4. Variablen

Variablen sind ein fundamentales Konzept in Programmiersprachen, da sie es ermöglichen, Daten temporär im Speicher zu speichern und später wieder abzurufen. Ohne Variablen könnten Programme keine Eingaben verarbeiten, Zwischenergebnisse speichern oder auf veränderte Daten reagieren. Sie fungieren als Container für Werte, die sich während der Programmausführung ändern können, was die Grundlage für dynamische und interaktive Programme bildet. Variablen machen Programme flexibel und wiederverwendbar, da derselbe Code mit unterschiedlichen Daten arbeiten kann. Darüber hinaus verbessern sie die Lesbarkeit und Wartbarkeit von Code, indem sie aussagekräftige Namen für Daten bereitstellen.

In Python sind folgende Datentypen für Variablen möglich:

- `int` für Ganzzahlen
- `float` für Fließkommazahlen
- `string` für Zeichenketten
- `boolean` für Wahrheitswerte
- `list` für Listen
- `tuple` für Tupel

Beispiel

```
1 note = 123
2 print(note)
```

```
123
```

Syntax

> **Syntax**
>
> ```
> Bezeichner = Ausdruck
> ```

Bemerkungen

- Python ist eine dynamische Sprache, d.h. Variablen müssen nicht deklariert werden. Der Typ einer Variablen wird durch den Wert bestimmt, der ihr zugewiesen wird.
- Pro Zeile nur eine Anweisung!
- linke Seite: Bezeichner = Variablennamen = Identifier

- rechte Seite: Ausdruck oder ein Wert
- Gleichheitszeichen ist der Zuweisungsoperator
- Variablen sind erst nach einer Anweisung verwendbar.
- Erst wird die rechte Seite ausgewertet und dann an die Variable zugewiesen.

> ⚠ Achtung
>
> Das Gleichheitszeichen ist der Zuweisungsoperator und nicht das mathematische Gleichheitszeichen (Vergleichsoperator).

Regeln für die Wahl des Bezeichners

- Buchstaben, Unterstriche und Ziffern
- erste Zeichen keine Ziffer
- keine Leerzeichen
- keine Python Schlüsselwörter (if, else, return, class, or,…)
- Python ist case-sensitive, d.h. Groß- und Kleinschreibung unterscheiden sich.

Ausdrücke

Ausdrücke können sein: - Operatoren - Literalen - Variablen

Die Auswertung eines Ausdrucks liefert einen Wert oder bricht mit Fehlermeldung ab. Die Auswertung bei arithmetischen Ausdrücken läuft nach folgenden Regeln ab:

1. Klammern zuerst
2. Potenzen
3. Multiplikation / Division
4. Addition / Subtraktion
5. ansonsten von links nach rechts.

Beispiele:

Zuweisung von Werten an Variablen

```
1 note = 4
```

Ausgabe der Variablen

```
1 print(note)
```

```
4
```

Mit Variablen rechnen

```
1 print(2*4**note)
```

```
512
```

Dynamische Typisierung

```
durchschnitt = 12.3
print(durchschnitt)
```

```
12.3
```

Rechnen mit unterschiedkichen Zahldatentypen

```
print(durchschnitt * note)
```

```
49.2
```

Fehler bei nicht definierten Variablen

```
print(goody)
```

```
NameError: name 'goody' is not defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[60], line 1
----> 1 print(goody)
NameError: name 'goody' is not defined
```

Strings sind Zeichenketten, die in Anführungszeichen stehen. Sie können mit Variablen verknüpft werden.

```
print("ham")
```

```
ham
```

analog geht das auch mit einfachen Anführungszeichen

```
print('egg')
```

> **Info**
>
> Der PEP 8 (Python Style Guide) empfiehlt, Strings in einfachen Anführungszeichen zu schreiben.

Der +-Operator wird bei Strings für die Verkettung von Strings (Konkatenation)verwendet.

```
print('ham'+'egg')
```

```
hamegg
```

Stings lassen sich in Python auch mit ganzen Zahlen über den *-Operator verknüpfen, was zu einer Wiederholung des Strings führt.

```
::: {.cell}
``` {.python .cell-code}
print('ham'*3)
```

hamhamham

:::
```

```
print('ham'*1.5)
```

```
TypeError: can't multiply sequence by non-int of type 'float'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[21], line 1
----> 1 print('ham'*1.5)
TypeError: can't multiply sequence by non-int of type 'float'
```

Mit `asserts` wird überprüft, ob der Wert eines Strings einem vorgegebenen Wert entspricht und wird für Tests verwendet. Stimmmt der Wert überein, passiert nichts, ansonsten wird eine Fehlermeldung ausgegeben.

```
assert 0* 'egg' == ""
```

```
assert 0* 'egg' == 'egg'
```

```
AssertionError:
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[24], line 1
----> 1 assert 0* 'egg' == 'egg'
AssertionError:
```

Python ist Case-Sensitive.

```
Number = 10
print(Number)
```

```
10
```

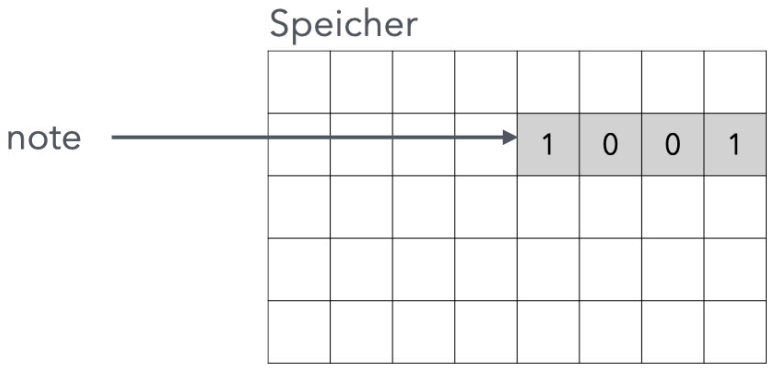
```
print(number)
```

```
NameError: name 'number' is not defined
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[26], line 1
----> 1 print(number)
NameError: name 'number' is not defined
```

Informationstechnische Betrachtung

Der Speicher des Computers lässt sich mit einem Tassenschrank vergleichen, in dem jede Tasse für eine Variable steht. In jede Tasse kann ein Wert gelegt werden und der Name der Tasse ist der Bezeichner der Variablen.

