Einführung in das Programmieren

Kontrollstrukturen in python

Christoph Draxler, Vanessa Reichel
draxler@phonetik.uni-muenchen.de
vanessa.reichel@campus.lmu.de

7. November 2024

Programmablauf

Die Reihenfolge der Anweisungen bestimmt den Ablauf eines Programms.

- von oben nach unten
- ein Befehl nach dem anderen

In python startet man einen Programmlauf durch Ausführen von Befehlen oder einen Funktionsaufruf.

Im wirklichen Leben...

Es gibt Tätigkeiten und Aufgaben, die immer wieder ausgeführt werden müssen, z. B. Reifen flicken.

1. Rad abmontieren
2. Mantel einseitig abheben
3. Schlauch herausziehen
4. leicht aufpumpen
5. im Wasser undichte Stellen suchen und markieren
6. flicken und leicht aufpumpen
7. Schlauch einsetzen
8. Mantel in die Felge bringen
9. Rad montieren
10. fertig aufpumpen

Weil diese Folge von Arbeitsschritten häufiger vorkommt, lohnt es sich, sie unter einem Namen zusammenzufassen: 'Reifen_flicken' und festzuhalten. Damit erreichen Sie zweierlei: Sie können Ihr Wissen weitergeben, und Sie können sogar jemand anderes beauftragen:

Im wirklichen Leben...

Es gibt Tätigkeiten und Aufgaben, die immer wieder ausgeführt werden müssen, z. B. Reifen flicken.

1. Rad abmontieren
2. Mantel einseitig abheben
3. Schlauch herausziehen
4. leicht aufpumpen
5. im Wasser undichte Stellen suchen und markieren
6. flicken und leicht aufpumpen
7. Schlauch einsetzen
8. Mantel in die Felge bringen
9. Rad montieren
10. fertig aufpumpen

Weil diese Folge von Arbeitsschritten häufiger vorkommt, lohnt es sich, sie unter einem Namen zusammenzufassen: 'Reifen_flicken' und festzuhalten. Damit erreichen Sie zweierlei: Sie können Ihr Wissen weitergeben, und Sie können sogar jemand anderes beauftragen: "Schatzi, flick' doch bitte mein Fahrrad!"

Funktionsaufrufe im wirklichen Leben

Die Folge von Arbeitsschritten zum Flicken eines Reifens gilt prinzipiell für alle Fahrräder, unabhängig davon, ob gerade ein Rad geflickt wird oder nicht. Erst durch eine Anwendung der Schritte auf ein konkretes Fahrrad passiert auch etwas.

Funktionsaufrufe im wirklichen Leben

Die Folge von Arbeitsschritten zum Flicken eines Reifens gilt prinzipiell für alle Fahrräder, unabhängig davon, ob gerade ein Rad geflickt wird oder nicht. Erst durch eine Anwendung der Schritte auf ein konkretes Fahrrad passiert auch etwas.

Definition Zusammenfassung einzelner Arbeitsschritte unter einem Namen

Aufruf Anwendung der Arbeitsschritte auf ein konkretes Objekt

In Programmiersprachen bezeichnet man eine Folge von Einzelschritten unter einem Namen als *Funktion*. Die Funktions*definition* legt fest, welche Schritte ausgeführt werden sollen. Der Funktions*aufruf* führt die einzelnen Schritte mit konkreten Werten aus.

Probleme

Mit dem, was wir bislang gelernt haben sind Funktionen nichts weiter als Abkürzungen – statt viele einzelne Befehle auszuführen müssen wir nur eine Funktion aufrufen.
Das ist für wirklich sinnvolle Programme nicht ausreichend. Es fehlen

- Verzweigungen
- Wiederholungen
- Fehlerbehandlung

Erst mit diesen Konstrukten wird eine Programmiersprache brauchbar.

Fallunterscheidung

branching

Die *Fallunterscheidung* erlaubt eine *Verzweigung* im Programmablauf

```
if n == 0:
    ...
else:
    ...
```

In der `if`-Klausel steht eine Bedingung, hier `n == 0`. Ist sie erfüllt, dann wird der direkt folgende (und eingerückte!) Code ausgeführt. Ist sie nicht erfüllt, dann wird der Code im `else`-Teil ausgeführt.

