Informatik 1

Python III - Kontrollstrukturen

Jonas Miederer

DHBW Stuttgart - Campus Horb

8. Januar 2021

DHBW
Duale Hochschule
Baden-Württemberg

Outline:

DHBW
Duale Hochschule
Baden-Württemberg

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Built-in Funktionen

Primär besteht die Aufgabe eines Entwicklers darin, sich konkrete oder abstrakte Lösungen zu einem bestehenden Problem zu überlegen und diese mithilfe eines Algorithmus in einer Programmiersprache umzusetzen. Häufig sind das Problem sowie die Lösung dabei so problemspezifisch, dass die Funktionalität hierfür in der gewünschten Form noch nicht existiert und deswegen vom Programmierer von Grund auf entwickelt werden muss.

In der Regel sind Programme/Algorithmen aber so aufgebaut, dass komplexe Funktionalitäten aus der Komposition (Zusammensetzung) vieler einfacher Funktionen bestehen. Diese "einfachen" Funktionen sind so elementar und abstrakt, dass sie unabhängig vom konkreten Problem universell genutzt werden können. Daher ist es nicht sinnvoll, dass diese Elementarfunktionen, die sehr häufig benötigt werden, von jedem Entwickler von neuem geschrieben werden. Vielmehr enthalten die meisten Programmiersprachen bereits eine Menge verschiedener Grundfunktionen, die bereits fest in der Sprache integriert sind und daher ohne weitere Umstände von jedem Entwickler genutzt werden können.

Diese Funktionen, die bereits mit der Installation des Python-Interpreters mitgeliefert werden, werden **Built-in Funktionen** genannt.

Built-in Funktionen

Python bietet eine große Zahl von Built-in Funktionen, die dem Entwickler zur Verfügung stehen, was eine schnelle und einfache Entwicklung ermöglicht.

Die wichtigsten und hilfreichsten Built-in Funktionen werden im Folgenden vorgestellt. Einige dieser Built-in Funktionen sind bereits aus vorherigen Themengebieten bekannt.

Eine komplette Übersicht aller eingebauten Funktionen findet sich in der offiziellen Dokumentation: https://docs.python.org/3/library/functions.html

Konvertierungsfunktionen und Konstruktoren

Funktionen, um Datentypenkonvertierungen (Casting) durchzuführen:

- **Numerische Typkonvertierungen:** `int(x)`, `float(x)`, `complex(x)`, `bool(x)`, `bin(x)`, ...
- **String Typkonvertierungen:** `str(x)`, `unicode(x)`, ...
- **Sequence Konstruktoren und Typkonvertierungen:** `list(x)`, `tuple(x)`, `range(x)`, ...
- **Map Konstruktoren und Typkonvertierungen:** `dict(x)`
- **Set Konstruktoren und Typkonvertierungen:** `set(x)`, `frozenset`

DHBW Duale Hochschule Baden-Württemberg

Sequence Funktionen

Funktionen, die auf Sequence (und teilweise andere) Objekte angewendet werden können. **Das übergebene Sequence-Objekt `x` bleibt dabei jeweils unverändert**.

- `len(x)` : Anzahl der Elemente in der Sequence `x`
- `reversed(x)` : Erzeugt eine neue Sequence mit den gleichen Elementen in `x`, jedoch in umgekehrter Reihenfolge
- `sorted(x)` : Erzeugt eine neue Sequence mit den gleichen Elementen in `x`, jedoch in sortierter Reihenfolge (alphanumerische Sortierung)
- `sum(x)` : Erzeugt ein neues Skalar mit der Summe aller Elemente innerhalb der Sequence. Nur anwendbar auf numerische Sequence-Elemente.
- `zip(x)` : Erzeugt eine Sequence (zip-Objekt) von Tuples, in der jedes Tuple an Stelle i das jeweils i-te Element der übergebenen Sequences enthält

```
1 >>> result = zip([1,2,3], ['Alice', 'Bob', 'Carol'])
2 >>> print(list(result))
3 [(1, 'Alice'), (2, 'Bob'), (3, 'Carol')]
```

DHBW Duale Hochschule Baden-Württemberg

Input & Output (I)

Elementare Funktionen zur Interaktion zwischen Mensch und System bzw. zur Eingabe und Ausgabe von Daten:

- **Ausgabe:** Um Werte (Numerische Werte, Strings, ...) auf der Standardausgabe auszugeben, wird `print()` verwendet.
- **Eingabe:** Analog existiert eine Funktion zur Eingabe von Werten: `input()`. Mithilfe dieser Funktionen können Werte von der Standardeingabe eingelesen werden, um beispielsweise Nutzereingaben abzufragen. Der `input()`-Funktion kann auch ein Parameter übergeben werden. Dabei handelt es sich um einen String, der dem Nutzer vor der Eingabe angezeigt wird ("Prompt"). Die Eingabe kann anschließend in einer Variable gespeichert werden.

```
1  >>> x = input()
2  <<< Hello world
3  >>> print(x)
4  Hello world
5
6  >>> x = input("Please enter text: ")
7  <<< Bitte Text eingeben: Hello world
8  >>> print(x)
9  Hello world
```

Input & Output (II)

Achtung

Da der Python-Interpreter nicht weiß, welchen Datentyp die Eingabe besitzen soll, wird die Eingabe standardmäßig als Strings eingelesen und gespeichert.
Soll jedoch ein anderer Datentyp eingelesen werden, z.B. eine Ganzzahl, muss der Wert nach der Eingabe mit der entsprechenden Funktion konvertiert werden

```
>>> age = input("Please enter your age: ")
<<< Please enter your age: 20
>>> type(age)
<class 'str'>
>>> age = int(age)
>>> type(age)
<class 'int'>
```

Input & Output (III)

- **Eingabe:** Neben der Eingabe von Text können auch Dateien eingelesen werden. Hierzu steht die `open()` -Funktion zur Verfügung. Diese Funktion erwartet den Dateinamen (ggfs. inkl. Dateipfad, falls sich die Datei nicht im aktuellen Verzeichnis befindet.). Außerdem muss ggfs. der Modus angegeben werden: `'r'` , falls die Datei nur zum Lesen (Read) geöffnet wird, `'w'` zum Schreiben (Write)

```
1  >>> content = open("main.py", "r")
2  >>> type(content)
3  <class '_io.TextIOWrapper'>
4  >>> lines = content.readlines()
5  >>> type(lines)
6  <class 'list'>
7  >>> print(lines)
8  ['print("Hello world")\n']
```

Aufgaben I

1. Wozu gibt es die Built-in Funktionen?
2. Was ist nötig, um die Built-in Funktionen zu nutzen?
3. Es existieren zwei Tuples, die jeweils unterschiedliche Namen enthalten. Wie können daraus ganz einfach jeweils Paare von zwei Namen erzeugt werden?
4. Wo erscheint die Ausgabe von `print()` ?
5. Was muss bei der `input()` -Funktion beachtet werden? Worin liegt der Unterschied zur `open()` -Funktion?
6. Wie können mithilfe der `open()` -Funktion Dateiinhalte gelesen werden, die sich außerhalb des aktuellen Verzeichnisses befinden?

Hinweis

Hinweis: Weitere Übungsaufgaben zum Themenblock 'Built-in Funktionen' finden sie im entsprechenden Übungsblatt.

DHBW Duale Hochschule Baden-Württemberg

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Steueranweisungen

Mit den bis hierhin gelernten Funktionalitäten ist es bereits möglich, grundlegende und erweiterte Abläufe in Python umzusetzen. Von den eingangs vorgestellten Grundeigenschaften einer Programmiersprache sind nun bereits folgende Charakteristiken bekannt:

- Ein- und Ausgabe (`print()` und `input()` bzw. `open()`)
- Variablendeklaration
- Mathematische Grundoperationen
- Zeichenkettenverarbeitung

Als letzte noch fehlende Eigenschaft werden im folgenden die unterschiedlichen Steueranweisungen vorgestellt und erläutert.

