

Einführung in die Programmierung

**Systematisch programmieren lernen
mit PYTHON**



“Everybody should learn to program a computer, because it teaches you how to think.” (Steve Jobs, 1995)

Inhaltsverzeichnis

1	Programme entwickeln	9
2	Variablen, Ausdrücke, Anweisungen	14
3	Funktionen	19
4	Gestaltung von Schnittstellen	24
5	Bedingungen und Rekursion	28
6	Funktionen mit Rückgabewert	33
7	Iteration	37
8	Strings	41
9	Wortspiele	43
10	Listen	45
11	Dictionaries	52
12	Tupel	56
13	Datenstrukturen	59
14	Dateien	61
19	(Einige) weitere PYTHON-Features	63

Warum ist es notwendig, dass Studierende der Informatik systematisch programmieren lernen?

- Sehr gute Programmierkenntnisse sind eine *Kernkompetenz aller Informatiker*. Sie bilden die Grundlage, um die weiterführenden Konzepte der Informatik zu verstehen und um diese praktisch umzusetzen. Sie eröffnen uns den prinzipiellen Zugang zu schätzungsweise zig Milliarden programmierbarer Geräte auf der Welt. Mit dieser universellen Fähigkeit können wir das Verhalten aller dieser Geräte bestimmen.
- Die wichtigste Fähigkeit in der Informatik ist das *Problemlösen*: Damit ist gemeint, ein Problem präzise zu formulieren, kreativ über Lösungen im Rahmen der vorhandenen Möglichkeiten nachzudenken, und eine Lösung klar auszudrücken und umzusetzen. Zum einen trainiert die Tätigkeit des Programmierens diese Fähigkeit in besonderem Maße, zum anderen ist das Programmieren selbst ein Werkzeug, um Lösungen zu realisieren.
- In der Informatik werden Methoden aus verschiedenen Disziplinen kombiniert und bilden so das Selbstverständnis des Faches als typische *Informatik-Denkweise*: Wie in der Mathematik verwenden wir Formalismen mit exakt festgelegter Bedeutung, wie in den Ingenieurwissenschaften entwerfen wir nützliche und zuverlässige Systeme mit technischer Kreativität, wie in den Naturwissenschaften betrachten wir komplexe Systeme und testen Hypothesen. Alle diese Aspekte treten beim Programmieren selbst auf, z.B. bei Syntax und Semantik von Programmiersprachen, beim Entwurf von Lösungsstrategien und bei der Analyse unserer Programme.
- *Zukunftssicherheit und Unabhängigkeit*: Praktisch alle Innovationen der digitalen Welt basieren auf Software. Programmierkenntnisse erlauben es uns, neue Technologien einzuordnen und diese zielgerichtet selbst zu verwenden. Dies gilt auch langfristig, denn Programmieren ist universell in dem Sinn, dass alles, was mit Computer überhaupt realisiert werden kann, mit den hier vermittelten Programmierkenntnissen möglich ist (das ist ein beweisbares Ergebnis aus der Theoretischen Informatik). Die enorme Nachfrage von Unternehmen nach Programmierkompetenzen bietet abwechslungsreiche Tätigkeiten in praktisch allen Anwendungsbereichen und führt zu persönlicher Unabhängigkeit derjenigen, die dieses Profil erfüllen.


Warum sind Programmierkenntnisse auch in Gegenwart generativer KI, die u.a. Code produzieren kann, unverzichtbar?