Fallunterscheidung

In python schreibt man Fallunterscheidungen so:

- beginnt stets mit `if`
- dann folgt die Bedingung; sie kann komplex sein und in Klammern stehen
- danach kommt ein ':' und die nachfolgenden Zeilen werden nach rechts eingerückt
- mit (optionalem) `elif` bzw. `else` werden Alternativen angeboten

Fallunterscheidungen können geschachtelt werden

Fallunterscheidungen

Fallunterscheidungen mit mehr als zwei Verzweigungen

```
if n == 0:
    ...
else:
    if n == 1:
        ...
    else:
        if n == 2:
            ...
        else:
            ...
```

Fallunterscheidungen

Fallunterscheidungen mit mehr als zwei Verzweigungen

```
if n == 0:
   ...
else:
   if n == 1:
      ...
   else:
      if n == 2:
         ...
      else:
         ...
```

```
if n == 0:
   ...
elif n == 1:
   ...
elif n == 2:
   ...
else:
   ...
```

Die Schreibweise mit `elif` ist meist besser zu lesen, weil die Tiefe der Einrückung nach rechts gleich bleibt.

Komplexe Bedingungen

Häufig ist es übersichtlicher, eine komplexe Formel mit logischen Operatoren zu schreiben.

```
if n == 0 and m == 5:
  ...
```

ist äquivalent zu

```
if n == 0:
  if m == 5:
    ...
```

```
if n == 0 or n == 5:
  ...
```

ist äquivalent (wirklich?) zu

```
if n == 0:
  ...
elif n == 5:
  ...
```

Die logischen Operatoren sind `not` für *NICHT*, `or` für *ODER* und `and` für *UND*

Praxistipp

Fallunterscheidungen können Programme *sicherer* machen

- prüfen, ob ein Wert überhaupt zulässig ist
- sicherstellen, dass alle Fälle abgedeckt sind

Beispiel: Hörverlust

Im Wikipedia-Artikel zu 'Hörverlust' finden Sie eine Tabelle, die einen Prozentwert mit einer Beurteilung verknüpft.

Grad der Schwerhörigkeit nach dem Sprachaudiogramm	Hörverlust
geringgradige Schwerhörigkeit	20–40 %
mittelgradige Schwerhörigkeit	40–60 %
hochgradige Schwerhörigkeit	60–80 %
Resthörigkeit	80–95 %
Taubheit	100 %

de.wikipedia.org/wiki/Hörverlust

Beispiel für verschachtelte Fallunterscheidungen: Hörverlust

```
def hoerverlust (wert):
  if (wert >= 20 and wert < 40):
    return ("geringgradige Schwerhörigkeit")

  else:
    if (wert >= 40 and wert < 60):
      return ("mittelgradige Schwerhörigkeit")

    else:
      if (wert >= 60 and wert < 80):
        return ("hochgradige Schwerhörigkeit")

      else:
        if (wert >= 80 and wert < 95):
          return ("Resthörigkeit")

        else:
          return ("Taubheit")
```

Ist diese Funktion *sicher* und *korrekt*?

Verbesserte Version

```
def hoerverlust (wert):
   if wert < 0 or wert > 100:
      return "Ungültige Eingabe, es sind nur Werte zwischen 0 und 100 erlaubt."
   elif wert >= 0 and wert < 20:
      return "Alles in Ordnung!"
   elif wert == 20:
      return "Auf der Grenze zwischen 'Alles in Ordnung' und 'geringgradiger Hö
   elif wert > 20 and wert < 40:
      return "Geringgradiger Hörverlust."
   elif wert >= 40 and wert < 60:
      return "Mittelgradiger Hörverlust."
   elif wert >= 60 and wert < 80:
      return "Hochgradiger Hörverlust."
   elif wert >= 80 and wert < 95:
      return "Resthörigkeit."
   else:
      return "Taubheit."
```

Einfacher zu lesen, und zusätzliche Fälle um zu prüfen, ob die Eingabe überhaupt erlaubt ist. Auch der fehlende Fall (welcher ist das?) ist nun berücksichtigt!

Nochmal verbessert?

```
def hoerverlust (wert):
  if wert < 0:
    return "Fehler, Wert muss >= 0 sein!"
  elif wert < 20:
    return "Alles in Ordnung!"
  elif wert < 40:
    return "Geringgradige Schwerhörigkeit."
  elif wert < 60:
    return "Mittelgradige Schwerhörigkeit."
  elif wert < 80:
    return "Hochgradige Schwerhörigkeit."
  elif wert < 95:
    return "Resthörigkeit")
  elif wert <= 100:
    return "Taubheit"
  else:
    return "Fehler, Wert muss <= 100 sein."
```

- Funktioniert diese Version?

Nochmal verbessert?

```
def hoerverlust (wert):
  if wert < 0:
    return "Fehler, Wert muss >= 0 sein!"
  elif wert < 20:
    return "Alles in Ordnung!"
  elif wert < 40:
    return "Geringgradige Schwerhörigkeit."
  elif wert < 60:
    return "Mittelgradige Schwerhörigkeit."
  elif wert < 80:
    return "Hochgradige Schwerhörigkeit."
  elif wert < 95:
    return "Resthörigkeit")
  elif wert <= 100:
    return "Taubheit"
  else:
    return "Fehler, Wert muss <= 100 sein."
```

- Funktioniert diese Version?
- Worin besteht die Gefahr bei dieser Formulierung?