DHBW
Duale Hochschule
Baden-Württemberg

Steueranweisungen

Mithilfe von Steueranweisungen kann der Ablauf eines Programms gesteuert werden. Somit ist es nun möglich, dass der Code nicht nur ausschließlich "von oben nach unten" durchläuft, sondern unterschiedliche Wege nehmen kann, an eine andere Stelle springt und dort fortgesetzt wird oder aber bestimmte Anweisungen mehrmals ausgeführt werden.

Dies ermöglicht eine effiziente Programmierung, mit der der statische Ablauf ohne spezifische Anpassungen in eine dynamische, flexiblere Arbeitsweise wechselt.

Es gibt unterschiedliche Arten von Steueranweisungen, die unterschiedliche Zwecke erfüllen. Die wichtigsten Steueranweisungen werden im Folgenden erläutert. Diese sind in fast jeder modernen Programmiersprache vorhanden, jedoch gibt es teilweise mehr oder weniger starke Abweichungen in der Syntax.

Conditionals

Die einfachste Art einer Steueranweisung sind **Conditionals** bzw. **Bedingte Anweisungen** oder auch **Verzweigungen** genannt.

Hiermit kann der Ablauf des Codes beeinflusst werden, indem anhand von klar definierten Bedingungen unterschiedliche Codeabschnitte ausgeführt werden können.

Conditionals sind eng mit den booleschen Werten verknüpft, da die Bedingungen jeweils als Wahrheitswert ausgewertet werden und daher entweder `True` oder `False` ergeben.

Bedingte Anweisungen sind insbesondere dann unabdingbar, wenn mit dynamischen Informationen, wie etwa Dateneinspeisungen von anderen System oder Nutzereingaben gearbeitet wird, da hier die exakten Daten zur Zeit der Programmierung noch nicht bekannt sind und daher der Programmablauf dynamisch angepasst werden muss.

DHBW Duale Hochschule Baden-Württemberg

if (I)

Die Conditionals folgen immer einem "Wenn Dann"-Schema, d.h. **wenn** die Bedingung wahr ist, **dann** wird der entsprechende Codeabschnitt ausgeführt. Andernfalls wird der Codeabschnitt übersprungen.

Dementsprechend existiert in Python das Schlüsselwort `if`, mit dem eine Bedingung eingeleitet und entsprechend ausgewertet wird.

Der Programmablauf folgt dabei der exemplarischen Darstellung 1, die auf der nächsten Seite abgebildet ist.

if (II)

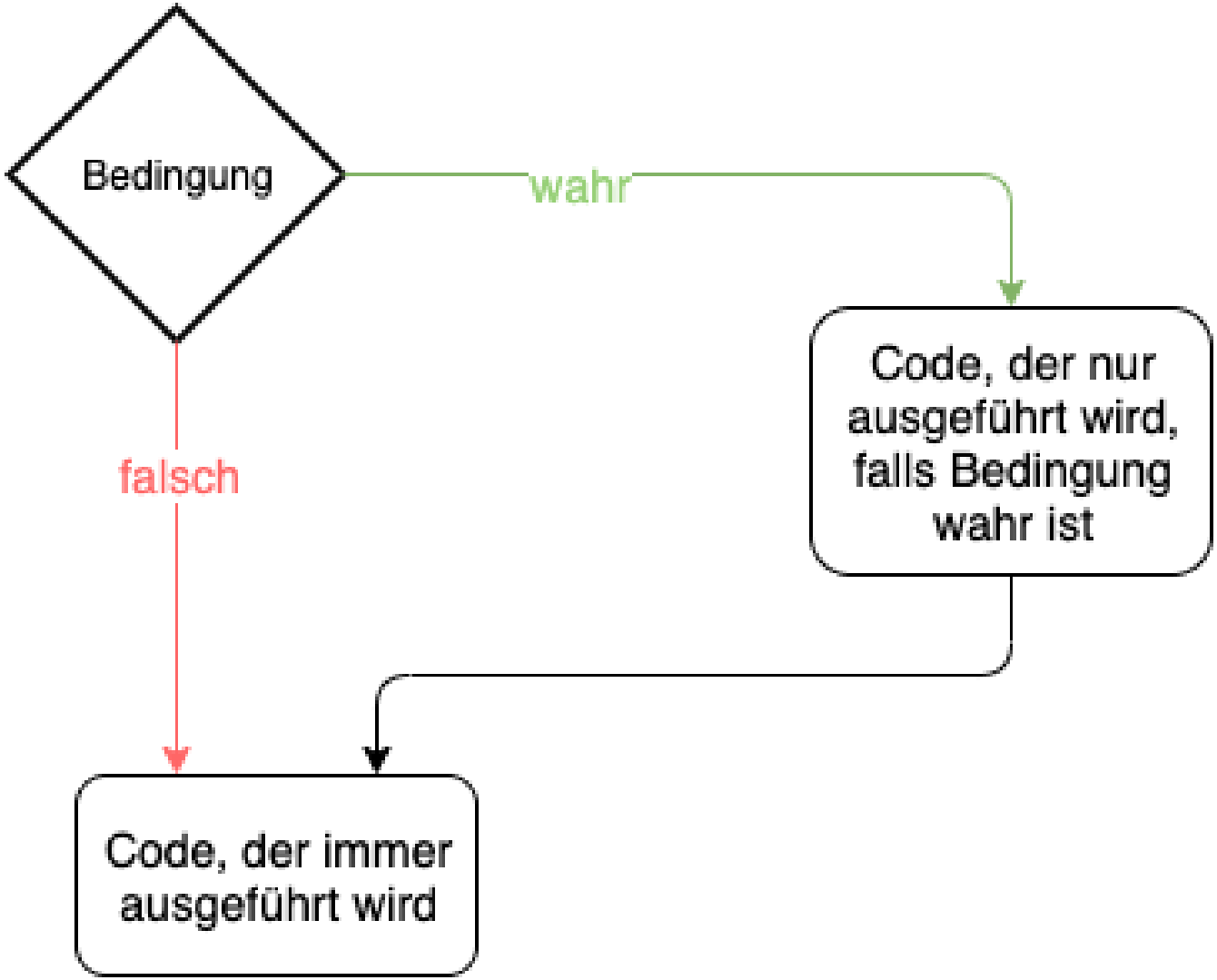

Abbildung 1: Der Programmablauf mit einer Verzweigung

if (III)

Die Syntax sieht dabei folgendermaßen aus:

```
1  if <Bedingung>:
2      # Anweisungen, die ausgeführt werden, wenn Bedingung wahr ist
3
4  # Anweisungen, die immer ausgeführt werden
```

Hierbei sind mehrere Dinge zu beachten:

- Die Bedingung muss immer entweder `True` oder `False` sein. Auch Verknüpfungen von logischen Aussagen mit z.B. `and` sind zulässig
- Nach der Bedingung folgt immer ein Doppelpunkt. Hiermit wird verdeutlicht, dass im Folgenden die Anweisungen stehen, die ausgeführt werden, falls die Bedingung wahr ist
- Diese Bedingungen **müssen um eine Ebene** eingerückt sein, um den Anfang und das Ende des Blocks zu kennzeichnen
- Anweisungen außerhalb der if-Clause werden wieder ausgerückt

DHBW Duale Hochschule Baden-Württemberg

if (IV)

Beispiel

```
1  age = int(input("Bitte Alter eingeben: "))
2  if age >= 18:
3      print("Der Nutzer ist volljährig")
4
5  print("Der Nutzer ist " + str(age) + " Jahre alt")
```

if (V)

Achtung

Der interaktive Interpreter wertet im Normalfall jede Eingabe des Nutzer direkt aus. Nachdem der Nutzer aber eine if-Clause eingegeben hat, kann diese der Interpreter noch nicht ausführen, da er noch die Anweisungen erwartet, die ausgeführt werden sollen, falls die Bedingung wahr ist. Aus diesem Grund zeigt der Interpreter nach Eingabe der if-Clause zunächst `...` an, wodurch der Nutzer nun seine Anweisungen eingeben kann. Soll die if-Clause abgeschlossen werden, muss der Nutzer dies erneut mit der Enter-Taste bestätigen.

if-else (I)