- *Interpretation der generierten Ausgaben:* Während generative KI beim Schreiben von Code helfen kann, ist es in der Informatik unerlässlich, die so generierten Ausgaben zu verstehen und zu interpretieren. Dies stellt sicher, dass wir Fehler identifizieren und korrigieren, die Leistung optimieren und den Code an spezifische Anforderungen anpassen können.
- *Fortgeschrittenes Problemlösen:* Generative KI kann standardmäßige Programmieraufgaben übernehmen, aber für komplexe, angepasste oder spezielle Probleme ist ein tiefes Verständnis der Programmierung notwendig. Wir müssen in der Lage sein, spezialisierten Code zu schreiben, der über die Fähigkeiten der KI hinausgeht.
- *Entwicklung und Verbesserung von KI-Systemen selbst:* Um KI-Modelle zu erstellen, zu trainieren und zu optimieren, ist eine solide Programmiergrundlage erforderlich. Wir müssen die Algorithmen und Datenstrukturen verstehen, die den KI-Systemen zugrunde liegen, um bestehende Technologien weiter zu verbessern.
- *Codequalität und -sicherheit:* Von der KI generierter Code muss auf Qualität, Sicherheit und Effizienz geprüft werden. Mit Programmierkenntnissen können wir außerdem sicherstellen, dass der Code frei von Schwachstellen ist, die ausgenutzt werden könnten.
- *Ethischer und verantwortungsvoller Einsatz von KI:* Das Verständnis der Programmierung hilft uns, die ethischen Implikationen von KI zu erfassen. So können wir sicherstellen, dass KI-Systeme verantwortungsvoll entwickelt und eingesetzt werden, unter Berücksichtigung von Datenschutz, Unvoreingenommenheit und Fairness.
- *Debugging und Wartung:* Selbst mit der Unterstützung von KI bleiben Debugging und Wartung von Code menschliche Aufgaben, die ein gründliches Verständnis der Programmierprinzipien erfordern. Wir benötigen diese Fähigkeiten, um den Code effektiv zu verwalten und Fehler zu beheben.

- *Innovationen über die KI-Fähigkeiten hinaus:* Während KI bei vielen Aufgaben helfen kann, bleibt die Fähigkeit, kreativ über neue Anwendungen von Technologie nachzudenken, eine einzigartig menschliche Eigenschaft. Programmierkenntnisse ermöglichen es uns, neue Lösungen zu erforschen und zu entwickeln, die KI möglicherweise noch nicht generieren kann. Mit Hilfe unserer Fähigkeit zur Abstraktion können wir größere Klassen konkreter Problemen lösen.
- *Interdisziplinäre Anwendungen:* Programmierkenntnisse ermöglichen es uns, das computergestützte Denken auf verschiedene Disziplinen anzuwenden. Ob in der Bioinformatik, im Finanzwesen oder in den Künsten, das Verständnis des Programmierens befähigt uns, Technologie in vielfältigen Bereichen zu nutzen.

Insgesamt ist Programmieren eine grundlegende Fähigkeit für Studierende der Informatik, die sicherstellt, dass sie in einer zunehmend von KI beeinflussten Ära fähig, vielseitig und innovativ bleiben.

Und noch eine Anmerkung zu *Wissen* und *Können* und *Problemlösen*

$$(\rightarrow x + 4) = 7u$$


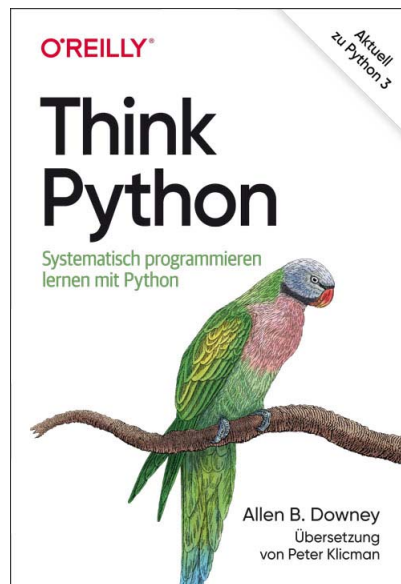
Lehrkonzept und Literaturempfehlung

Diese Einführung in die Programmierung basiert auf folgenden Überlegungen:

- Fokus auf Programmierung, nicht auf die Programmiersprache: Wir lernen die Programmiersprache PYTHON als Seiteneffekt. Es ist nicht das Ziel, PYTHON möglichst umfassend kennenzulernen. Stattdessen konzentrieren wir uns auf Programmierkonzepte, um diese (später) auch in anderen Programmiersprachen erkennen und anwenden zu können.
- So kurz wie möglich: Wir behandeln wenige, aber sehr wichtige Konzepte. Diese wollen wir vollständig verstehen - und nicht von möglichst vielen Dingen 'schon mal etwas gehört haben'.
- Üben, üben, üben: Man kann das Programmieren nur erlernen, wenn man es selbst tut und eine gewisse Routine entwickelt. Die Betonung liegt auf *selbst*!
 - Die Vorlesung (2 SWS) besteht zu großen Teilen aus *Live Programming* mit der Aufforderung, dass Sie dies später sorgfältig nachvollziehen. Sie sind explizit dazu eingeladen, selbst zu experimentieren und zu recherchieren: Variieren Sie die behandelten Beispiele und versuchen Sie nachzuvollziehen, was passiert.
 - In den Übungsstunden (2+2 SWS) werden Aufgaben gestellt, die Sie während der Übungen bearbeiten sollen. Hier ist es besonders wichtig, dass Sie die Aufgaben *selbst* lösen. Sie werden Ihre Lösungen in den Übungen auch vorstellen, denn das Sprechen über Code ist ebenfalls eine Übungssache.
Details sind wichtig!

Um dieses Lehrkonzept umzusetzen, basiert dieses Modul auf einem Buch, das die genannten Überlegungen unterstützt. Das hier vorliegende Skript kann als Begleittext dazu verstanden werden.

Think Python - Systematisch programmieren lernen mit Python von Allen B. Downey, dpunkt-Verlag / O'Reilly, Übersetzung der 2. englischen Auflage, ISBN Print: 978-3-96009-169-1, 2021



Dieses Buch kann hier erworben werden: <https://dpunkt.de/produkt/think-python/>

Verfügbarkeiten der englischen Version (2. Auflage):

- Homepage <https://greenteapress.com/wp/think-python-2e/>
- Als **PDF** und in **HTML**.

Die aktuelle **dritte** englischsprachige Auflage verwenden wir nicht.

1 Programme entwickeln

Ein *Programm*¹ ist ein strukturierter Text, der aus einer Folge von Anweisungen an einen Computer besteht. Die Anweisungen legen genau fest, wie eine Berechnung abläuft, wenn das Programm ausgeführt wird.

$$(3 \times 4) = 7u$$

```
167 def write_evaluation(id_tasks, id_name, groups_of_participant, task_name):
168     print("Ausgabe schreiben")
169     filename = idb + ".csv"
170     with open(filename, mode="w", newline="", encoding="utf-8") as out_file:
171         csv_writer = csv.writer(out_file, delimiter=";", quotechar="'", quoting=csv.QUOTE_MINIMAL)
172         csv_writer.writerow(["Participantname", "Name", "Gruppen", idb, Aufgaben:1 = task_name])
173     number = 0
174     for id, task_list in id_tasks.items():
175         number += 1
176         row = []
177         row.append(number)
178         row.append(id)
179         row.append(id_name[id])
180         row.append(groups_of_participant[id] if groups_of_participant is not None else [])
181         row.append("")
182         row.append("")
183         for task_id in task_list:
184             row.append(task_id)
185         else:
186             row.append("")
187     csv_writer.writerow(row)
188     print(f" (number) Zeilen geschrieben nach (filename)")
189
190 =====
191 = Auswertungsprozess
192 =====
193
194 # Aufrufen von n und Abgleich der aktuellen Übung erstellen
195 task_name = read_task_name()
196 dir_name = determine_assignment_directory()
197
198 # Teilnamen (Name und Task) auf (Gruppen) aufteilen
199 participants_lecture = read_participants(participants_file_lecture + ".csv")
200
201 groups_of_participant = None
202 if participants_file_groups is not None:
203     participants_group = read_participants_group()
204     groups_of_participant = determine_group_of_participants(participants_lecture, participants_group)
205     check_for_participants_not_in_group(participants_lecture, participants_group)
206
207 # Datenname validieren und gültige Aufgaben zusammenfassen
208 valid_filenames = validate_files(dir_name, task_name)
209 (id_tasks, id_name) = combine_assignments_per_id(valid_filenames)
210
211 # Teilnamen (Name und Task) auf (Gruppen) aufteilen
212 if participants_file_groups is not None:
213     copy_files_into_separate_directories(participants_lecture, dir_name, valid_filenames)
```

¹Zu allen hier verwendeten Begriffen wie *Programm*, *Anweisung*, *Berechnung*, usw. gibt es Definitionen, die deren Form (Syntax) und Bedeutung (Semantik) genau festlegen. Wir beschreiben die Begriffe hier weniger formal, aber es ist wichtig zu wissen, dass diesen Begriffen präzise Definitionen zugrunde liegen. Dies wird z.T. in späteren Modulen behandelt.