Letzte Verbesserung

Die Funktion soll den Wert nur berechnen und ihn *zurückgeben*.

```
def hoerverlust (prozent):
   if prozent < 0 or prozent > 100:
      befund = "Fehler: Wert muss zwischen 0 und 100 sein"
   elif prozent <= 20:
      befund = "Kein Befund, alles ok"
   elif prozent <= 40:
      befund = "Geringgradiger Hörverlust"
   elif prozent <= 60:
      befund = "Mittelgradiger Hörverlust"
   elif prozent <= 80:
      befund = "Hochgradiger Hörverlust"
   elif prozent <= 95:
      befund = "Resthörigkeit"
   else:
      befund = "Taubheit"

   return (befund)
```

`return()` steht als letzter Befehl in einer Funktion. Er beendet diese und gibt einen Wert zurück. Auf diese Weise lässt sich das Ergebnis z.B. in einen Text einbauen: `print("Der Befund lautet: "+ hoerverlust(55) + ".")`

Rekursion

Im Vorspann der Simpsons muss Bart jedes Mal eine Strafaufgabe erledigen: er muss einen Satz x-mal auf die Tafel schreiben.

```
def bartSimpson (n, text):
    if n == 0:
        print ("fertig!")
    else:
        print(text)
        bartSimpson (n-1, text)
```

Was passiert bei einem Aufruf `bartSimpson (10, 'Ich muss meine Hausaufgaben machen!')`?

Rekursion

Mit Rekursion lassen sich viele Aufgaben sehr elegant lösen

- mathematische Berechnungen, z.B. von Folgen
- Suchverfahren, z.B. in Zeichenketten
- Sortierverfahren, z.B. für Mengen
- Traversierungsverfahren, z.B. für Netzwerke

Rekursive Funktionsdefinitionen sind oft sehr nah an den entsprechenden mathematischen Formulierungen.

When a function calls itself recursively, it creates a stack of pending operations that need to be resolved. These operations are not executed immediately but are held in memory until the base case is reached. Once the base case is hit, the stack starts to "unwind" (resolve), working its way back through the recursive calls.

Rekursion

Hier ein Beispiel der mathematischen Funktion 'Fakultät'. In Wikipedia finden Sie folgende Definition:

$$n! = \begin{cases} 1 & : \quad n = 0 \\ n \cdot (n-1)! & : \quad n > 0 \end{cases}$$

Was berechnet diese Funktion?

Rekursion

In python kann man diese Definition fast genauso schreiben:

```
def fakultaet (n):
    if n == 0:
        return 1
    else:
        return n * fakultaet (n - 1)
```

Testen Sie: was ergeben `fakultaet(5)` bzw. `fakultaet(10)`?

Wie komme ich zu einer rekursiven Funktion?

1. Funktionskopf festlegen
2. Fallunterscheidung
   - 2.1 Terminalfall: welche Bedingung muss gelten, damit ich fertig bin?
   - 2.2 rekursiver Fall: mit welchen Argumentwerten mache ich weiter?
3. `return` gibt einen Funktionswert *zurück*, d. h. man kann das Ergebnis eines Aufrufs weiterverwenden, z. B. in einer Variable speichern

```
# 1) Kopf
def addieren (a, b):
    # 2) Fallunterscheidung
    if a == 0:
        # 2.1) wenn a gleich 0 ist,
        # kann ich nichts mehr addieren,
        # ich bin fertig und gebe b zurück
        return b
    else:
        # 2.2 weitermachen:
        # addiere 1 zum Ergebnis der
        # addition von a - 1 und b
        return 1 + addieren (a-1, b)

#3 Ergebnis speichern und ausgeben
summe = addieren(17, 4)
print(f'Die Summe ist {summe}')
```

Aufgaben

Definieren Sie die folgenden Funktionen mit Rekursion:

- `summiere(von, bis)` berechnet die Summe der Zahlen im Intervall `von - bis`
- `quersumme(n)` berechnet die Quersumme der Zahl `n`
- `enthaeltZeichen(kette, zeichen)` prüft, ob die Zeichenkette `kette` das Zeichen `zeichen` enthält
- `enthaeltTeilstring(kette, teil)` prüft, ob die Zeichenkette `kette` die Zeichenkette `teil` enthält[1]

Tipp: die Schreibweise `x[0]` gibt den ersten Buchstaben der Zeichenkette `x` zurück, die Funktion `len(x)` die Länge der Zeichenkette `x`.

[1] für Fortgeschrittene!