In vielen Fällen soll nicht nur überprüft werden, ob eine Bedingung wahr ist, sondern auch Anweisungen ausgeführt werden, wenn dies nicht der Fall ist. Eine Möglichkeit besteht darin, die verneinte Aussage erneut zu überprüfen:

```
1  age = int(input("Bitte Alter eingeben: "))
2  if age >= 18:
3      print("Der Nutzer ist volljährig")
4
5  if age < 18:
6      print("Der Nutzer ist nicht volljährig")
```

Da dieses Vorgehen jedoch redundant ist gibt es hierfür eine häufig verwendete Kurzform: Das **if-else** Konstrukt

if-else (II)

Mithilfe von **if-else** kann zunächst überprüft werden, ob eine Aussage wahr ist und dementsprechend Anweisungen ausgeführt werden. Ist die Aussage jedoch nicht wahr, werden die Anweisungen ausgeführt, die in der `else` -Clause enthalten sind

Der Programmablauf folgt dabei der exemplarischen Darstellung 2, die auf der nächsten Seite abgebildet ist.

if-else (III)

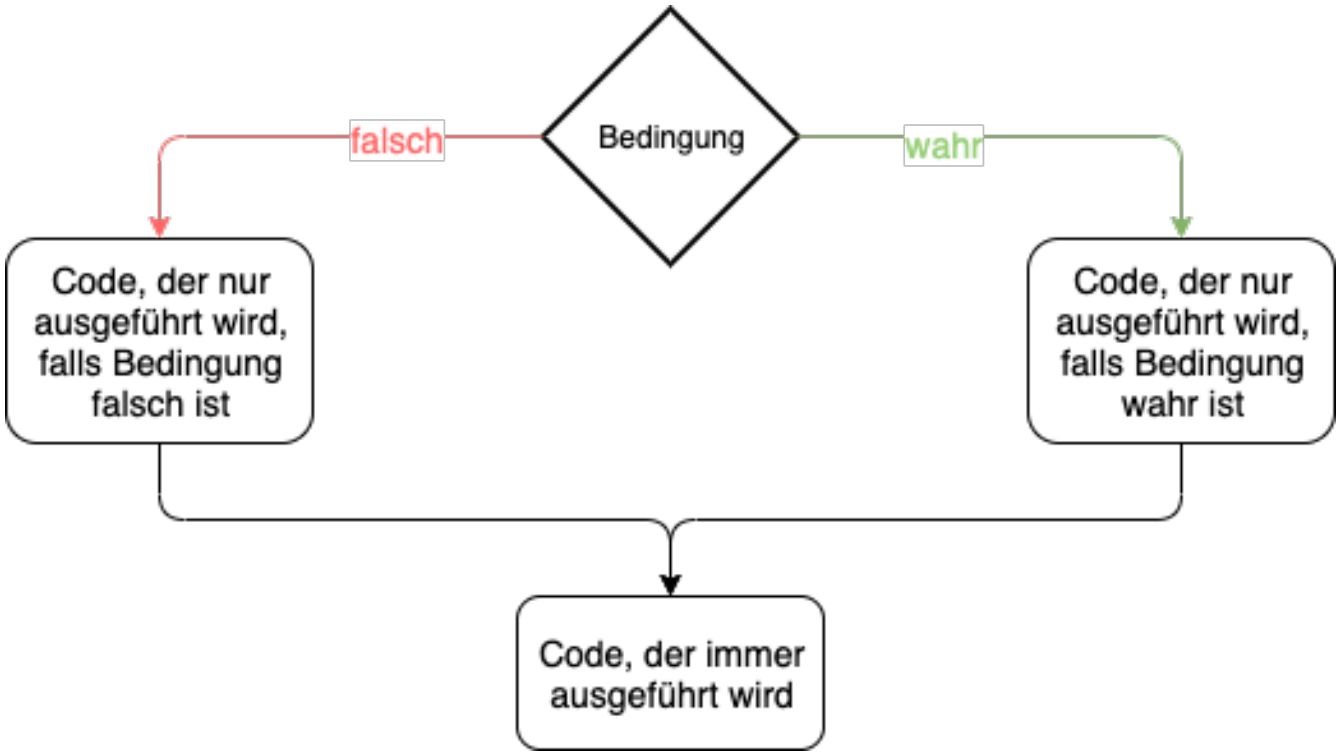

Abbildung 2: Der Programmablauf mit einer if-else Verzweigung

if-else (IV)

Die Syntax sieht dabei folgendermaßen aus:

```
1  if <Bedingung>:
2      # Anweisungen, die ausgeführt werden, wenn Bedingung wahr ist
3  else:
4      # Anweisungen, die ausgeführt werden, wenn Bedingung falsch ist
5
6  # Anweisungen, die immer ausgeführt werden
```

Hierbei sind mehrere Dinge zu beachten:

- Bei der else-Clause wird nie eine Bedingung überprüft, da diese automatisch ausgeführt wird, falls die Bedingung der if-Clause zu `False` auswertet.
- Es wird **entweder** die if-Clause oder **oder** die else-Clause ausgeführt. Es ist nicht möglich, dass keine der beiden oder beide gleichzeitig ausgeführt werden.
- Nach den Anweisungen der if-Clause muss eine Ausrückung erfolgen, da hier ein neuer Block beginnt. Die Anweisungen der else-Clause werden dann wieder eingerückt.
- Eine else-Clause ist immer optional

DHBW Duale Hochschule Baden-Württemberg

if-else (V)

Beispiel

```
1  age = int(input("Bitte Alter eingeben: "))
2  if age >= 18:
3      print("Der Nutzer ist volljährig")
4  else:
5      print("Der Nutzer ist nicht volljährig")
6
7  print("Der Nutzer ist " + str(age) + " Jahre alt")
```

if-elif-else (I)

Um komplexere Abfragen noch einfacher umsetzen zu können, gibt es eine Erweiterung des `if` bzw. `if-else` Konstrukts:

Mithilfe von else-if (Keyword: `elif`) können beliebig viele Aussagen auf ihren Wahrheitswert überprüft werden. Sobald eine dieser Aussagen wahr ist, werden die entsprechenden Anweisungen ausgeführt. Die nachfolgenden `elif`-Clauses werden dann nicht mehr überprüft, auch die `else`-Anweisungen werden dann nicht mehr ausgeführt.

Der Programmablauf folgt dabei der exemplarischen Darstellung 3, die auf der nächsten Seite abgebildet ist.

if-elif-else (II)

