Algorithmische Methoden 04: Kontrollstrukturen & Funktionen

Dr. Sven Naumann

WS 20/21

1 Bedingungen

Zu den wichtigsten Kontrollstrukturen in Programmiersprachen gehören **Schleifen** und **bedingte Anweisungen**. Da wir Schleifen bereits im letzten Kapitel kurz behandelt haben, gehen wir hier zunächst auf die bedingten Anweisungen ein.

Elementarer Bestandteil einer bedingten Anweisung ist ein (einfacher oder komplexer) Test, der auch als Prädikat oder Bedingung bezeichnet wird. Abhängig von dem Ergebnis dieses Tests wird eine Folge `F1` von Anweisungen ausgeführt oder auf ihre Ausführung verzichtet bzw. eine andere Folge `F2` von Anweisungen ausgeführt.

Vergleichsoperatoren

Zur Formulierung einfacher Bedingungen werden häufig Vergleichsoperatoren verwendet, von denen wir einige in dem Abschnitt über Zahlen bereits erwähnt hatten:

Operator	Bedeutung	Beispiel	Ausgabe
`<`	kleiner	`1 < 2`	`True`
`<=`	kleiner oder gleich	`1 <= 1`	`True`
`>`	größer	`1 > 2`	`False`
`>=`	größer oder gleich	`1 >= 1`	`True`
`==`	gleich	`1 == 1.1`	`False`
`!=`	ungleich	`1 != 1.0`	`False`

Achtung: `==` ist nicht zu verwechseln mit dem **Zuweisungsoperator** `=`

Allerdings erlauben diese Operatoren auch nicht-numerische Operanden:

- Booleans:

```
>>> True > False
True
>>> True < False
False
```

- Listen:

```
>>> [] < [1]
True
>>> [1] < [10]
True
>>> [1, 0] < [10]
True
```

- 'Größen' von Zeichenketten:

```
>>> 'a' > 'b'
False
>>> 'a' < 'b'
True
>>> 'A' > 'a'
False
>>> 'A' < 'a'
True
>>> 'Aa' > 'Aac'
False
>>> 'Aa' < 'Aac'
True
```

- Vergleich von Zeichenketten:

```
>>> auto = "bmw"
>>> auto == "bmw"
True
>>> auto == "audi"
False

>>> auto = "BMW"
>>> auto == "bmw"
False
>>> auto.lower == "bmw"
True

>>> "audi" != "bmw"
True
>>> "bmw" != "bmw"
False
```

Neben dem allgemeinen Vergleichoperator `==` , der prüft, ob beide Ausdrücke denselben Wert bezeichnen gibt es noch die Operatoren `is` und `is not` , die prüfen, ob beide Ausdrücke dasselbe Objekt bezeichnen oder nicht:

```
>>> 1 == 1.0
True
>>> 1 is 1.0
False
>>> 1 != 1.0
False
>>> 1 is not 1.0
True
```

Beachten Sie den Unterschied!

`and`, `or` & `not`

Zur Bildung komplexer Bedingungen kann man die logischen Operatoren `and`, `or` und `not` verwenden. Mit Hilfe runder Klammern lassen sich Ambiguitäten vermeiden:

`(B1 and B2)` evaluiert zu `True` **gdw.** (genau dann, wenn) beide Teilbedingungen erfüllt sind;
`(B1 or B2)` evaluiert zu `True` **wenn eine der beiden** Teilbedingung erfüllt ist und
`not(B)` kehrt den (Wahrheits)Wert um, zu dem `B` evaluiert.

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_0 >= 21 or age_1 >= 21
True
>>> not(age_0 >= 21 and age_1 >= 21)
True
```

Grundsätzlich gilt in Python: Eine Bedingung ist erfüllt, solange sie nicht zu `False` evaluiert.
Neben dem booleschen Wert False bezeichnen in Python auch

- die numerischen Nullwerte `0`, `0L`, `0.0`, `0.0+0.0j`;
- die leere Zeichenkette `""`;
- die leere Liste `[]` bzw. das leere Tupel `()`;
- leere Wörterbücher `{}` und
- der Wert `NONE` das logisch Falsche.

`in` & `not in`

Mit den Operatoren `in` und `not in` lässt sich prüfen, ob eine Sequenz (Liste, Tupel) ein bestimmtes Element (nicht) enthält:

```
>>> bundesliga = ['Schalke', 'Gladbach', 'Dortmund', 'Bayern', 'Bremen']
>>> print('Gladbach' in bundesliga)
True
```

Methoden und Funktionen für Zeichenketten

Zur Verarbeitung von Zeichenketten gibt es zahlreiche Prädikate, also Funktionen oder Methoden, die als Wert einen Wahrheitswert liefern:

- `.startswith(t)` prüft, ob die Zeichenkette `s` mit der Zeichenkette `t` beginnt
- `.endswith(t)` bzw. endet
- `t in s` prüft, ob die Zeichenkette `s` den Ausdruck `t` als Teilkette enthält
- `.islower()` prüft, ob die Zeichenkette nur aus **Kleinbuchstaben** besteht
- `.isupper()` prüft, ob die Zeichenkette nur aus **Großbuchstaben** besteht
- `.isalpha()` prüft, ob die Zeichenkette nur aus **Buchstaben/Wortzeichen** besteht
- `.isalnum()` prüft, ob die Zeichenkette nur aus **Ziffern & Buchstaben** besteht
- `.isdigit()` prüft, ob die Zeichenkette nur aus **Ziffern** besteht
- `.istitle()` prüft, ob die Zeichenkette mit einem **Großbuchstaben** beginnt

2 `if` this, `then` that or `else`

Die grundlegende bedingte Anweisung in praktisch allen bekannten Programmiersprachen ist die `if` - `then` - `else` -Anweisung. Sie besteht im einfachsten Fall neben den Schlüsselwörtern `if` und `then` aus einer Bedingung und einem Anweisungsblock. Die Anweisungen werden sequentiell ausgeführt, sofern die Bedingung erfüllt ist; anderenfalls geschieht nichts und die auf die `if` -Anweisung folgende Anweisung wird ausgeführt.

In Python wird auf das Schlüsselwort `then` verzichtet. Auf die Bedingung folgt einfach ein Doppelpunkt `:`

```
if Bedingung :
    Anweisungsblock
```

```
>>> alter = 19
>>> if alter >= 18:
... print("Sie sind alt genug, um zu wählen!“)
Sie sind alt genug, um zu wählen!
>>> if alter >= 18:
... print("Sie sind alt genug, um zu wählen!“)
... print("Welche Partei mögen Sie?“)
Sie sind alt genug, um zu wählen!
Welche Partei mögen Sie?
```

Natürlich ist es möglich eine Folge von Anweisungen zu spezifizieren, die ausgeführt wird, wenn die Bedingung nicht erfüllt ist:

```
if Bedingung :
    Anweisungsblock 1
else:
    Anweisungsblock 2
```

```
>>> alter = 17
>>> if alter >= 18:
... print("Sie sind alte genug, um zu wählen!“)
... print("Welche Partei mögen Sie?“)
... else:
... print("Sie sind leider noch zu jung.“)
Sie sind leider noch zu jung.
```

2 `if` this, `then` that or `else`

- Wenn es darum geht, komplexe Bedingung-Aktion-Strukturen zu modellieren, stellen viele Programmiersprachen spezielle Sprachkonstrukte (wie z.B. `SWITCH`-, `CASE`- oder `COND`-Anweisungen) zur Verfügung. Im Falle von Python muss man sich dagegen mit verschachtelten `if`-Anweisungen behelfen:

```
if Bedingung 1 :
    Anweisungsblock 1
elif Bedingung 2 :
    Anweisungsblock 2
(...)
elif Bedingung n :
    Anweisungsblock n
```

```
>>> alter = 17
>>> if alter < 6:
...     preis = 0
... elif age < 18:
...     preis = 5
... elif alter >= 18:
...     preis = 10
... print("Der Eintritt kostet Sie " + str(preis) + " €.")
Der Eintritt kostet Sie 5 €.
```

An Stelle des letzten `elif` kann auch ein `else` (ohne Bedingung) stehen:

```
>>> alter = 21
>>> if alter < 6:
...     preis = 0
... elif age < 18:
...     preis = 5
... else:
...     preis = 10
... print("Der Eintritt kostet Sie " + str(preis) + " € .")
Der Eintritt kostet Sie 10 €.
```

3 Listen und bedingte Anweisungen

Wir hatte in der letzten Sitzung gesehen, wie über alle Elemente einer Liste iteriert werden kann. Natürlich ist es möglich, die Verarbeitung an bestimmte Bedingungen zu knüpfen. In diesem Fall ist es naheliegend, Schleifen und bedingten Anweisungen miteinander zu kombinieren:

```
>>> lottozahlen = [3, 21, 33, 34, 39, 47, 49]
>>> mein_tippzettel = [9, 17, 33, 37, 39, 43, 49]
>>> treffer = 0
>>> for zahl in mein_tippzettel: # alle Zahlen des Tippzettels abarbeiten
>>>     if zahl in lottozahlen:
...         treffer += 1 # Zählen der Treffer
...     else:
...         print("Die Zahl "+ str(zahl) + " ist leider nicht gezogen worden.")
>>> print("Sie haben " + str(treffer) + " Richtige!")
Sie haben 3 Richtige!
```

Da eine leere Liste als `False` interpretiert wird, ist es einfach zu prüfen, ob eine Liste leer ist oder nicht:

```
>>> if mein_tippzettel: # Die Bedingung ist erfüllt, sofern die Liste nicht leer ist
...     (...)
... else:
...     print("Sie haben vergessen, ihren Tippzettel auszufüllen!")
```

Angenommen, die Sätze eines Textes werden durch Listen von Zeichenketten repräsentiert:

```
>>> satz1 = ['Call', 'me', 'Ishmael', '.']
>>> for wort in satz1:
...     if wort.endswith('l'): # Wir untersuchen, ob das Wort mit 'l' endet:
...         print(wort)
Call
Ishmael
```

Nun möchten wir für jedes Token eines Satzes prüfen, ob es mit einem Groß- oder Kleinbuchstaben beginnt oder einfach ein Interpunktionszeichen ist:

```
>>> for wort in satz1:
...     if wort.islower():
...         print(wort, 'ist ein kleingeschriebenes Wort')
...     elif wort.istitle():
...         print(wort, 'ist ein großgeschriebenes Wort')
...     else:
...         print(wort, 'ist ein Interpunktionszeichen')
Call ist ein großgeschriebenes Wort
me ist ein kleingeschriebenes Wort
Ishmael ist ein großgeschriebenes Wort
. ist ein Interpunktionszeichen
```

Natürlich können bedingte Anweisungen auch innerhalb einer Listenabstraktion (oder *list comprehension*) genutzt werden. Angenommen, wir haben einen als Liste von Token repräsentierten Text, der der Variablen `text2` zugewiesen wurde und wir möchten alle Wörter dieses Textes finden, die die Zeichenfolge `'cie'` oder `'cei'` enthalten, dann können wir diese Aufgabe mit dem folgenden kleinen Programm erledigen:

- Zunächst werden alle Duplikate via `set()` entfernt und dann die Menge der Token gefiltert:

```
>>> tricky = sorted(w for w in set(text2) if 'cie' in w or 'cei' in w)
```

- Anschließend werden alle Elemente dieser Menge ausgegeben:

```
>>> for wort in tricky:
    ... print(wort, end=' ')
ancient ceiling conceit conceited conceive conscience
conscientious conscientiously deceitful deceive ...
```

4 `while`-Schleifen

Neben den zählerkontrollierten `for`-Schleifen, die wir bereits kennengelernt haben, gibt es in Python wie in vielen anderen Programmiersprachen einen weiteren, bedingungskontrollierten Schleifentyp:
die `while`-Schleife. Wie der Name vermuten lässt, werden hier die Anweisungen des Anweisungsblocks immer wieder ausgeführt, solange eine bestimmte Bedingung erfüllt ist. Sie hat folgende Struktur:

```
while Bedingung:
    Anweisungsblock 1
else:
    Anweisungsblock 2
```

Der `else`-Teil ist optional.
Die folgenden Zeilen geben die Eingaben solange auf dem Bildschirm aus bis ein `q` eingegeben wird:

```
eingabe = ''
while eingabe != 'q':
    eingabe = input()
    print(eingabe)
```

In den meisten Fällen gibt es eine Variable (Schleifenvariable; hier `eingabe`), deren Wert in jedem Durchgang aktualisiert wird und die auch in der Abbruchbedingung genutzt wird. Man sollte unbedingt darauf achten, dass schließlich eine Situation eintritt, in der die Schleifenbedingung nicht mehr erfüllt ist; anderenfalls ist das Ergebnis eine **nicht-terminierende Schleife** (Endlosschleife):

```
>>> while True:
...     print(42)
42
42
∞
```

Eine `for`- oder `while`-Schleife kann mit Hilfe einer `break`-Anweisung (vorzeitig) verlassen werden:

```
>>> prompt = „\nGeben Sie bitte den Name einer Sängerin ein, die Sie mögen:“
>>> prompt += „\n(Zum Beenden bitte 'quit' eingeben.) "
>>> while True:
...     saengerin = input(prompt)
...     if saengerin == 'quit': # Wenn 'quit' eingegeben wird, wird die Schleife verlassen.
...         break
...     else:
...         print(„Ich höre auch gerne " + saengerin.title() + "!")
```

Statt die Schleife vollständig zu verlassen, kann es manchmal sinnvoll sein, die aktuelle Iteration bzw. Ausführung des Anweisungsblocks zu überspringen. Dazu dient die `continue`-Anweisung:

```
>>> zahl = 0
>>> while zahl < 10:
...     zahl += 1
...     if zahl % 2 != 0:
...         continue # Wenn zahl ungerade ist überspringe print() und kehre zum Anfang zurück
...     print(zahl)
```

Das Programm gibt die positiven einstelligen geraden Zahlen `2`, `4`, `6`, `8` aus.

Zum Abschluss dieses Abschnitts schauen wir uns noch ein paar einfache Programme an. Das erste Programm simuliert den Anmeldeprozess zu einer Veranstaltung:

- Zunächst gibt es eine Liste mit Interessenten und eine leere Liste, in die später die zugelassenen Personen eingetragen werden:

```
>>> interessenten = ['alice', 'brian', 'candace']
>>> zugelassene_personen = []
```

- Diese Liste wird dann abgearbeitet; d.h die Elemente nacheinander in die Liste der zugelassenen Personen übertragen:

```
>>> while interessenten:
...     aktuelle_person = interessenten.pop()
        # .pop() entfernt einen Listeneintrag und gibt ihn z.B. an eine Variable weiter
...     print("\nÜberprüfung von: " + aktuelle_person.title())
...     zugelassene_personen.append(aktuelle_person)
        # .append() fügt einer Liste einen Eintrag hinzu
...     print("\nDie folgenden Personen wurden zugelassen:“)
...     for person in zugelassene_personen:
...         print(person.title())
```

Output

```
Überprüfung von: Candace

Die folgenden Personen wurden zugelassen:
Candace

Überprüfung von: Brian

Die folgenden Personen wurden zugelassen:
Candace
Brian

Überprüfung von: Alice

Die folgenden Personen wurden zugelassen:
Candace
Brian
Alice
```

Im zweiten Programm geht es eine kulinarische Umfrage, bei der beliebte Gerichte gesucht werden:

```
umfrage = {}
umfrage_aktiv = True
while umfrage_aktiv:
    antwort = input("Nennen Sie bitte Ihr Lieblingsgericht: ")
    if umfrage.get(antwort):
        umfrage[antwort] += 1
    else:
        umfrage[antwort] = 1
    repeat = input("Soll eine weitere Person befragt werden (ja/nein)?")
    if repeat == 'nein':
        umfrage_aktiv = False
print("\n--- Umfrageergebnis ---")
for name, anzahl in umfrage.items():
    print(str(anzahl) + " Person(en) isst/essen gerne " + name + ".")
```

5 Benutzerdefinierte Funktionen

In den vorangegangen Sitzungen haben wir eine Reihe von vordefinierten Funktionen und Methoden kennengelernt. Natürlich können wir die Menge der in Python vordefinierten Funktionen erweitern, in dem wir neue Funktionen definieren. Grundsätzlich nimmt eine (mathematische) Funktion eine Anzahl von Argumenten, führt mit diesen Argumenten eine Berechnung durch und gibt den Wert dieser Berechnung zurück.

In Python wird dieses Konzept nicht vollständig (oder besser: **nicht konsequent**) umgesetzt: Bei der Definition einer Funktion wird - *vereinfacht gesagt* - einem Anweisungsblock ein Name bzw. Bezeichner zugeordnet. In den Anweisungen können die Parameter verwendet werden, die im Funktionskopf deklariert und beim Aufruf der Funktion an die Werte der Argumente gebunden wurden. Die Funktion kann, muss aber nicht einen Wert zurückgeben. Eine Funktion gibt in Python nur dann einen Wert zurück, wenn sie eine `return`-Anweisung enthält; sonst gibt sie `None` zurück. `None` ist das einzige Element der `NoneType`-Klasse und dient dazu, die Tatsache, dass die Funktion keinen Wert liefert, auszudrücken.

- Eine Funktionsdefinition hat in Python folgende Struktur:

```
def funktions-name(Parameterliste):
    Anweisungsblock
```

Die Anweisungen des Anweisungsblocks werden eingerückt.

Das Ende der Einrückung markiert das Ende des Anweisungsblocks.

Neben Funktionen kennt Python auch Methoden.

Auf die Unterschiede zwischen Funktionen und Methoden werden wir in der folgenden Sitzung eingehen.

Parameter und Werte

Grundsätzlich kann die Parameterliste einer Funktion auch leer sein.
Das leere Klammerpaar ist allerdings bei Definition wie Aufruf der Funktion obligatorisch:

```
>>> def hallo():
...     print('Hallo, Welt!') # Das Ende des Blocks wird nicht explizit markiert.
>>> hallo() # Aufruf der zuvor definierten Funktion
Hallo, Welt!
```

Die Funktion `hallo()` scheint einen Funktionswert zurückzugeben. Allerdings täuscht dieser Eindruck: Die Zeichenkette `'Hallo, Welt!'`, die ausgegeben wird, ist nicht der Funktionswert, sondern die Ausgabe der Zeichenkette ist ein durch die Funktion `print()` ausgelöster Seiteneffekt, was sich leicht zeigen lässt:

```
>>> type(hello())
Hallo, Welt!
<class 'NoneType'>
>>> len(hello())
TypeError: object of type 'NoneType' has no len()
```

Ablauf

- Wenn eine Funktion mehrere Werte zurückgeben soll, müssen diese Werte in einer Liste, einem Tupel oder Dictionary gespeichert werden. Dieses Objekt wird dann als Wert zurückgegeben.
- Wenn die Parameterliste einer Funktionsdefinition nicht leer ist, enthält sie ein oder mehrere durch Kommata getrennte Bezeichner.
- Beim Aufruf der Funktion werden die Anweisungen des Anweisungsblocks sequentiell ausgeführt.
- Der Aufruf terminiert, nachdem die letzte Anweisung ausgeführt wurde oder eine besondere Anweisung bzw. ein Fehler die Ausführung vorzeitig terminiert hat.

`return`

Zu den besonderen Anweisungen gehören `return`-Anweisungen:

- Nach der Ausführung einer `return`-Anweisung terminiert der Aufruf der Funktion sofort und als Funktionswert wird der Wert dieser Anweisung zurückgegeben.
- Folgt auf das `return` keine Anweisung oder enthält der Funktionskörper kein `return`, wird als Wert der spezielle Wert `None` zurückgegeben.
- Der Körper einer Funktion kann mehrere `return`-Anweisungen enthalten.

```
>>> def fahrenheit(T_in_celsius):
...     return (T_in_celsius * 9 / 5) + 32
>>> for t in (22.6, 25.8, 27.3):
...     print(t, ": ", fahrenheit(t))
22.6 : 72
25.8 : 78
27.3 : 81
```

Im Beispiel wird der Parameter `T_in_celsius` nacheinander an die Werte `22.6,` `25.8`, `27.3` gebunden.

Typen von Parametern

Im einfachsten Fall muss die Zahl der Argumente beim Aufruf einer Funktion mit der Zahl der Parameter übereinstimmen. In diesem Fall spricht man auch von positionalen Argumenten. Die Parameter werden an den Wert des jeweiligen Argumentes gebunden. Stimmt die Zahl der Argumente nicht mit der Zahl der Parameter überein, kommt es zu einer Fehlermeldung:

```
>>> def rechteck_flaeche(Seite_A, Seite_B):
...     return Seite_A * Seite_B
>>> rechteck_flaeche(3, 7) # Seite_A wird an 3 und Seite_B an 7 gebunden
21
```

```
>>> rechteck_flaeche(3)
TypeError: rechteck_flaeche() missing 1 required positional argument: 'Seite_B'
```

```
>>> rechteck_flaeche(3, 7, 1)
TypeError: rechteck_flaeche() takes 2 positional arguments but 3 were given
```

Allerdings stellt Python einen einfachen Mechanismus zur Verfügung, der es erlaubt, die Argumente beliebig anzuordnen. Durch Gleichungen der Form $\mathrm{Parameter}_i = \mathrm{Argument}_i$ können den Parametern der Funktion Argumente zugeordnet werden, ohne auf die bei der Definition der Funktion festgelegte Reihenfolge achten zu müssen. Ausdrücke dieser Form werden als `keyword`-Argumente bezeichnet. Allerdings müssen natürlich die bei der Definition der Funktion verwendeten Parameterbezeichner bekannt sein und verwendet werden:

```
>>> rechteck_flaeche(Seite_B=7, Seite_A=3)
21
```

Zu beachten ist, dass positionale Argumente beim Aufruf einer Funktion immer vor möglichen keyword-Argumenten stehen müssen:

```
>>> rechteck_flaeche(3, Seite_B=7)
21
>>> rechteck_flaeche(Seite_B=7, 3)
SyntaxError: positional argument follows keyword argument
```

Der gleiche Mechanismus kann bei der Definition einer Funktion verwendet werden, um für ein oder mehrere Parameter Standardwerte (default values) zu vereinbaren. Dadurch werden diese Parameter zu optionalen Parametern, die beim Aufruf der Funktion nicht mehr in jedem Fall spezifiziert werden müssen. Optionale Parameter müssen bei der Definition einer Funktion in der Parameterliste immer auf alle positionalen Parameter folgen:

```
>>> def rechteck_flaeche(Seite_A, Seite_B=0)
...     return Seite_A * Seite_B
>>> rechteck_flaeche(7)
0
>>> rechteck_flaeche(3, 7)
21
```

Wie auch in vielen anderen (Skript)Sprachen ist es möglich, Funktionen zu definieren, die eine beliebige Zahl von Argumenten akzeptieren: In einer Funktionsdefinition kann die Parameterliste als letzten Parameter einen Bezeichner enthalten, der mit `*` beginnt. Er wird beim Aufruf der Funktion an die Liste der Argumente gebunden, die nicht an andere Parameter gebunden werden konnten:

```
>>> def mittelwert(*werte):
...     sum = 0
...     for x in werte:
...         sum += x
...     return sum / len(werte)
>>> mittelwert(2,3,4,5,6,7)
4.5
```

Allerdings darf es sich bei den Argumenten nicht um `keyword` -Argumente handeln. Für sie gibt es aber auch eine Lösung: Der finale Parameter der Parameterliste wird mit einem `**` markiert:

```
def kochen(gericht, **zutaten):
    zutaten['Gericht'] = gericht
    return zutaten
rezept = kochen('pizza', Mehl='300 Gramm', Olivenöl='50 ml', Tomaten='4 Stück')
```

Output

```
>>> print(rezept)
{'Mehl': '300 Gramm', 'Olivenöl': '50 ml', 'Tomaten': '4 Stück', 'Gericht': 'pizza'}
```

Beide Sprachmittel lassen sich auch bei einem Funktionsaufruf verwenden. Der erzielte Effekt ist allerdings ein entgegengesetzter:
`*` bzw. `**` in einer Funktionsdefinition dienen dazu, eine Folge von Werten in eine Liste oder dictionary zusammenzuziehen; werden sie dagegen innerhalb eines Funktionsaufrufes genutzt, lösen sie die Elemente einer Liste bzw. eines dictionary aus dieser Datenstruktur heraus.

```
>>> werte = [2,3,4,5,6,7]
>>> mittelwert(werte)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 4, in mittelwert
TypeError: unsupported operand type(s) for +=: 'int' and 'list'...
>>> mittelwert(*werte)
4.5
>>> def rechteck(x, y):
...     return x * y
>>> rechteck(3, 4)
12
>>> werte = {'x': 3, 'y': 4}
>>> rechteck(**werte)
12
```

Variablen in Funktionen

Der Geltungsbereich (***scope***) einer Variablen `v` ist der Bereich eines Programms, innerhalb dessen sie referenzierbar ist; d.h. innerhalb dessen die Evaluierung des sie bezeichnenden Symbols nicht zu einer Fehlermeldung führt. Unter einer Umgebung (***environment***) versteht man die Menge von Wertbindungen, die zu einem bestimmten Zeitpunkt (bei der Ausführung eines Programms) bestehen. Wie bei den im Funktionskopf verwendeten Parametern ist auch der Geltungsbereich der innerhalb einer Funktionsdefinition deklarierten Variablen normalerweise die Funktionsdefinition selbst; d.h. es handelt sich bei ihnen um **lokale** (im Gegensatz dazu: *globale*) Variablen, auf die außerhalb der Funktion nicht zugegriffen werden kann:

```
>>> def test(a, b):
...     c = 3
...     return a + b + c
test(1, 2)
6
>>> a
NameError: name 'a' is not defined
>>> c
NameError: name 'c' is not defined
```

Anders formuliert: Die Funktionsdefinition erzeugt eine Umgebung, in der die Parameter- und Variablenbezeichner eine 'Bedeutung' haben; außerhalb dieser Umgebung sind sie ungebunden. Umgebungen können andere Umgebungen enthalten (*Matroschka-Prinzip*). Variablen, die direkt im Programm (Null-Umgebung) deklariert werden, können innerhalb des Programms und auch den anschließend definierten Funktionen genutzt werden; d.h. sie verhalten sich wie **globale** Variablen:

```
>>> c = 9
>>> def test(a, b):
...     return a + b + c
>>> test(1, 2)
12
```

Wenn es eine gleichlautende Variable (oder Parameter) in Funktion und der sie umschließenden Umgebung gibt, verschattet die Variable in der Funktionsdefinition die andere Variable:

```
>>> c = 9
>>> def test(a, b):
...     c = 3
...     return a + b + c
>>> test(1, 2)
6
>>> c
9
```

Interessant ist es zu beobachten, was geschieht, wenn innerhalb einer Funktion auf eine nicht in der Funktion deklarierte Variable zugegriffen und dann eine gleichnamige lokale Variable erzeugt wird:

```
>>> c = 9
>>> def test(a, b):
...     print(c)
...     c = 3
...     return a + b + c
>>> test(1, 2)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in test
UnboundLocalError: local variable 'c' referenced before assignment
```

Aus Sicht von *Python* liegt in diesem Fall bei der ersten Verwendung von `c` in der Funktionsdefinition eine Ambiguität vor, die aber dadurch aufgehoben werden kann, dass `c` explizit als globale Variable gekennzeichnet wird:

```
>>> def test(a, b):
...     global c
...     print(c)
...     c = 3
...     return a + b + c
>>> test(1, 2)
9 # Wert der globalen Variable c / Seiteneffekt
6 # Funktionswert
```

Alle Vorkommen von c in der Funktionsdefinition werden jetzt als Referenz auf die außerhalb der Funktion erzeugten Variable c interpretiert; d.h. durch die Zuweisung c = 3 innerhalb der Funktion wird die Bindung der globalen Variable c verändert:

```
>>> test(1, 2)
9
6
>>> c
3
```

6 Module

Neben den den Funktionen und Methoden, die nach dem Starten des Python-Interpreters zur Verfügung stehen, gibt es viele zusätzliche Funktionen, die in diversen Modulen organisiert sind. Module sind dazu gedacht, eine spezifische Funktionalität bereitzustellen, auf die bei Bedarf zugegriffen werden kann (Wiederverwendbarkeit von Code). Es ist für alle möglich, neue Module anzulegen. Formal betrachtet ist ein Modul nichts anderes als eine Python-Datei (Extension `.py`), die Definitionen und Anweisungen enthält. Mit `import MODUL` wird ein Modul eingelesen und die dort definierten Funktionen können nun genutzt werden. Um sie aufzurufen, muss der Modulname als Präfix verwendet werden; denn jedes Modul stellt einen eigenen Namensraum zur Verfügung, der hilft, potentielle Namenskonflikte zu vermeiden: `modul.funktion()`

Angenommen, das Modul `fibo.py` enthält die Definitionen der Funktionen `fibo1()` und `fibo2()`, dann kann nach dem Laden des Moduls durch `import fibo` so auf beide Funktionen zugegriffen werden:

```
>>> fibo.fibo1(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fibo2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Statt das gesamte Modul zu importieren ist es natürlich auch möglich, einzelne Funktionen/Methoden zu importieren: `from MODUL import FUNKTION 1, ..., FUNKTION n`

Auf die so importierten Funktionen kann jetzt ohne Angabe des Modulpräfixes zugegriffen werden: Also einfach `fibo1` (`fibo2`) anstelle von `fibo.fibo1` bzw, `fibo.fibo2` . Um auf alle Funktionen eines Moduls ohne Modulpräfix zugreifen zu können, verwendet man folgende Variante: `from MODUL import *`

Hier wird das Modul eingelesen und die dort definierten Funktionen können nun genutzt werden. Ihr Aufruf erfolgt nun einfach durch Verwendung ihres Namens (ohne Modulpräfix). Nicht importiert werden Namen, die mit einem Unterstrich '_' beginnen.

