



Ugh, today's kids are forgetting the old-fashioned art of absentmindedly reading the same half-page of a book over and over and then letting your attention wander and picking up another book.

## 5. Schleifen

Mit einer **Schleife** kann Python die Wiederholung von Befehlen direkt übernehmen, ohne dass wir die Befehle mehrmals eintippen müssen. Die Lernziele für dieses Kapitel sind:

- ☐ Sie verwenden eine **for**-Schleife, um Befehle wiederholt auszuführen.
- ☐ Sie setzen die **range**-Funktion im Schleifenkopf ein.
- ☐ Sie verwenden **import**-Aliasing für eine noch kompaktere Notation.

### 5.1 Quadrat ultra kompakt

Die bisherigen „Turtle-Programme“ waren meist sehr repetitiv. Schon im Programm für das Quadrat der Seitenlänge 100 (Listing 1) wiederholen sich die Funktionsaufrufe `fd(100)` und `lt(90)` jeweils viermal. Wir können nun ein Programm in Python erstellen (siehe Listing 10), um diese beiden Funktionsaufrufe durch Python viermal wiederholen zu lassen.

```

1  import turtle
2
3  for _ in range(0, 1, 2, 3):
4      turtle.fd(100)
5      turtle.lt(90)
6  turtle.done()
7

```

Listing 10: Die Zeilen 4 und 5 werden jeweils viermal ausgeführt (`quadrat_loop.py`).

**H** Das Zeichen `_` soll verdeutlichen, dass hier eine Einrückung mit der **Tabulatortaste** (kurz: **Tab**) erfolgen muss.

Die **for**-Schleife besteht aus zwei Teilen (Schleifenkopf und Schleifenkörper):

- **Schleifenkopf:** **for** und **in** sind Python-Schlüsselwörter und wir müssen diese exakt so notieren. Dazwischen notieren wir einen **Underscore** (Unterstrich/Bodenstrich). Danach notieren wir in den **eckigen Klammern** so viele Werte, wie Wiederholungen durchgeführt werden sollen. Den Schleifenkopf beenden wir mit einem **Doppelpunkt**.

■ **Beispiel 13** In Listing 10 befindet sich der Schleifenkopf in Zeile 3. In den eckigen Klammern stehen vier Werte, es werden also vier Wiederholungen durchgeführt, wenn wir das Programm ausführen.

- **Schleifenkörper:** In diesem Schleifenteil notieren wir die Befehle, die wiederholt ausgeführt werden sollen. Diese Befehle müssen wir gleichmässig **einrücken**. Alle eingerückten Befehle müssen den gleichen Abstand zum „Rand“ haben. Die Einrückung dient **nicht** der optischen „Verschönerung“. Nur durch die Einrückung (engl. indentation) weiss Python, welche Befehle wiederholt werden müssen. Die eingerückten Befehle bilden den Schleifenkörper.

■ **Beispiel 14** In Listing 10 bilden die Zeilen 4 und 5 den Schleifenkörper, da diese Zeilen eingerückt sind. Die Zeile 6 gehört nicht mehr dazu, da sie nicht mehr eingerückt ist. ■

 **Clean Code** — **Leerzeichen 3.** In den **eckigen Klammern** wird **nach** jedem **Komma** ein **Leerzeichen** eingefügt.

## 5.2 Typische Fehlerquellen

Bei der Programmierung ist auf die folgenden typischen Fehler zu achten:

- Schlüsselwörter falsch geschrieben.
- Doppelpunkt am Ende des Schleifenkopfs vergessen.
- Schleifenkörper ist falsch eingerückt.

Listing 11 zeigt ein fehlerhaftes Programm mit **drei Fehlern**. In Zeile 3 ist das Schlüsselwort **IN** falsch geschrieben und der Doppelpunkt am Ende der Zeile fehlt. In Zeile 5 ist die Einrückung falsch. Der Befehl muss um ein Leerzeichen nach rechts eingerückt werden (oder noch besser: Tabulatortaste verwenden).

```
1 import turtle
2
3 for _ IN [0, 1, 2, 3]
4     turtle.fd(100)
5     turtle.lt(90)
6 turtle.done()
7
```

Listing 11: Achtung, dies ist ein **fehlerhaftes** Programm.

 **Clean Code** — **Einrückung 1.** Die Einrückung (engl. indentation) des **Schleifenkörpers** erfolgt mit der Tabulatortaste.

 **Clean Code** — **Leerzeichen 4.** Nach dem Schlüsselwort **in** (im Schleifenkopf) fügen wir ein **Leerzeichen** ein.

## 5.3 range-Funktion

Wir können den Schleifenkopf etwas kompakter gestalten, indem wir die eckigen Klammern samt Inhalt durch einen **range**-Funktionsaufruf ersetzen. Listing 13 zeigt die Verwendung der **eingebauten Funktion**. Wir können uns den Funktionsaufruf so vorstellen, als würden wir in den eckigen Klammern die Zahlen 0, 1, 2 und 3 notieren (siehe Listing 12).

```

1 import turtle
2
3 for _ in [0, 1, 2, 3]:
4     turtle.fd(100)
5     turtle.lt(90)
6 turtle.done()

```

Listing 12: Eingangsbeispiel

```

1 import turtle
2
3 for _ in range(4):
4     turtle.fd(100)
5     turtle.lt(90)
6 turtle.done()

```

Listing 13: Erste Verbesserung.

**Wichtig!** Der Funktionsaufruf von `range` erfolgt **ohne** ein davor notierter Modulname. Wir müssen auch kein Modul importieren, um den Funktionsaufruf einzusetzen. `range` ist eine **eingebaute Funktion** (eng. **built-in function**). Diese Funktionsaufrufe sind **immer** und **überall** (ohne Import) durchführbar. ■

## 5.4 import-Aliasing

Unsere Programme werden noch kompakter, wenn wir bei `import`-Anweisungen einen Alias (dt. Parallelbezeichnung) verwenden. Dadurch müssen wir nicht jedes Mal den vollständigen Modulnamen verwenden, sondern den von uns gewählten Alias.

■ **Beispiel 15** Listing 14 zeigt ein Beispiel mit einem `import`-Alias.

```

1 import turtle as t
2
3 for _ in range(4):
4     t.fd(100)
5     t.lt(90)
6 t.done()

```

Listing 14: Das Turtle-Modul erhält in Zeile 1 die Bezeichnung `t`. Danach können wir die Funktionsaufrufe aus diesem Modul nur noch mit dieser Bezeichnung verwenden. ■

■ **Beispiel 16** Listing 15 zeigt ein Programm mit zwei `import`-Anweisungen. Pro `import`-Anweisung wird ein `import`-Alias benutzt.

```

1 import turtle as t
2 import random as r
3
4 a = r.randrange(25, 101)
5 for _ in range(4):
6     t.fd(a)
7     t.lt(90)
8 t.done()

```

Listing 15: Zwei `import`-Anweisungen mit `import`-Aliasing. ■

Wir können für **jeden Modulimport** einen Alias benutzen. Dazu erweitern wir die `import`-Anweisung und verwenden am Ende das Schlüsselwort `as` mit einem gültigen Namen.

**Wichtig!** Zwei `import`-Anweisung dürfen **nicht** den gleichen Alias besitzen. ■