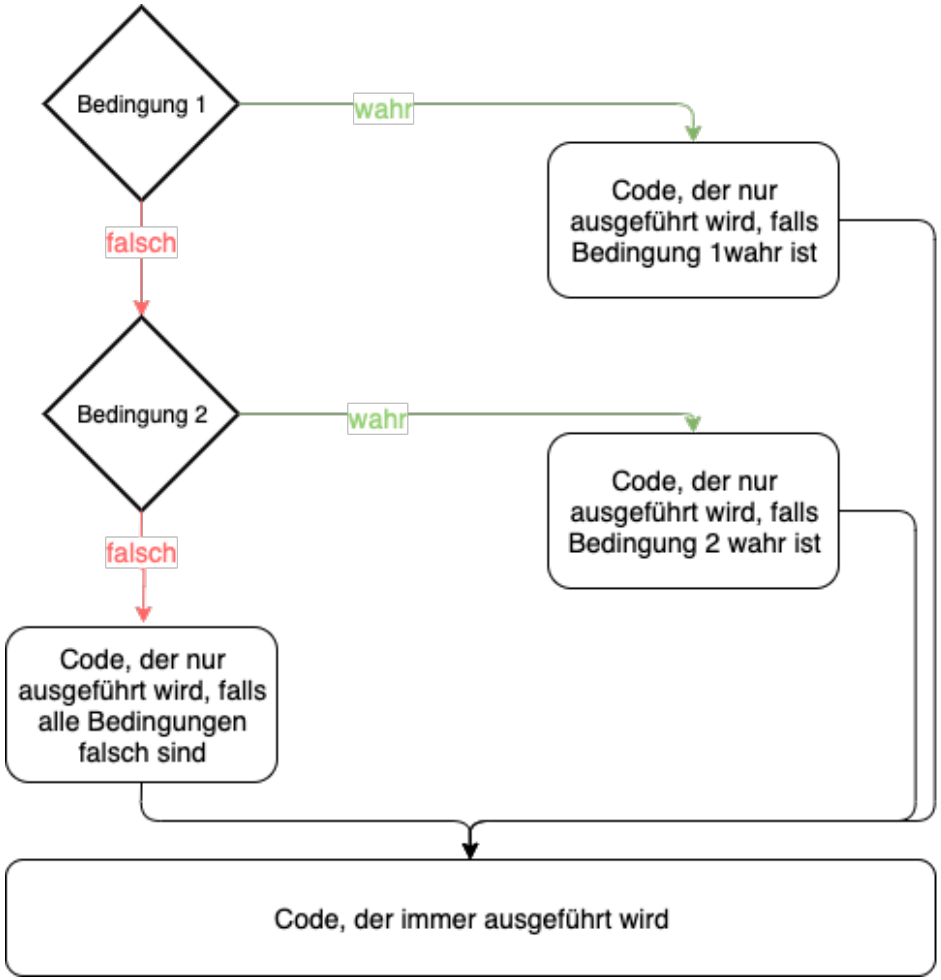

Abbildung 3: Der Programmablauf mit einer if-elif-else Verzweigung

if-elif-else (III)

Die Syntax sieht dabei folgendermaßen aus:

```
1  if <Bedingung 1>:
2      # Anweisungen, die ausgeführt werden, wenn Bedingung 1 wahr ist
3  elif <Bedingung 2>:
4      # Anweisungen, die ausgeführt werden, wenn Bedingung 2 wahr und Bedingung 1 falsch
       ↪ ist
5  elif <Bedingung 3>:
6      # Anweisungen, die ausgeführt werden, wenn Bedingung 3 wahr und Bedingung 1 und
       ↪ Bedingung 2 falsch sind
7  else:
8      # Anweisungen, die ausgeführt werden, wenn alle Bedingungen falsch sind
9
10 # Anweisungen, die immer ausgeführt werden
```

Hierbei sind mehrere Dinge zu beachten:

- Es werden die Bedingungen der Reihe nach geprüft. Die Anweisungen der ersten Bedingung, die `True` ergibt, werden ausgeführt, die restlichen `elif` bzw. `else`-Statements ignoriert.
- Wertet keine der `if` oder `elif`-Clauses zu `True` aus, wird die `else`-Clause ausgeführt (falls vorhanden)
- Auch hier ist die `else`-Clause optional

if-elif-else (IV)

Beispiel

```
1  age = int(input("Bitte Alter eingeben: "))
2  if age > 18:
3      print("Der Nutzer ist volljährig")
4  elif age == 18:
5      print("Der Nutzer ist dieses Jahr volljährig geworden.")
6  else:
7      print("Der Nutzer ist nicht volljährig")
8
9  print("Der Nutzer ist " + str(age) + " Jahre alt")
```

Verschachtelte Conditionals (I)

Bedingte Anweisungsblöcke können auch verschachtelt werden, d.h. in einer `if`, `elif` oder `else`-Clause können wiederum weitere bedingte Anweisungen enthalten sein.

Eine (praktische) Maximalzahl an erlaubten Verschachtelungen gibt es nicht.

Verschachtelte Conditionals (II)

Achtung

Für jeden Anweisungsblock muss der Code um eine Ebene eingerückt werden. Enthält also beispielsweise eine `if` -Clause eine weitere `if` -Clause, muss der Code um 2 Ebenen eingerückt werden.

Verschachtelte Conditionals (III)

Beispiel

```
1  username = int(input("Bitte Benutzername eingeben: "))
2  password = input("Bitte Passwort eingeben: ")
3
4  if username in  ["Alice", "Bob", "Carol"]:
5      if len(password) < 6:
6          print("Das Passwort ist unsicher")
7      else:
8          print("Der Nutzer " + username + " existiert und das Passwort ist sicher")
9  else:
10     print("Der Nutzername ist nicht bekannt")
```

Aufgaben I

1. Wofür werden Conditionals benötigt?
2. Kann jede Funktionalität der Conditionals auch ohne Conditionals genutzt werden?
3. Warum gibt es Einrückungen?
4. Können bei Conditionals auch andere Datenwerte als boolesche Werte auf Wahrheit getestet werden?
5. Warum werden bei else-Clauses keine Bedingungen abgefragt?
6. Wie viele Anweisungen darf eine if-Clause besitzen?
7. Kann eine `elif` -Clause auch anders ausgedrückt werden?
8. Werden stets alle Bedingungen eines if-elif-else-Konstrukts überprüft?
9. Was muss bei der Verschachtelung von Conditionals beachtet werden?

Hinweis

Hinweis: Weitere Übungsaufgaben zum Themenblock 'Conditionals' finden sie im entsprechenden Übungsblatt.

DHBW Duale Hochschule Baden-Württemberg

Schleifen (I)

Schleifen stellen in der Programmierung ein wichtiges Werkzeug dar, das zu einer effizienten, schnellen und einfachen Entwicklung beiträgt.

Der Sinn einer Schleife besteht darin, eine Anweisung mehrmals auszuführen, ohne jede einzelne dieser Anweisungen "hartgecoded" (also explizit aufgeführt) in den Programmcode zu schreiben.

Schleifen werden häufig auch *Loops* genannt.

Schleifen (II)

Im Folgenden ist ein Beispiel dargestellt, wie alle Elemente einer Liste (ohne Schleife) der Reihe nach ausgegeben werden:

Beispiel

```
1  names = ['Alice', 'Bob', 'Carol', 'Dave']
2  index = 0
3
4  print(names[index])
5  index = index+1
6  print(names[index])
7  index = index+1
8  print(names[index])
9  index = index+1
10 print(names[index])
```

Hier ist deutlich zu erkennen, dass diese Art der Ausgabe aller Listenelemente sehr ineffizient ist. Insbesondere bei längeren Listen ist eine solche Vorgehensweise unpraktikabel.

DHBW Duale Hochschule Baden-Württemberg

Schleifen (III)

Abhilfe hierfür schaffen Schleifen, da mit diesen wiederkehrende Anweisungen nur einmal angegeben werden müssen und diese dann beliebig oft ausgeführt werden kann.

Generell gibt es in Python zwei Arten von Schleifen: `while`-Schleifen und `for`-Schleifen.

Die Funktionsweise der beiden Schleifentypen ist sehr ähnlich, mit beiden Arten können jeweils die gleichen Funktionalitäten umgesetzt werden. Jedoch gibt es feine Unterschiede, durch die sich `for` und `while`-Schleifen unterscheiden und sich je nach Anwendungsfall der eine oder der andere Typ eher anbietet.

Schleifen (IV)

Alle Schleifen folgen dem gleichen Ablauf:

1. Prüfe, ob die Abbruchbedingung der Schleife erfüllt ist. Falls nicht, gehe zu Schritt 2
2. Führe die Anweisungen aus, die im Schleifenrumpf enthalten sind. Anschließend gehe zu Schritt 1.

DHBW Duale Hochschule Baden-Württemberg

while (I)

Mithilfe von `while`-Schleifen können Anweisungen solange ausgeführt werden, solange die Anweisung im Schleifenkopf wahr ist.

Das bedeutet, dass der Entwickler sich zunächst überlegen muss, was das **Abbruchkriterium** ist.
Die Syntax des `while`-Loops sieht folgendermaßen aus:

```
1  while <Bedingung>:
2      # Anweisungen, die ausgeführt werden, solange die Bedingung wahr ist
3
4  # Anweisungen, die ausgeführt werden, nachdem die Schleife beendet wurde /
   ↪  durchgelaufen ist
```

while (II)

Die Bedingung, die im Schleifenkopf angegeben wird, muss stets ein boolescher Ausdruck sein, die entweder `True` oder `False` ergibt. Sobald die Anweisung zu `False` ausgewertet wird, ist die Schleife beendet und der Codeablauf wird fortgesetzt.