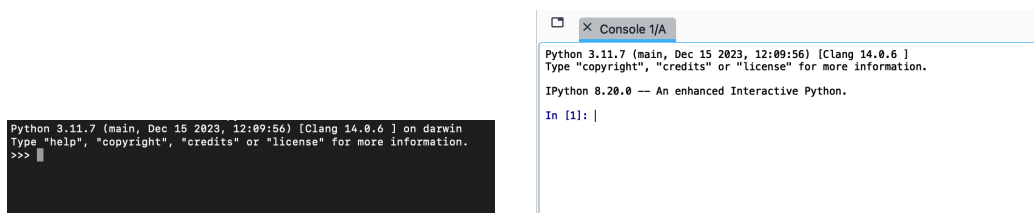
Es gibt nur wenige verschiedene Arten von Anweisungen:

- *Eingabe*: Programme nehmen mit Hilfe solcher Anweisungen während der Ausführung Daten von der Tastatur, aus Dateien, der Maus usw. entgegen.
- *Ausgabe*: Programme zeigen mit diesen Anweisungen Daten an, schreiben sie in eine Datei oder senden Daten an andere Geräte.
- *Mathematische Anweisungen*: Mit dieser Art von Anweisungen können wir den Computer wie einen Taschenrechner benutzen, d.h. Addieren, Subtrahieren, Multiplizieren usw.
- *Bedingte Ausführung*: Wir können festlegen, dass eine Anweisung nur ausgeführt wird, wenn in diesem Moment während der Berechnung eine bestimmte Bedingung gilt.
- *Wiederholung*: Wir können angeben, dass Anweisungen für eine bestimmte Zeit wiederholt werden (statt sie mehrfach hinzuschreiben). Das passiert meist mit jeweils leicht anderen Daten.

Das ist schon (fast) alles! Jedes Programm ist aus solchen Anweisungen aufgebaut, egal wie umfangreich die Funktionen sind.

Der PYTHON-*Interpreter* übernimmt die Ausführung von PYTHON-Anweisungen. Er nimmt Code entgegen und führt die zugehörige Berechnung unmittelbar aus.

$$6 \cdot 3 + 4 = 22$$



```
Python 3.11.7 (main, Dec 15 2023, 12:09:56) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>

Python 3.11.7 (main, Dec 15 2023, 12:09:56) [Clang 14.0.6 ]
Type "copyright", "credits" or "license" for more information.

IPython 8.20.0 -- An enhanced Interactive Python.
In [1]: |
```

Ausführung von Code an der *Eingabeaufforderung* (Prompt), das 'Hello, world!'-Programm und einige mathematische Operatoren



Daten repräsentieren Informationen, die übertragen, verarbeitet, gespeichert und verändert werden können. Eine konkrete Ausprägung ist ein *Wert*, z.B. ein Buchstabe oder eine Zahl. Wir haben einige Werte eben schon gesehen:

```
42      42.0      'Hallo, Welt!'      1679616
```

Die Daten unterscheiden sich darin, was wir mit ihnen anstellen können: Zwar wollen wir bei Bedarf zwei Zahlen dividieren, aber es macht keinen Sinn, einen Text durch einen anderen Text zu dividieren.

Deshalb wird die unendliche Menge von Werten in verschiedene Kategorien unterteilt, die sog. *Datentypen*. Das sind jeweils gleichartige Werte, auf die wir die gleichen Operationen anwenden können.

