Python Syntax

Vorkurs Bauingenieurwesen - Informatik

Rechnen

Zeichen	Beschreibung	Beispiel	Ergebnis
`+`	Plus	`4 + 5`	`9`
`-`	Minus	`12 - 7`	`5`
`*`	Mal	`3 * 6`	`18`
`/`	Geteilt (exakt)	`5 / 2`	`2.5`
`//`	Geteilt (abgerundet)	`5 // 2`	`2`
`%`	Rest	`5 % 2`	`1`
`**`	Hoch	`10 ** 3`	`1000`

Variablen

Eine Variable ist ein Name für ein Speicherort.

```
# lege einen neuen Speicherort mit name "friedhelm" an,
# speichere dort vorerst die Zahl 3.
friedhelm = 3

# überschreibe was an Speicherort "friedhelm" steht
# mit dem Ergebnis der Rechnung (also der Zahl 26)
friedhelm = 4 * 5 + 6

# speichere in "friedhelm" das was
# vorher dort gespeichert war plus 4 (also 30)
friedhelm = friedhelm + 4
```

Typen

Eine "Klasse- von Werten, die eine Variable annehmen kann, nennt man einen Typ. Wenn ein Typ von anderen Typen abhängt, schreibt man diese anderen Typen in eckigen Klammern hinter den Name des *generischen* Typen.
Will man den Typ einer Variablen `x` herausfinden, muss man `type(x)` schreiben. Beispielsweise wird `type(3.1415)` zu `float` ausgewertet.

- `int` enthält alle ganzen Zahlen, etwa `3` oder `-913781376139818372638713 67`
- `float` sind *Fließkommazahlen*, also Annährungen für relle Zahlen, etwa `3.0` oder `6.02214076e23`
- `bool` enthält nur die zwei Werte `True` und `False`
- `str` sind Zeichenketten, wie `"Hallo Freunde!"` oder `"123"`
- `list` sind Auflistungen mit Reihenfolge: `[1, 2, 3]` ist `list[int]`, `["Hi", "Ola"]` ist `list[str]` und `[[1], [2, 3]]` ist `list[list[int]]`
- `set` sind Auflistungen ohne definierbare Reihenfolge, wie `{1, 2, 3}` (vom exakten Typ `set[int]`) oder `{"Hallo", "Hi"}` (vom Typ `set[str]`)

- `dict` ist wie eine Liste, nur dass man mit beliebigen Typen statt nur `int` indizieren kann, etwa ist `{"1": 1, "2": 2, "?": 6}` vom Typ `dict[str, int]`, während der Typ `dict[str, list[int]]` den Wert `{"Tim": [0,-3], "Eva": [9], "Ulf": [1, 2, 3, 4]}` enthält
- `tuple` sind Auflistungen fester Länge von Werten möglicherweise unterschiedlicher Typen: der Wert `(1, 2)` hat den exakten Typ `(int, int)`, `(1.0, 1, "1", "eins")` hat den exakten Typ `(float, int, str, str)` und `([":)", "O.o"], {1, 2, 3, 8, 200, 201, 1341, 1343})` hat Typ `(list[str], set[int])`
- `NoneType` enthält exakt einen Wert: `None`

Entscheiden

Es gibt nur zwei Werte vom Typ `bool`: `True` und `False`. Diese sind aber höchst relevant um zu entscheiden ob das Programm eine bestimme Sache machen soll, oder wie oft etwas passiert (siehe Kapitel Kontrollfluss).

Zeichen	Beschreibung	Beispiel	Ergebnis
`==`	gleicher Wert	`"hallo" == "hallo"`	`True`
`!=`	ungleicher Wert	`"hello" != "Hallo"`	`True`
`<`	kleiner als	`4 < 5`	`True`
`>`	größer als	`4 > 5`	`False`
`<=`	kleiner oder gleich	`7 <= 7`	`True`
`>=`	größer oder gleich	`4 >= 5`	`False`
`not`	Gegenteil	`not True`	`False`
`and`	Und	`True and False`	`False`
`or`	Oder	`True or False`	`True`
`in`	ist enthalten	`9 in [4, 12, 9, 8]`	`True`

Kontrollfluss

Bedingte Ausführung

Grundsätzlich wird ein Programm immer Zeile für Zeile ausgeführt. Will man eine Zeile nur in bestimmten Fällen ausführen, nutzt man die `if`-Bedingung.
Nur, wenn der Ausdruck nach `if` und vor `:` zum Wert `True` ausgewertet wird, werden die weiter eingerückten Anweisungen darunter ausgeführt.

```
if 3 > 4:
    print("Hilfe, die Mathematik ist kaputt!")
    print("ruft die Feuerwehr!")
print("Diesen Text sehen wir immer.")
```

Ausgabe:

```
Diesen Text sehen wir immer.
```

Wenn die Bedingung Wahr ist mache das eine, wenn nicht das andere:

```
if 3 > 4:
    print("Hilfe, die Mathematik ist kaputt!")
    print("ruft die Feuerwehr!")
else:
    print("Puuh, die Mathematik ist noch heil.")
print("Diesen Text sehen wir immer.")
```