`while`-Schleifen werden i.d.R. dann verwendet, wenn eine Anweisung bestimmt oft (x Mal) ausgeführt werden soll.

while (III)

Achtung

`while`-Schleifen sind **kopfgesteuerte** Schleifen. Das bedeutet, dass die Abbruchbedingung **vor** jedem Schleifendurchgang überprüft wird. Wenn die Bedingung `False` ist, wird der nächste Schleifendurchgang nicht mehr ausgeführt. Es gibt auch **fußgesteuerte** Schleifen, bei denen **nach** jedem Durchgang die Bedingung geprüft wird und abhängig davon der nächste Schleifendurchgang gestartet wird.

Das hat zur Folge, dass fußgesteuerte Schleifen mindestens einmal durchlaufen werden, bei kopfgesteuerten Schleifen ist es auch möglich, dass kein Schleifendurchlauf stattfindet.

while (IV)

Beispiel

Im folgenden Codeausschnitt sollen die Zahlen von 0 bis 5 in absteigender Reihenfolge ausgegeben werden, ohne die expliziten Werte auszugeben:

```
>>> i = 5
>>> while i >= 0:
>>>     print(i)
>>>     i = i-1

5
4
3
2
1
0
```

while (V)

Achtung

Die Anweisung `i = i-1` im vorherigen Beispiel ist sehr wichtig, da hierdurch die Variable `i` um den Wert 1 verringert wird. Wird diese Anweisung vergessen, wird bei jedem Schleifendurchgang überprüft, ob `i>0` wahr ist. Da sich `i` nicht ändert, wird stets überprüft ob `5 > 0` wahr ist, was immer der Fall ist. Dies hat eine **Endlosschleife** zur Folge, da die Abbruchbedingung nie erreicht wird.

for (I)

Neben den `while` Schleifen gibt es noch eine weitere Schleifenart: Die `for` Schleife.

Anders als `while` Schleifen sind `for` Schleifen nicht zähler- sondern elementbasiert. Der Vorteil liegt darin, dass so ein Durchlaufen (Iterieren) aller Elemente einer Liste oder listenähnlichen Struktur sehr einfach ist.

Die Gefahr, dass der Entwickler vergisst, eine Zählvariable zu in-/dekrementieren und so eine Endlosschleife zu erzeugen, sinkt.

for (II)

Die Syntax einer `for`-Schleife ist folgendermaßen:

```
1  for <Elementvariable> in <Sequence>:
2      # Anweisungen, die für jedes Element der Sequence ausgeführt werden
3
4  # Anweisungen, die nach Durchlaufen der Schleife ausgeführt werden
```

Bei `for` Schleifen wird über jedes Element der Sequenz iteriert, welches automatisch einer Variablen (oben `<Elementvariable>`) zugewiesen wird.

Im Gegensatz zur `while` Schleife muss der Entwickler hier also keine explizite Abbruchbedingung angeben, sondern der Schleifendurchlauf endet, wenn alle Elemente der Sequenz ein Mal durchlaufen wurden.

for (III)

Beispiel

Im folgenden Codeausschnitt sollen die Namen, die in der Liste enthalten sind, ausgegeben werden:

```
1   names = ['Alice', 'Bob', 'Carol', 'Eve', 'Dave']
2
3   for name in names:
4       print(name)
5
6   Alice
7   Bob
8   Carol
9   Eve
10  Dave
```

Der Name der Variablen `name` als Variable für das jeweilige Listenelement kann frei gewählt werden [1]

[1] solange er den Regeln der Variablenbenennung entspricht

range (I)

Eine der am häufigsten im Zusammenhang mit der `for` Schleife genutzten Built-In Funktionen ist `range()`.

Mithilfe von `range()` können einfach Sequenzen von Zahlen erzeugt werden.

Wird der `range()`-Funktion nur ein Parameter übergeben, wird eine Sequenz von Zahlen von 0 bis zur entsprechenden Zahl erzeugt:

```
>>> intervall = range(5)
>>> type(intervall)
<class 'range'>

>>> intervall_list = list(intervall)
>>> type(intervall_list)
<class 'list'>
>>> print(intervall_list)
[0, 1, 2, 3, 4]
```

DHBW Duale Hochschule Baden-Württemberg

range (II)

Werden der `range()` -Funktion 2 Parameter übergeben, wird eine Sequenz von Zahlen vom Anfangswert (Parameter 1) bis zum Endwert (Parameter 2) erzeugt:

```
1   intervall = range(5,12)
2   for number in intervall:
3       print(number)
4
5   5
6   6
7   7
8   8
9   9
10  10
11  11
```

range (III)

Werden der `range()` -Funktion 3 Parameter übergeben, wird eine Sequenz von Zahlen vom Anfangswert (Parameter 1) bis zum Endwert (Parameter 2) mit einer Schrittweite (Parameter 3) erzeugt:

```
1  intervall = range(5,12,2)
2  for number in intervall:
3      print(number)
4
5  5
6  7
7  9
8  11
```

Wird keine Schrittweite angegeben, wird automatisch der Standartwert 1 verwendet.

range (IV)

Achtung

Bei der `range()` -Funktion ist der Anfangswert stets inklusiv, der Endwert immer exklusiv. `range(2,7)` beinhaltet also alle Zahlen zwischen 2 und 6.

range (V)

Die `range()` -Funktion wird häufig in Kombination mit der `for` Schleife genutzt, da so die Funktionalität einer zählerbasierten `while` Schleife nachgebildet werden kann, ohne sich um die Initialisierung und Inkrementierung des Zählers kümmern zu müssen.

Beispiel

Mit der folgenden `while` Schleife werden die Zahlen von 1 - 20 ausgegeben:

```
1  i = 1
2  while i<=20:
3      print(i)
4      i = i+1
```

Die selbe Funktionalität kann mit folgender `for` Schleife in Kombination einfacher und übersichtlicher umgesetzt werden:

```
1  for i in range(1, 21):
2      print(i)
```

continue - Iterationsabbruch (I)

Es gibt zwei Hilfsanweisungen, die bei der Entwicklung von Schleifen hilfreich sein können. Hierzu gehört `continue`.

Die `continue` Anweisung wird genutzt, falls der aktuelle Iterationsdurchgang abgebrochen werden und mit dem nächsten Iterationsdurchgang weitergemacht werden soll (solange die Abbruchbedingung nicht erfüllt ist).

Oft wird die `continue`-Anweisung zu den schlechten Codepraktiken gezählt, da es deutlich schwieriger werden kann, als Mensch den Programmablauf nachzuvollziehen. Aus diesem Grund sollte auf `continue` verzichtet werden falls möglich

DHBW Duale Hochschule Baden-Württemberg

continue - Iterationsabbruch (II)

Beispiel

Im folgenden Beispiel wird für die Zahlen von 2 - 10 überprüft, ob sie gerade oder ungerade sind:

```
1    for num in range(2, 10):
2        if num % 2 == 0:
3            print("Gerade Zahl gefunden")
4            continue
5        print("Ungerade Zahl gefunden")
```

Dieser Codeausschnitt soll lediglich die Funktionsweise von `continue` verdeutlichen. Der abgebildete Code sollte so nicht verwendet werden, da er unnötig umständlich ist. Eine einfachere und bessere Variante stellt hier eine Implementierung mit if-else dar.

break - Schleifenabbruch (I)

Neben `continue` gibt es noch eine weitere Hilfsanweisung: `break`.

Die `break` Anweisung wird genutzt, um das Durchlaufen der Schleife abzubrechen, auch wenn die Abbruchbedingung noch nicht erfüllt ist. Im Gegensatz zu `continue` wird also nicht nur die aktuelle Iteration, sondern die komplette Schleife abgebrochen.

Auch `break` sollte nur genutzt werden, wenn die Anweisung wirklich benötigt wird, andernfalls sollten eindeutigere Codeumsetzungen bevorzugt werden.