Wert	Datentyp	Bezeichnung in PYTHON
...
42	ganze Zahl (Integer)	int
42.0	Fließkommazahl	float
'42'	Zeichenkette (String)	str
-3	ganze Zahl (Integer)	int
0	ganze Zahl (Integer)	int
'A'	Zeichenkette (String)	str
3.777	Fließkommazahl	float
-0	ganze Zahl (Integer)	int
'Hallo, Welt!'	Zeichenkette (String)	str
"Hallo, Welt!"	Zeichenkette (String)	str
.0000010	Fließkommazahl	float
.000001	Fließkommazahl	float
000001	ganze Zahl (Integer)	int
0.000001	Fließkommazahl	float
+42	ganze Zahl (Integer)	int
'!#_*?=='	Zeichenkette (String)	str
' .000001'	Zeichenkette (String)	str
2.13.2	Syntaxfehler: kein zulässiger Wert	
...

In PYTHON haben alle Werte einen Datentyp, deshalb nennt man PYTHON *stark typisiert*.

Den Typ eines Wertes ermitteln



Programme sind in einer *formalen Sprache* geschrieben, die sehr strenge Vorgaben macht:

- Die einzelnen Bestandteile (*Tokens*) einer Programmiersprache entsprechen in etwa den einzelnen Wörtern unserer natürlichen Sprache. Tokens sind z.B. die oben aufgeführten Werte und die Anweisung **print**. Sie haben einen fest vorgegebenen Aufbau, den wir lernen und einhalten müssen (sonst Syntaxfehler s.o.)
- Darüber hinaus ist die *Struktur* eines Programms wichtig, d.h. wie die Tokens aneinander gereiht werden. Wir werden noch sehen, dass in PYTHON sogar die Anzahl der Leerzeichen an bestimmten Stellen eine Rolle spielt.

Beide Vorgaben, für Tokens und Struktur, legen die *Syntax* einer Programmiersprache fest. Dies geschieht mit dem Ziel, dass Programme von Computern gelesen und verstanden werden können. Dazu muss ein Programm syntaktisch korrekt sein, ansonsten kann es nicht ausgeführt werden.

Beispielsweise ist festgelegt, was während einer Berechnung passiert, wenn die Anweisung **print** in einem Programm an einer dafür passenden Stelle auftritt. Steht dort stattdessen `pirnt`, dann bricht die Berechnung an dieser Stelle mit einem Fehler ab, weil die Bedeutung von `pirnt` nicht festgelegt ist und der Rechner damit (zurecht) nichts anfangen kann.

Die Zuordnung einer Bedeutung zu allen syntaktisch korrekten Sprachbestandteilen einer Programmiersprache ergibt deren *Semantik*. Nur Programme mit korrekter Syntax haben eine Semantik, d.h. erst bei korrekter Syntax kann man feststellen, ob ein Programm auch während der Berechnung das Richtige tut.

Fehler in Programmen werden als *Bugs* bezeichnet und die Suche nach Fehlern in einem Programm ist das *Debugging*. Auf der Ebene der Syntax ist die Fehlersuche vergleichsweise einfach, hier gibt es oft passende Hinweise der Entwicklungsumgebung. Sehr viel schwieriger ist die Suche nach Fehlern in einem zwar syntaktisch korrekten Programm, das sich aber während einer Berechnung doch nicht so verhält, wie beabsichtigt. Die Berechnung erfolgt dann zwar strikt gemäß der Semantik, aber wir haben bei der Konstruktion des Programms einen Denkfehler gemacht.

Das Debugging geht mit dem Programmieren einher und muss geübt werden. Die Entwicklungsumgebung unterstützt dabei. Die Analyse des eigenen Codes während des Debuggens ist ein wichtiger Bestandteil des Programmierenlernens.

2 Variablen, Ausdrücke, Anweisungen

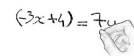
Eine *Variable* ist ein Name, der sich auf einen Wert bezieht. Während einer Berechnung kann sich eine Variable nacheinander auf verschiedene Werte beziehen, sodass der gleiche Name mal für diesen und mal für jenen Wert steht: variabel eben!

Eine neue Variable wird in PYTHON mit einer *Zuweisung* erstellt. Diese Zuweisung ordnet dem Namen erstmals einen Wert zu. Weitere Zuweisungen zum gleichen Namen ändern dann den zugeordneten Wert. Den jeweils aktuellen Wert können wir im *Variable Explorer* der IDE sehen.