Ausgabe:

```
Puuh, die Mathematik ist noch heil.
Diesen Text sehen wir immer.
```

Unterscheide mehrere Einzelfälle:

```
if 3 == 4:
    print("Ganze Zahlen sind kaputt!")
elif 3.0 > 4.0:
    print("Ganze Zahlen sind heil.")
    print("Kommazahlen sind kaputt!")
elif "hallo" == "tschüss":
    print("Ganze Zahlen sind heil.")
    print("Kommazahlen sind heil.")
    print("Zeichenketten sind kaputt!")
else:
    print("Entwarnung")
print("Diesen Text sehen wir immer.")
```

Ausgabe:

```
Entwarnung
Diesen Text sehen wir immer.
```

Wiederholen von Anweisungen (Schleifen)

Will man *fast* die selben Anweisungen mehrfach ausführen, nutzt man Schleifen. Die `while`-Schleife funktioniert dabei zu Beginn wie eine `if`-Bedingung: Nur wenn der Logikausdruck zwischen `while` und `:` zum Wert `True` ausgewerted wird, springt das Programm in den weiter eingerückten Bereich und führt diese Zeilen aus.
Im Gegensatz zur `if`-Bedingung wird dieser Prozess nach Ausführung des eingerückten Bereiches wiederholt, bis der Logikausdruck das erste Mal `False` ist.

Version 1:

```
x = 2
while x < 6:
    print(x)
    x = x + 1
print("fertig!")
```

Version 2:

```
for x in range(2, 6):
    print(x)
print("fertig!")
```

Ausgabe:

```
2
3
4
5
fertig!
```

For-Schleifen

Eine `for`-Schleife kann Element für Element beliebiges *iterierbares* Objekt durchgehen. Beispiele für iterierbare Objekte sind Ausdrücke der Typen `str`, `range`, `list`, `set` oder `dict`. Die erste Zeile einer `for`-Schleife hat die Form `for x in xs:`, wobei `x` ein beliebiger gültiger Variablenname sein kann und `xs` ein beliebiger Ausdruck, der iterierbar ist.
Die weiter eingerückten Zeilen nach dieser ersten Zeile werden so oft ausgeführt, wie `xs` Einträge besitzt und für jede Ausführung hat die Variable `x` den Wert eines anderen Eintrages.

```
xs = [1, 2, 3, 4, 5]
print("xs = [1, 2, 3, 4, 5]")
for x in xs:
    print("x =", x)
print() #eine Zeile Platz

ys = range(4)
print("ys = range(4)")
for y in ys:
    print("y =", y)
print() #eine Zeile Platz

zs = { "Elli", "Hans" }
print("zs = { \"Elli\", \"Hans\" }")
for z in zs:
    print("z =", z)
```

Ausgabe:

```
xs = [1, 2, 3, 4, 5]
x = 1
x = 2
x = 3
x = 4
x = 5

ys = range(4)
y = 0
y = 1
y = 2
y = 3

zs = { "Elli", "Hans" }
z = Elli
z = Hans
```

Überspringen einer Schleifeniteration

Möchte man eine Schleifeniteration vorzeitig beenden und sofort die nächste Iteration starten, nutzt man `continue`. Trifft Python während der Codeausführung auf eine Zeile mit `continue`, springt das Programm hinter die letzte Zeile der nächstäußeren Schleife, testet also im nächsten Schritt ob die `while`-Schleife noch ein Mal wiederholt werden soll, bzw. ob es einen weiteren Wert gibt, für den man die `for`-Schleife ausführen kann.

Version 0:

```
x = 1
while x <= 6:
    x = x + 1
    if x != 3:
        print(x)
print("fertig!")
```

Version 1:

```
x = 1
while x <= 6:
    x = x + 1
    if x == 3:
        continue
    print(x)
print("fertig!")
```

Version 2:

```
for x in range(2, 6):
    if x == 3:
        continue
    print(x)
print("fertig!")
```

Ausgabe:

```
2
4
5
fertig!
```

Abbruch einer Schleifenausführung

Möchte man eine Schleife vorzeitig abbrechen, nutzt man `break`. Trifft Python während des Ausführens eines Programms auf eine Zeile die `break` enthält, wird die nächstäußere Schleife abgebrochen und in der ersten Zeile nach dieser Schleife weitergemacht.