DHBW Duale Hochschule Baden-Württemberg

break - Schleifenabbruch (II)

Beispiel

Im folgenden Beispiel wird für die Zahlen ab 100 überprüft, ob sie durch 7 teilbar sind. Falls das der Fall ist, wird die Schleife abgebrochen:

```
1  index = 100
2  divider = 7
3  while True:
4      if index % divider == 0:
5          print(str(index) + " ist teilbar durch " + str(divider))
6          break
7      else:
8          print(str(index) + " ist nicht teilbar durch " + str(divider))
9          index = index +1
```

Aufgaben I

1. Wofür sind Schleifen hilfreich?
2. Welche Arten von Schleifen gibt es? Was sind die Unterschiede?
3. Welche kopfgesteuerten bzw. fußgesteuerten Schleifen gibt es in Python?
4. Wann werden primär `while` und wann `for` Schleifen genutzt?
5. Was muss bei `while` Schleifen unbedingt beachtet werden? Warum ist das bei `for` Schleifen nicht relevant?
6. Welche Funktionalität bietet die `range()` -Funktion?
7. Warum wird beim Erzeugen einer Range eine Variable des Typs `range` und nicht `list` erzeugt?
8. Was muss beim Erzeugen einer `range` beim Start- und Endwert beachtet werden?
9. Was ist das Ergebnis der Anweisung `list(range(20,0,-2))` ?
10. Wozu wird `continue` genutzt und warum sollte es vermieden werden?

DHBW Duale Hochschule Baden-Württemberg

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Funktionen

Funktionen gehören zu den Grundkonzepten einer Programmiersprache und sind in so gut wie allen Programmiersprachen vorhanden und verwendbar. ˜

Sie stellen ein wichtiges Werkzeug dar, um den Code zu strukturieren, Funktionalitäten zusammenzufassen und Codewiederholungen zu vermeiden.
Da die Lesbarkeit und die übersichtliche Strukturierung von Code in Python eine wichtige Rolle einnimmt (siehe *The Zen of Python* bzw. *Computer Programming for Everybody* (Foliensatz *Python 1*)), nehmen die Funktionen hier eine wichtige Rolle ein.

Darüber hinaus trägt eine klare Strukturierung und Aufteilung des Codes zu einer guten Codequalität bei, die es ermöglicht, Fehler schneller zu finden und den Codeablauf leichter zu beschreiben.

DHBW
Duale Hochschule
Baden-Württemberg

Bereits bekannte Funktionen

In Abs. 1 wurden bereits einige Funktionen vorgestellt, die in Python standardmäßig enthalten sind und ohne weiteres aufrufbar sind. Python bietet im Gegensatz zu einigen anderen Programmiersprachen einen großen Umfang an Built-in Funktionen, die in vielen Fällen sehr praktisch sein können.

Beispielsweise gehören hierzu die Ein- / Ausgabefunktionen `input()` und `print()`. Da diese Funktionalitäten sehr häufig benötigt werden, wäre es sinnlos, wenn jeder Entwickler sie jedes Mal aufs Neue implementieren müsste.

Eigene Funktionen

Da Built-in Funktionen nur die häufigsten und gebräuchlichsten Anwendungsfälle abdecken, existieren für spezifische Problemstellungen keine standardmäßigen Funktionen im Python-Interpreter. Diese müssen dann vom Programmierer selbst entwickelt werden.

Häufig werden aber auch innerhalb eines Problems ähnliche Funktionalitäten mehrmals benötigt, sodass Funktionen, die vom Entwickler einmal geschrieben wurde, an unterschiedlichen Stellen im Programmcode wiederverwendet werden können.

Deklaration vs. Aufruf

Bei der Entwicklung bzw. Verwendung von Funktionen müssen zwei Fälle getrennt bzw. unterschieden werden, die bei der Implementierung von Funktionen von Bedeutung sind:

Deklaration/Definition von Funktionen
Bevor Funktionen verwendet werden können, müssen sie zunächst definiert bzw. deklariert werden. Hierbei wird beschrieben, wie die Funktion heißt, welche Parameter sie erwartet, welche Anweisung sie ausführen soll und welche Werte sie zurückgibt.
Allein durch die Deklaration wird sie jedoch noch nicht ausgeführt, hier wird sie lediglich **beschrieben**.

Aufruf von Funktionen
Erst nachdem Funktionen definiert wurden, können sie auch **ausgeführt** werden. Ein Funktionsaufruf kann sich an komplett anderer Stelle befinden als die entspr. Definition. Wichtig ist aber, dass sich die Funktion im gleichen Scope befindet. Durch den Funktionsaufruf werden der Funktion die benötigten Parameter übergeben, anschließend werden die Anweisungen ausgeführt und letztendlich ein oder mehrere Werte zurückgegeben.

DHBW
Duale Hochschule
Baden-Württemberg

Funktionsdefinition (I)

Eine Funktionsdefinition besteht aus zwei Teilen:

Eine Funktion beginnt immer mit dem **Funktionskopf**. Um zu kennzeichnen, dass eine Funktionsdefinition folgt, beginnt dieser immer mit dem `def` Schlüsselwort. Anschließend folgt der Name der Funktion. Bei der Benamung gelten die gleichen Regeln wie bei Variablen (siehe Foliensatz Python 1). In runden Klammern sind dahinter die erwarteten Parameter aufgeführt. Der Funktionskopf endet mit einem Doppelpunkt.

Die eigentliche Funktionalität ist im **Funktionsrumpf** enthalten. Hier befinden sich die Anweisungen , die ausgeführt werden, wenn die Funktion aufgerufen wird. Um zu kennzeichnen, dass eine Funktion einen eigenen Scope bildet und auch kenntlich zu machen, wo die Funktion endet, ist der Funktionsrumpf jeweils um eine Ebene eingerückt.

Funktionsdefinition (II)

Beispiel

Hier wird eine Funktion mit dem Namen `greet_user` definiert. In dieser Funktion wird der Anwender zuerst nach seinem Namen gefragt, dieser wird in der `name`-Variable gespeichert. Anschließend wird der Nutzer mit seinem Namen gegrüßt.

```
1  def greet_user():                          # Funktionskopf
2      name = input("What is your name? ")     # Funktionsrumpf
3      print("Hello " + name)                  # Funktionsrumpf
```

Achtung: Allein mit diesem Codeabschnitt erfolgt noch keine Ausgabe, da die Funktion bisher lediglich definiert, nicht aber aufgerufen wurde.

DHBW Duale Hochschule Baden-Württemberg

Funktionsaufruf (I)

Nachdem eine Funktion definiert wurde, kann sie an einer beliebigen Stelle im Code aufgerufen werden. Der Aufruf erfolgt dabei mit dem Namen der jeweiligen Funktion sowie den zusätzlich zu übergebenden Parametern.

Sobald die Funktion aufgerufen wird, wird bei der Ausführung des Codes an die Stelle der Funktionsdefinition gesprungen, die darin enthaltenen Anweisungen sequentiell ausgeführt und anschließend wird wieder zur Stelle des Funktionsaufrufs zurückgesprungen und der weitere Ablauf des Codes dort fortgesetzt (siehe Abb. 4).

Funktionsaufruf (II)

```
1  print("This program will greet the user")

   def greet_user():
3      name = input("What is your name? ")
4      print("Hello " + name)

2  greet_user()

5  print("This program will exit now")
```

Funktionsdefinition

Funktionsaufruf

Abbildung 4: Der Programmablauf beim Aufruf einer Funktion. Die Ausführungsreihenfolge der einzelnen Anweisungen ist links in rot dargestellt.

Funktionsaufruf (III)

Achtung

Beim Aufruf einer Funktion muss immer sichergestellt sein, dass die Funktion zum Zeitpunkt des Aufrufs bereits bekannt ist. Folgender Codeabschnitt würde also einen Fehler erzeugen, da die Funktion aufgerufen wird, bevor sie definiert wurde.

```
1  greet_user()
2
3  def greet_user():
4      name = input("What is your name? ")
5      print("Hello " + name)
```

Parameter & Argumente (I)

In vielen Fällen werden von Funktionen zusätzliche Informationen erwartet, die beispielsweise lediglich in Variablen außerhalb der Funktion existieren und daher im Rumpf der Funktion nicht zugänglich sind.

Hierfür können den Funktionen zusätzliche Daten übergeben werden, welche dann innerhalb der Funktionsausführung genutzt werden können. Diese Daten müssen sowohl bei der Definition als auch dem Aufruf von Funktionen berücksichtigt werden.