Angenommen, das Modul `fibo.py` enthält die Definitionen der Funktionen `fibo1()` und `fibo2()` , dann kann nach dem Laden des Moduls durch `from fibo import *` so auf beide Funktionen zugegriffen werden:

```
>>> fibo1(1000) # Ausgabe der Fibonacci-Zahlen unter 1000
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo2(100) # Ausgabe der Liste der Fibonacci-Zahlen unter 100
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Bei Bedarf ist es möglich, für Funktionen und Module Aliasse zu vereinbaren und anschließend zu verwenden: `from MODULE-NAME import FUNKTIONS_NAME as MEIN_NAME`

```
from fibo import fibo2 as fibolist
```

Module können grundsätzlich auch andere Module importieren; dabei werden die von einem Modul importierten Funktionen in den Namensraum des importierenden Moduls eingefügt.

Normalerweise sollte die `import`-Anweisungen immer **am Anfang der Datei** stehen.

7 Dokumentation

Auch wenn es viel Zeit kostet und aus diesem Grund gerne vernachlässigt wird:
Funktionen/Methoden und Programme sollten immer kommentiert werden.
Bei der Definition einer Funktion/Methode kann nach dem Funktionskopf eine Zeichenkette (`doc string`) angegeben werden, die die Syntax/Semantik der Funktion beschreibt. Diese Zeichenkette kann durch `FUNKTION.__doc__` extrahiert werden.

```
>>> def mittelwert(*werte):
... """Diese Funktion berechnet den arithmetischen Mittelwert einer
... beliebig langen Folge von Zahlen."""
... (...)
>>> mittelwert.__doc__
Diese Funktion berechnet den arithmetischen Mittelwert einer
beliebig langen Folge von Zahlen.
```

Für Parameter und Variablen sollten sinnvolle Bezeichner verwendet werden, die die intendierte Semantik der mit ihnen normalerweise assoziierten Werte widerspiegelt. Daneben kann es sinnvoll sein, der Definition einen Kommentar voranzustellen, der in schematischer Form Zahl und Art der Argumente, mögliche Seiteneffekt sowie den Typ des zurückgelieferten Wertes beschreibt.

8 Eingaben

Was die Interaktion zwischen Mensch und Maschine betrifft, haben wir uns bisher auf einfache Ausgaben (`print()`) beschränkt. Bevor wir uns in einer der nächsten Sitzungen ausführlich mit Ein- und Ausgaben beschäftigen werden, führen wir hier zwei kleine vordefinierte Funktionen ein, die es erlauben, Zeichenketten und Zahlen einzulesen.

Die Funktion `input()` nimmt optional eine Zeichenkette als Argument, die als Prompt ausgegeben wird. Dann wartet sie auf eine Eingabe und gibt als Wert eine Zeichenkette zurück, die diese Eingabe enthält. Die Eingabe muss mit `<RETURN>` `↵` abgeschlossen werden.

```
>>> eingabe = input('> ')
> Hallo! ↵
>>> print(eingabe)
Hallo!
>>> type(eingabe)
<class 'str'>
```

Leider ist der Wert immer eine Zeichenkette. Anders als in anderen Skriptsprachen werden rein numerische Eingaben nicht als Zahlen interpretiert:

```
>>> eingabe = input('> ')
> 42 ↵
>>> type(eingabe)
<class 'str'>
```

Ein Änderung des Typs eines Objektes (casting) kann auf verschiedene Weise erreicht werden: durch die Verwendung sogenannten Konstruktorfunktionen: `int()`, `float()`, `str()` oder durch die Funktion `eval()`, die eine Zeichenkette als Argument nimmt und ihn als eine Python-Anweisung interpretiert und ihren Wert zurückgibt:

```
>>> eingabe = int(input('> '))
> 42 ↵
>>> type(eingabe)
<class 'int'>
>>> eingabe = eval(input('> '))
> 42 ↵
>>> type(eingabe)
<class 'int'>
```

Auch bei dem naiven Versuch eine Liste von Werten einzulesen kommt es zu einer Überraschung:

```
>>> staedte = list(input("Liste: "))
Liste: ["Berlin", "Ulm", "Frankfurt", "Stuttgart", "Freiburg"] ↵
>>> staedte
['[', '"', 'B', 'e', 'r', 'l', 'i', 'n', '"', ',', ' ', '"', 'U', 'l', 'm', ... ,]']
```

Python wandelt die Zeichenkette in eine Liste, die jedes Zeichen aus dieser Kette als Listenelement enthält. In diesem Fall hilft `eval()`:

```
>>> staedte = list(eval(input("Liste: ")))
Liste: ["Berlin", "Ulm", "Frankfurt", "Stuttgart", "Freiburg"]
>>> staedte
['Berlin', 'Ulm', 'Frankfurt', ,'Stuttgart', 'Freiburg']
```