Version 0:

```
x = 2
weiter = True
while weiter:
    print(x)
    x = x + 1
    if x == 6:
        weiter = False
print("fertig!")
```

Version 1:

```
x = 2
while True:
    print(x)
    x = x + 1
    if x == 6:
        break
print("fertig!")
```

Version 2:

```
for x in range(2, 100):
    if x == 6:
        break
    print(x)
print("fertig!")
```

Ausgabe:

```
2
3
4
5
fertig!
```

Funktionen

Die meisten Funktionen beim Programmieren sind von Außen identisch zu mathematischen Funktionen. Es wird ein (oder mehrere) Argument(e) von der Definitionsmenge auf die Bildmenge abgebildet. Definitionsmenge und Bildmenge sind in einem Programm bestimmte Typen. Als Beispiel wird die selbe Funktion f in der Mathematik geschrieben als

$$f\colon \mathbb{Z} \to \mathbb{Z}, \quad x \mapsto 2 \cdot x + 5$$

und in Python als

```
def f(x: int) -> int:
    return 2 * x + 5
```

Die Menge der ganzen Zahlen $\mathbb{Z}$ entspricht in Python dem Typ `int`.
Eine äquivalente Definition von f in Python wäre

```
def f(x: int) -> int:
    a = 2 * x # der name a existiert nur, solange f berechnet wird
    b = a + 5 # der name b existiert nur, solange f berechnet wird
    return b # der wert von b wird zurückgegeben
```

Die Variablen `a` und `b` existieren nur, bis der Computer an der Zeile `return b` ankommt. Erreicht der Computer eine Zeile die mit `return` beginnt, gilt die Funktion als fertig berechnet und der Ausdruck rechts davon (in diesem Fall `b`) wird ausgewertet und als Funktionswert zurückgegeben. Das Schlüsselwort `def` zeigt dem Computer an, dass an dieser Stelle eine neue Funktion beginnt.

So wie Python den Ausdruck `3 * 2 + 4` zu dem Wert `10` auswertet, kann auch die Funktion `f` angewendet werden. Der Ausdruck `f(7) - 1` etwa wird zu `18` berechnet.
In einer Funktion selbst kann eine beliebige Abfolge von Anweisungen ausgeführt werden:

```
def fakultaet(n: int) -> int:
    ergebnis = 1 # diese Variable existiert nur, solange fakultaet berechnet wird.
    while n > 0:
        ergebnis *= n
        n -= 1
    return ergebnis # beende Funktion und gebe ergebnis zurück
    n = 7 # diese Zeile wird nie ausgeführt
```

Die Funktion `fakultatet` berechnet die Fakultät einer Zahl. In Matheschreibweise ist die Fakultät (geschrieben mit einem Ausrufezeichen) definiert als

$$n! = n \cdot (n-1) \cdot \ \ldots \ \cdot 3 \cdot 2 \cdot 1$$

Listen

Möchte man eine ganze Gruppe von Werten zusammen speichern, etwa alle Namen von Leuten in einer Klasse oder alle Primzahlen kleiner als 100 nutzt man eine Liste. Eine Liste kann beliebig viele Einträge speichern. Jeder Eintrag hat dabei eine feste Position, der erste Eintrag hat Position 0.
Der Folgende Code produziert die Ausgabe `b` in der Konsole.

```
x = ["a", "b", "c", "d", "e", "f", "g"]
print(x[1])
```

Die eckigen Klammern haben dabei zwei verschiedene Rollen. Zuerst zeigen sie an, wo eine neue Liste anfängt und aufhört. Diese neue Liste kriegt den Name `x`. Wenn man hinter eine bestehende Liste (hier `x`) eckige Klammern schreibt, greift man auf Einträge der Liste zu. Im Beispiel wird der Eintrag an Index `1` gelesen, also `"b"`. Weitere Varianten sind

- `x[1:3]` erzeugt eine neue Liste die die Werte von `x` an `x`' Indices `1` und `2` hat, also `["b", "c"]`.
- `x[:3]` erzeugt eine neue Liste, die gleich beginnt wie `x`, aber schon nach den ersten `3` Einträgen aufhört, also `["a", "b", "c"]`.
- `x[1:]` erzeuge eine neue Liste, die gleich endet wie `x`, aber den ersten Eintrag von `x` nicht enthält, also `["b", "c", "d", "e", "f", "g"]`.
- `x[::2]` erzeuge eine Liste, die jeden zweiten Eintrag von `x` enthält, beginnend am Anfang, also `["a", "c", "e", "g"]`.
- `x[-1]` gibt den letzen Eintrag von `x` zurück, also `"g"`.
- `x[-2]` gibt den vorletzen Eintrag von `x` zurück, also `"f"`.

Möchte man eine neue Liste erzeugen, gibt es verschiedene Möglichkeiten

- explizit hinschreiben: `[1, 2, 3, 4]`.
- kleinere Listen aneinanderhängen: `[0, 2] + [1, 1, 1]` ergibt `[0, 2, 1, 1, 1]`.
- Eine kleinere Liste wiederholen: `5 * [0, 1]` ergibt `[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]`.
- Etwas anderes (iterierbares, siehe `for`-Schleifen) in eine Liste umwandeln: `list("abcde")` ergibt `["a", "b", "c", "d", "e"]`.

Weitere wichtige Operationen mit einer Liste `x` sind

- Ein neuen Eintrag hinten dran hängen: `x.append(6)`
- Den letzten Eintrag entfernen: `x.pop()`
- Den Eintrag an Index `3` überschreiben: `x[3] = "Lelele"`
- Alle Einträge löschen: `x.clear()`