Sowohl Parameter als auch Argumente werden jeweils durch Komma getrennt.

Funktionsdefinition - Parameter (I)

Bei der Funktionsdefinition muss angegeben werden, welche bzw. wie viele Daten erwartet werden. Diese Werte werden dann einem Variablennamen zugewiesen, der lediglich innerhalb der Funktion (Scope) existiert. Sobald die Funktion abgearbeitet wurde und im Ablauf aus der Funktion gesprungen wird, sind dem Interpreter die zugewiesenen Werte nicht mehr bekannt.

Die Werte, die von einer Funktion erwartet werden, heißen **Parameter**.

Funktionsdefinition - Parameter (II)

Beispiel

In diesem Codeabschnitt werden der Funktion `add` zwei Werte übergeben, nämlich die zwei Summanden. Diese Werte können innerhalb der Funktion mithilfe der Variablen `summand1` und `summand2` verwendet werden. Diese beiden Werte sind die Parameter, die beim Aufruf der Funktion übergeben werden müssen.
Die Bezeichnung der Parameter ist dem Entwickler selbst überlassen, die Parameternamen sollten jedoch ebenso wie normale Variablen möglichst aussagekräftig sein.

```
1    def add(summand1, summand2):
2        sum = summand1 + summand2
3        print("The result of " + summand1 + " + " + summand2 + " is " + sum)
```

Funktionsaufruf - Argumente (I)

Falls eine Funktion per ihrer Definition Werte erwartet, wie im vorherigen Beispiel die `add`-Funktion, so müssen die konkreten Werte beim Aufruf der Funktion zwangsläufig mit übergeben werden.

Wird also die `add`-Funktion aufgerufen, müssen zwei Werte (die beiden Summanden) übergeben werden. Die konkreten Werte, die beim Funktionsaufruf an die Funktionsdefinition übergeben werden, heißen **Argumente**.

Funktionsaufruf - Argumente (II)

Beispiel

In diesem Codeabschnitt wird die Funktion `add` mit den beiden Summanden 3 und 5 aufgerufen. 3 und 5 sind also die Argumente, die der `add` Funktion übergeben werden.

Innerhalb der `add` -Funktion können die beiden Argumente mithilfe der Variablennamen `summand1` und `summand2` verwendet werden

```
1  def add(summand1, summand2):
2      sum = summand1 + summand2
3      print("The result of " + summand1 + " + " + summand2 + " is " + sum)
4
5
6  add(3,5)
```

```
1  The result of 3 + 5 is 8
```

Parameter & Argumente (II)

Im Normalfall muss die Anzahl der Parameter immer der Anzahl der Argumente entsprechen.

Werden weniger Argumente übergeben als Parameter definiert wurden, sind ein oder mehrere Werte unbesetzt, woraufhin der Interpreter einen Fehler wirft:

```
1  add(3)
2  TypeError: add() missing 1 required positional argument: 'summand2'
```

Werden mehr Argumente übergeben als Parameter definiert, kann die Funktion auch nicht ausgeführt werden:

```
1  add(3,5,2)
2  TypeError: add() takes 2 positional arguments but 3 were given
```

DHBW Duale Hochschule Baden-Württemberg

Function Overloading (I)

Achtung

Es ist in Python nicht möglich, zwei oder mehr unterschiedliche Funktionen mit gleichem Namen und gleicher Anzahl von Parametern aufzurufen, da die Funktionen anhand ihres Funktionskopfes nicht eindeutig unterscheidbar sind.

Es ist aber möglich, zwei oder mehr Funktionen mit gleichem Namen aber unterschiedlicher Anzahl an Parametern zu definieren, da beim Aufruf anhand der Anzahl der übergebenen Argumente darauf geschlossen werden kann, welche der Funktionen ausgeführt werden soll.

Dies wird als **Overlaoding** (bzw. Überladen) von Funktionen bezeichnet.

Function Overloading (II)

Beispiel

In diesem Beispiel wird die erste `add` -Funktion durch die zweite Funktion überschrieben, sodass letztendlich nur eine `add` -Funktion existiert.

```
1  def add(summand1, summand2):
2      print("Result of 1st function: " + str(summand1 + summand2))
3
4  def add(addend1, addend2):
5      print("Result of 2nd function: " + str(addend1 + addend2))
```

In diesem Beispiel koexistieren beide Funktionen, da sie eindeutig unterscheidbar sind:

```
1  def add(summand1, summand2):
2      print("Result of 1st function: " + str(summand1 + summand2))
3
4  def add(summand1, summand2, summand3):
5      print("Result of 2nd function: " + str(summand1 + summand2 + summand3))
```

Keyword Argumente (I)

Standardmäßig entspricht die Reihenfolge der übergebenen Argumente der Reihenfolge der Parameter, d.h. im obigen Beispiel wird das erste Argument `3` dem Parameter `summand1` und das zweite Argument `5` dem Parameter `summand2` zugewiesen.

Um dies noch offensichtlicher bzw. expliziter zu machen, können die Argumente auch zusammen mit der Parameterbezeichnung (mit `=` verbunden) übergeben werden, was als *Keyword Argumente* bezeichnet wird:

```
1  def add(summand1, summand2):
2      sum = summand1 + summand2
3      print("The result of " + summand1 + " + " + summand2 + " is " + sum)
4
5
6  add(summand1=3, summand2=5)
7
8  The result of 3 + 5 is 8
```

Keyword Argumente (II)

Auf diese Weise ist es auch möglich, die Reihenfolge der Argumente zu ändern (was jedoch vermieden werden sollte, da es der Übersichtlichkeit und einfachen Lesbarkeit des Codes widerspricht).

```
1  def add(summand1, summand2):
2      sum = summand1 + summand2
3      print("The result of " + summand1 + " + " + summand2 + " is " + sum)
4
5
6  add(summand2=5, summand1=3)
7
8  The result of 3 + 5 is 8
```

Obwohl hier der Summand 2 als erstes und Summand 1 als zweites übergeben wurden, wurden sie anhand der Parameter-Keywords trotzdem richtig zugeordnet.

Default Argumente (I)

Es ist auch möglich, einer Funktion weniger Argumente zu übergeben als Parameter definiert wurden, sofern die fehlenden Parameter Standardwerte besitzen.

Bei der Definition von Funktionen könnnen nämlich für die Parameter Standardwerte angegeben werden, die verwendet werden, wenn für den jeweiligen Parameter kein Argument übergeben wurde. Ist ist aber auch möglich, ein Argument für einen Parameter zu übergeben, für den ein Standardwert definiert wurde. In diesem Fall wird der Standardwert ignoriert und der übergebene Argumentwert genutzt.

Default Argumente (II)

Beispiel

```
1  def greet_user(name="stranger"):
2      print("Hello, nice to meet you, " + name)
```

Falls der Nutzer für den `name`-Parameter ein Argument übergibt, so wird dieses verwendet:

```
1  greet_user("Alice")
2  "Hello, nice to meet you, Alice"
```

Wird kein Argument übergeben, so wird der Standardwert ("stranger") verwendet:

```
1  greet_user()
2  "Hello, nice to meet you, stranger"
```

Default Argumente (III)

Achtung

Standardargumente müssen in der Funktionsdefinition immer **nach** den Pflichtargumenten stehen, da andernfalls ein Fehler geworfen wird.

Folgendes ist also nicht möglich:

```
1  def check_legal_age(legal_age=18, user_age):
2      if user_age >= legal_age:
3          print("User is full-age")
4      else:
5          print("User is under full-age")
6
7  SyntaxError: non-default argument follows default argument
```

Korrekt ist der Funktionskopf folgendermaßen:

```
1  def check_legal_age(user_age, legal_age=18):
2      ...
```

Rückgabewerte (I)

Häufig werden Funktionen genutzt, um wiederkehrende Berechnungen mit unterschiedlichen Argumenten durchzuführen.

