^^^^^^^^^
KeyError: 'mouse'
```

dictionary[]

```
Wenn der Schlüssel eine Zahl ist, bezieht sich der index auf den Eintrag (nicht auf Nummer wie bei Listen)
>>> 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"

translations[alter_key]=neuer_wert
```

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:

- Nicht **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 ?

Während die **Werte <u>beliebig</u>** sein können

```
# 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)
   "variable" : abc             # Wert der Variable wird übernommen
}
```

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"
   }
}
person['wohnort']['plz']   # 12345
```

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!)**

Aufgaben

Erzeuge map von Hauptstädten fuer 3 Länder
a) füge 4tes Land hinzu
b) frage ein Land ab
c) überschreibe Hauptstadt von USA als "San Fran"
d) gebe Dict aus

Extra:
e) füge zwei maps zusammen
f) lösche ein Element
g) prüfe ob Eintrag vorkommt

⚠️ keys dürfen NICHT (Variablen)Namen ohne Tüttel sein:
⚠️ {name:"Frank", alter:32} darf man in python NICHT! stattdessen als string:

```
{"name":"Frank", "alter":32}  # ✅
```

Wann dictionaries?

SQL vs ad-hoc?

Große Dictionaries werden normalerweise entweder **aus Dateien** geladen,
oder aus Datenbanken abgerufen!

Im code ad-hoc Dictionaries für kleine Zuordnungen.

Für unstrukturierte Daten: für strukturierte Daten nehmen wir später besser **Klassen**.

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: Datenformat Austausch
{
  "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.

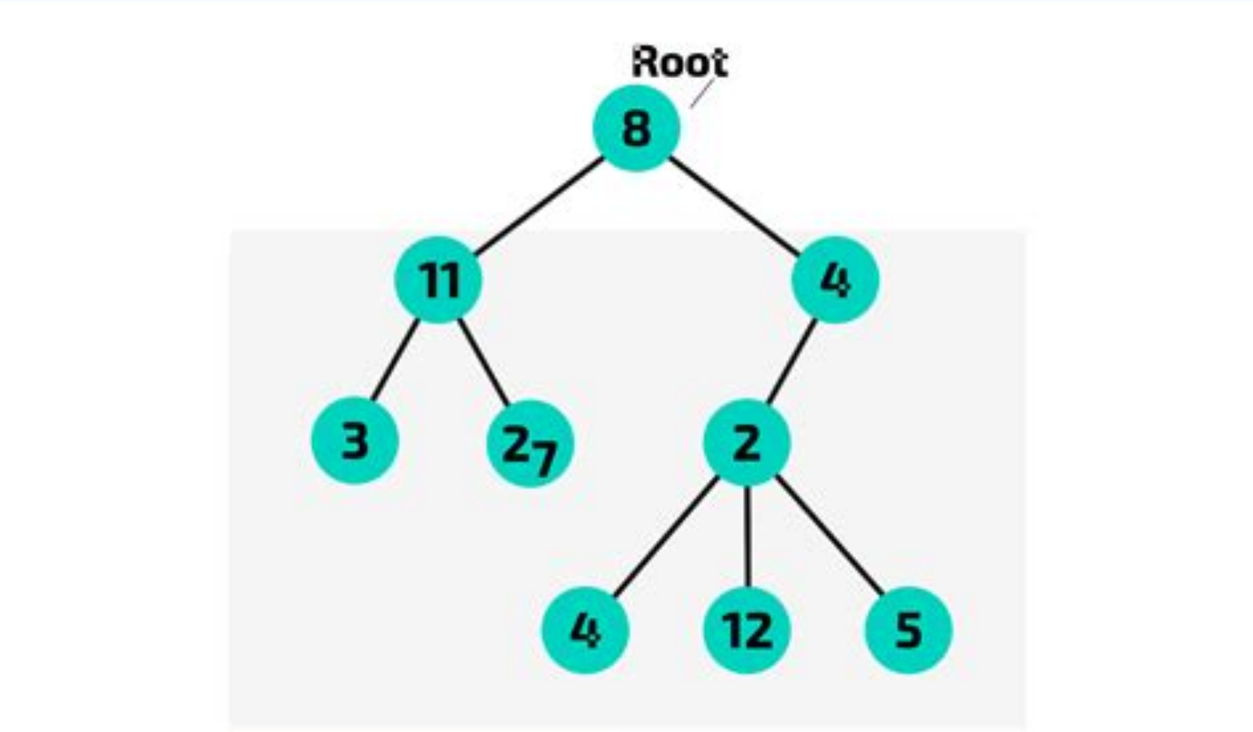
Verwendungszweck

- **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

```
tree={8:{11:{3,27},4:{2:{4,12,15}}}}
```



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)?

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"

Aufgaben Tag 4 (Freitag)

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

Aufgaben Dictionary: siehe Liste

Vokabular Tag 4 (Freitag)

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 4 (Freitag)

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

Vokabular Tag 4 (Freitag)

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**