Genauer betrachtet wird bei der Zuweisung von Variablen im Speicher ein Bereich reserviert, der den Wert der Variablen als Binärzahl speichert. Der Variablenname (Bezeichner) ist ein daher der Zeiger auf diesen Bereich.



Die Speicheradresse lässt sich mit der id-Funktion ermitteln.

```
1 id(note)
```

```
4379198056
```

> Interpretation der id() Funktion
>
> Die `id()` Funktion gibt eine **eindeutige Ganzzahl** zurück, die das Objekt im Speicher identifiziert.
> **Was bedeutet diese Zahl?** - Die Zahl ist die **Speicheradresse** des Objekts (in CPython) - Jedes Objekt hat zur Laufzeit eine eindeutige ID - Solange das Objekt existiert, bleibt die ID konstant - Nach dem Löschen des Objekts kann die ID wiederverwendet werden

Beispiel-Interpretation:

```
x = 42
print(id(x))  # Ausgabe z.B.: 140712345678912
```

Die Zahl 140712345678912 ist die **Hexadezimal-Adresse im Arbeitsspeicher**, wo Python das Objekt mit dem Wert 42 gespeichert hat.
Wichtige Erkenntnisse: - Gleiche ID = Gleiches Objekt im Speicher - Verschiedene ID = Verschiedene Objekte im Speicher - Die absolute Zahl ist nicht wichtig, nur der Vergleich zwischen IDs

Wird ein weiterer Bezeichner auf die gleiche Variable eingeführt, so referenziert dieser bei den einfachen Datentypen (int float) nicht den gleichen Speicherbereich, sondern erhält eine Wertkopie. Damit ändert sich der Wert der ersten Variablen nicht, wenn man den Wert der zweiten Variablen ändert.

```
test1 = 10
print(test1)
```

```
10
```

```
test2=test1
print(test2)
```

```
10
```

```
id(test1)
```

```
4379194440
```

```
id(test2)
```

```
4379194440
```

```
test2=9
print(test2)
```

```
9
```

```
id(test2)
```

```
4379194408
```

```
print(test1)
```

```
10
```

> **!** Erklärung
>
> Python verwendet **Integer Caching** (Small Integer Optimization) für kleine Ganzzahlen (typischerweise -5 bis 256). Python speichert die kleinen Ganzzahlen weil sie häufig verwendet werden, nur einmal im Speicher und lässt alle Variablen mit demselben Wert auf dasselbe Objekt zeigen. Sobald Sie test2 einen neuen Wert zuweisen, zeigt es auf ein anderes gecachtes Objekt und erhält eine neue ID, während test1 weiterhin auf das ursprüngliche Objekt für den Wert 10 zeigt. Om Detail bedeudet dies:
>
> - Bei `test1 = 10` wird das gecachte Objekt für 10 verwendet
> - Bei `test2 = test1` zeigt test2 auf dasselbe gecachte Objekt gleiche ID
> - Bei `test2 = 9` wird test2 auf das gecachte Objekt für 9 umgeleitet neue ID
>
> **Wichtig:** Obwohl beide Variablen zunächst die gleiche ID haben, sind sie konzeptionell unabhängig. Eine Änderung von test2 beeinflusst test1 nicht, da bei der Neuzuweisung ein anderes Objekt referenziert wird.
> Dies ist anders als bei komplexeren Datentypen wie Listen, wo echte Referenzen verwendet werden.

Und wie sieht das bei Fließkommazahlen aus?

```
1 test3 = 1334.2
2 id(test3)
```

```
4420604848
```

```
1 test4=test3
2 id(test4)
```

```
4420604848
```

Bei `float`-Werten gibt es natürlich kein Integer Caching wie bei kleinen Ganzzahlen. Trotzdem haben `test3` und `test4` die gleiche ID, weil es sich um ein immutable Objekt handelt. Dieses Verhalten ist bei allen unveränderlichen (immutable) Datentypen in Python gleich.

> **i** Definition
>
> **Immutable** (unveränderlich) bedeutet, dass der Wert eines Objekts nach seiner Erstellung nicht mehr verändert werden kann.

Immutable Datentypen in Python: - `int` (Ganzzahlen) - `float` (Fließkommazahlen) - `str` (Strings/Zeichenketten) - `tuple` (Tupel) - `bool` (Wahrheitswerte) - `frozenset` (unveränderliche Mengen)

Mutable Datentypen in Python: - `list` (Listen) - `dict` (Dictionaries/Wörterbücher) - `set` (Mengen)

> **!** Wichtig
>
> Bei immutable Datentypen wird bei jeder “Änderung” ein neues Objekt erstellt!

Zustand eines Programms nach Ausführung wird vollständig beschrieben durch die Belegung aller Variablen mit Werten.