Das Problem der bisher gezeigten Funktionen besteht darin, dass die Funktionen inhaltlich vom aufrufenden Code abgekapselt sind. Werte bzw. Variablen, die innerhalb einer Funktion erzeugt werden, sind nur innerhalb des jeweiligen Funktionsrumpfes gültig (*Scoping*). Wird beispielsweise innerhalb einer Funktion eine Summe berechnet und in einer Variablen gespeichert, ist die Variable außerhalb der Funktion nicht "sichtbar".

```
1  def add(summand1, summand2):
2      calculated_sum = summand1 + summand2
3
4  add(3,5)
5  print(calculated_sum)
6
7  NameError: name 'calculated_sum' is not defined
```

Rückgabewerte (II)

Jedoch sind genau diese Werte, die innerhalb von Funktionen berechnet werden, für die Entwicklung von Bedeutung, da diese Ergebnisse beispielsweise auch außerhalb der Funktionen für den jeweils folgenden Codeablauf benötigt werden.

Daher können Werte von Funktionen an die aufrufende Stelle zurückgeben, diese Werte werden als **Rückgabewerte** bzw. **Return Values** bezeichnet.

Sie können analog zu Argumenten verstanden werden, die als Eingabewerte für Funktionen genutzt werden. Anstatt Werte von außerhalb innerhalb des Funktionsrumpfs verfügbar zu machen werden durch Rückgabewerte Variablen innerhalb des Funktionsrumpfs außerhalb verfügbar gemacht.

DHBW
Duale Hochschule
Baden-Württemberg

Rückgabewerte (III)

Rückgabewerte werden durch das `return` Schlüsselwort an die Stelle zurückgegeben, an der die Funktion aufgerufen wurde. Dort kann der Wert beispielsweise in einer Variable gespeichert werden.

Beispiel

Im folgenden Beispiel wird innerhalb der `add` -Funktion eine Summe gebildet und in der `calculated_sum` -Variable gespeichert. Diese wird anschließend zurückgegeben an die Stelle, an die die Funktion aufgerufen wurde. Der Wert von `calculated_sum` wird also außerhalb der Funktion in der Variablen `result` gespeichert.

```
1  def add(summand1, summand2):
2      calculated_sum = summand1 + summand2
3      return calculated_sum
4
5  result = add(3,5)
6  print(result)
7  8
```

Rückgabewerte (IV)

Jede Funktion in Python gibt einen Wert zurück. Auch in den zuvor gezeigten Funktionen ohne explizite `return` -Anweisung wird ein Wert zurückgegeben.
Falls die Ausführung einer Funktion am Ende angelangt ist und kein `return` -Statement existiert, wird standardmäßig der Wert `None` zurückgegeben.

```
>>> def add(summand1, summand2):
...     calculated_sum = summand1 + summand2

>>> result = add(3,5)

>>> print(result)
None

>>> print(type(result))
<class 'NoneType'>
```

Rückgabewerte (V)

Achtung

Durch das `return`-Keyword wird die Ausführung einer Funktion immer abgebrochen und es wird zur Stelle des Funktionsaufrufs zurückgesprungen. Daher macht es keinen Sinn, wenn eine Funktion nach einem `return`-Statement weiteren Code enthält, da dieser ignoriert wird.

```
1  def add(summand1, summand2):
2      calculated_sum = summand1 + summand2
3      return calculated_sum
4
5      print(calculated_sum) # Dieser Code wird nicht ausgeführt
```

pass

Manchmal soll zwar eine Funktion bereits definiert werden, jedoch soll der Funktionsrumpf erst später implementiert werden. In solchen Fällen ist es nicht erlaubt, nur den Funktionskopf aufzuführen und den Rumpf leer zu lassen, da Python, um den Beginn und das Ende der Funktion zu erkennen, einen eingerückten Block erwartet.

```
1  def add(summand1, summand2):
2
3  IndentationError: expected an indented block
```

Um den Rumpf dennoch leer zu lassen und keinen Fehler zu erzeugen, kann das Platzhalter-Schlüsselwort `pass` genutzt werden. Es besitzt keinerlei Funktionalität, sondern dient nur dazu, einen (eingerückten) Block zu kennzeichnen, der eigentlich leer ist.

```
1  def add(summand1, summand2):
2      pass
```

Aufgaben I

1. Wozu werden Funktionen genutzt / benötigt?
2. Aus welchen 3 Teilen bestehen Funktionen?
3. Worin besteht der Unterschied zwischen Built-in Funktionen und eigenen Funktionen?
4. Worauf muss beim Aufruf von Funktionen prinzipiell geachtet werden?
5. Worin unterscheiden sich Funktionsdefinitionen und -aufrufe?
6. Worin besteht der Unterschied zwischen Parametern und Argumenten?

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Module (I)

Da insbesondere bei umfangreichen Projekten der Codeumfang sehr groß sein kann, ist es schwierig, die Übersichtlichkeit zu wahren und die Komplexität in der Struktur gering zu halten.

Den gesamten Code in einer einzigen Datei zu verwalten ist zwar für simple Projekte oder einfache Skripte ausreichend, für umfangreiche Arbeiten sollte dies aber vermieden werden.

DHBW
Duale Hochschule
Baden-Württemberg

Module (II)

Aus diesem Grund stehen in Python, wie in vielen anderen (modernen) Programmiersprachen auch, sogenannte **Module** zur Verfügung.[2]

Diese helfen dabei, den Code zu gliedern, strukturieren und separieren. So können zusammengehörige Funktionalitäten in gleichen Modulen untergebracht werden, während Funktionalitäten, die sich auf eine andere Domäne oder ein anderes Anwendungsgebiet beziehen, in andere, separate Module gegliedert werden.

[2] Die Bezeichnung kann sich zwischen den Programmiersprachen unterscheiden, z.B. *Pakete*, *Container*, ...

Module (III)

Funktionen und Variablen, die sich in Modulen befinden, können in anderen Dateien/Modulen importiert werden. Nachdem sie importiert wurden, stehen sie ganz normal in der jeweiligen Datei zur Verfügung. Sie verhalten sich also so, als wären sie in der aktuellen Datei enthalten.

Hierzu wird das `import` -Keyword genutzt.

Ein Modul besitzt den Namen der Datei, in der die Anweisungen gespeichert sind. Falls die Datei also `calculations.py` heißt, kann das Modul mit dem Namen `calculations` importiert werden:

```
1    import calculations
```

Module (IV)

Nachdem das Modul importiert wurde, sind die darin enthaltenen Funktionen und Anweisungen mit einem Punkt getrennt aufrufbar.

Falls also folgendes in der Datei `calculations.py` enthalten ist:

```
1  def add(summand1, summand2):
2      return summand1 + summand2
```

dann kann der Code in einer anderen Datei folgendermaßen ausgeführt werden:

```
1  import calculations
2
3  calculations.add(3,5)
```

Built-in Module I

Ebenso wie bei Funktionen gibt es auch Module, die bereits im Python-Interpreter vorhanden sind und vom Entwickler genutzt werden können. Entsprechend der Trennung der Funktionalitäten sind die Funktionalitäten gegliedert, z.B existieren folgende Module:

- **math**: Mathematische Funktionen wie `math.floor()`, `math.log()`, `math.tan()`, ...
- **time**: Funktionen zur Verarbeitung von Zeiten, Datumsangaben u.ä. wie `time.time()`, `time.sleep()`, ...
- **os**: Modul für plattformunabhängige Funktionalitäten wie `os.getlogin()`, `os.mkdir()`
- **sqlite3**: Funktionen zum Zugriff auf SQLite-Datenbanken wie `sqlite3.connect()`, `sqlite3.close()`, ...
- **threading** und multiprocessing : Funktionalitäten zum (Multi-)Threading und (Multi-)Processing wie `threading.main_thread()`, `multiprocessing.Process.run()`
- **json**: Modul für JSON-Funktionalitäten wie `json.load()`, `json.dump()`

Built-in Module II

- **http**: Modul, das mehrere Untermodule für Funktionalitäten bezogen auf HTTP enthält wie `http.client`, `http.server`, ...
- viele weitere (siehe Docs)

Beispiel

```
>>> import math
>>> import time

>>> print(math.log(1))
0.0

>>> print(time.time())
1580846245.522497
```

Inhaltsverzeichnis

DHBW
Duale Hochschule
Baden-Württemberg

Bibliographie I