Variablen und Zuweisungen



Es ist eine passende Vorstellung, dass eine Variable auf einen Wert 'zeigt', denn damit wird die aktuelle Zuordnung visualisiert. Wenn sich zugeordnete Werte ändern, können wir Pfeile 'umbiegen'.



Ein solches *Zustandsdiagramm* stellt die Belegung aller Variablen zu einem Zeitpunkt dar und beschreibt so die Gesamtsituation nach einer Reihe von Zuweisungen. Nach weiteren Zuweisungen sieht das Diagramm entsprechend anders aus.

Namen für Variablen müssen eine bestimmte Form haben. Zulässig sind nur endliche Folgen von Symbolen bestehend aus Buchstaben a, \dots, z, A, \dots, Z und Ziffern $0, \dots, 9$, sowie Unterstrich $_$. Die Folge muss mit einem Buchstaben oder Unterstrich beginnen und darf keines der folgenden Wörter sein:

not, and, or, if, else, while, for, in, range,
def, return, print, False, class, finally, is,
None, continue, lambda, try, True, from,
nonlocal, del, global, with, as, elif, yield,
assert, import, pass, break, except, raise

Für diese sog. *Schlüsselwörter* ist bereits eine andere Bedeutung reserviert.

Variablennamen



Ausdrücke sind Bestandteile eines Programms, die während einer Berechnung zu einem Wert ausgewertet werden. Wir haben schon Ausdrücke gesehen wie $6 * 7$, die stets zum gleichen Wert ausgewertet werden.

In Ausdrücken dürfen aber neben Werten und Operatoren auch Variablen vorkommen, so dass der resultierende Wert davon abhängt, welchen Wert die Variable in diesem Moment hat. Das vergrößert sofort unsere Möglichkeiten: Wir können Ausdrücke wiederverwenden und brauchen dazu bloß die Werte der vorkommenden Variablen ändern!

Ausdrücke haben keinen weiteren Effekt: An dieser Stelle im Code steht zur Laufzeit der resultierende Wert. Davon zu unterscheiden sind die *Anweisungen*: Das sind Bestandteile des Codes, die etwas bewirken. Wir kennen schon zwei Anweisungen: **print** erzeugt eine Ausgabe und eine Zuweisung $x = \dots$ erstellt eine neue Variable x oder ändert deren Wert, falls sie schon besteht.

Ausdrücke und Anweisungen



Beachten Sie: Zuweisungen sind keine Gleichungen im mathematischen Sinne, die genau dann wahr sind, wenn beide Seiten der Gleichung übereinstimmen. Vielmehr haben Sie den schon beschriebenen Effekt. Und es gibt sogar eine Reihenfolge: *Zuerst* wird der Ausdruck rechts vom Gleichheitszeichen ausgewertet, und dieser Wert dann *anschließend* der Variablen links vom Gleichheitszeichen zugeordnet.

Wir verwenden PYTHON bisher im *interaktiven Modus*. Dabei werden u.a. die resultierenden Werte von Ausdrücken unmittelbar ausgegeben, wenn sie ausgewertet werden. Das ist gut geeignet, um kleine Teile von Programmen zu testen und das werden wir noch oft verwenden.

Im sog. *Skriptmodus* wird eine gespeicherte PYTHON-Datei (Endung `.py`) als Ganzes ausgeführt. Dieses Szenario entspricht der Abbildung zu Beginn von Kapitel 1. Im Skriptmodus werden im Unterschied zum interaktiven Modus nur Ausgaben gemacht, die per Anweisung vorgegeben wurden (z.B. durch **print**).

Interaktiver und Skriptmodus



Wenn wir Ausdrücke aufschreiben, müssen wir auch die Reihenfolge der Auswertung festlegen, denn davon hängt unter Umständen das Ergebnis ab. Es ist nicht so, dass Operatoren stets von links nach rechts in Leserichtung ausgewertet werden, sondern wir orientieren uns an den üblichen Festlegungen der Rangfolge aus der Mathematik:

Klammern
vor
Exponenten
vor
Multiplikation und Division
vor
Addition und Subtraktion

Das dient vor allen Dingen dazu, Klammern zu sparen. Die müssten wir sonst sehr viel häufiger setzen, um die Auswertungsreihenfolge eindeutig zu machen.

Rangfolge von Operatoren



Wir unterscheiden Werte nach Typ, weil jeweils unterschiedliche Operationen angewendet werden. Bei Strings machen die obigen mathematischen Operatoren zum Beispiel keinen Sinn. Eine Ausnahme bilden `+` und `*`: Sie kommen auch im Zusammenhang mit Strings vor, haben dann aber eine andere Bedeutung.

String-Operationen



Das ein erstes Beispiel dafür, dass das gleiche Symbol mit verschiedenen Bedeutungen 'überladen' wird. Das kommt später häufiger vor. Die tatsächliche Bedeutung des Symbols ergibt sich eindeutig aus den beteiligten Typen. Deshalb ist es wichtig bei Variablen stets den Typ der zugeordneten Werte zu kennen.

Eine der häufigsten Fragen beim Lesen von Code lautet 'Was habe ich mir nur dabei gedacht?'. Das ist völlig normal und geht allen so, egal wieviel Programmiererfahrung vorhanden ist. Deshalb schreiben wir *Kommentare* in unseren Code, die genau diese Frage beantworten. Kommentare werden während der Ausführung des Codes ignoriert.

Das ist auch deshalb notwendig, weil sehr häufig Personen Code lesen, die ihn nicht selbst verfasst haben. Ohne gute Kommentare ist das sehr ineffizient, manchmal fast unmöglich.

Kommentare



Gute Kommentare zu schreiben ist eine Kunst. Auch das Schreiben von Kommentaren kann man üben und wir wollen von Anfang an darauf Wert legen. Eine gute Leitlinie ist es, im Kommentar auszudrücken, *warum* etwas im Code inhaltlich oder fachlich gemacht wird, im Unterschied zur verbalen Wiederholung *wie* etwas im Code umgesetzt ist. Letztere Art der Kommentare ist sinnlos, weil wir sowieso davon ausgehen, dass der Leser oder die Leserin die Programmiersprache beherrscht. Gute Kommentare gehen Hand in Hand mit der Wahl sprechender Namen für Variablen.

Was hier zunächst harmlos erscheint, ist ein großer wirtschaftlicher Faktor, weil die Herstellungskosten von Software sehr hoch sind. Um diese Investitionen zu schützen muss Code unbedingt leicht verständlich sein, damit Wartung und Weiterentwicklung effizient gelingen.

Wir unterscheiden in Programmen drei Arten von Fehlern:

Syntaxfehler Sie treten auf, wenn wir gegen die Vorgaben für Tokens und für die Struktur von Programmen verstoßen. Dann ist das Programm nicht ausführbar, weil dessen Bedeutung nicht ermittelt werden kann. Sie treten am Anfang am häufigsten auf, sind aber auch am einfachsten zu finden. Oft gibt der Editor der IDE bereits Hinweise während der Eingabe des Codes.

Laufzeitfehler Solche Fehler werden erst während der Ausführung eines Programms bemerkt. Die Ursache könnten z.B. unerwartete Eingabedaten sein, auf die das Programm nicht vorbereitet ist und mit denen es folglich nicht umgehen kann. Solche *Ausnahmen* können unter Umständen im Programm selbst abgefangen werden, später mehr dazu.

Semantische Fehler Wenn ein Programm während einer Berechnung sich nicht so verhält, wie wir das möchten, dann handelt es sich um einen semantischen Fehler. Das ist meistens eine Diskrepanz zwischen dem, was wir eigentlich programmieren *wollten*, und dem, was wir tatsächlich programmiert *haben*. Diese Fehler sind besonders schwierig zu finden, weil ihnen meist ein Denkfehler zugrunde liegt, den wir zunächst einsehen müssen.

3 Funktionen

Eine *Funktion* ist eine Folge von Anweisungen, die unter einem Namen zusammengefasst werden. Um diese Anweisungen auszuführen, wird an anderer Stelle lediglich der Name 'aufgerufen'. Es gibt sehr viele Funktionen, die bereits in PYTHON eingebaut sind. Wir können aber auch selber Funktionen definieren, indem wir den Namen der Funktion und die zugehörigen Anweisungen vorgeben. So strukturieren wir unseren Code und vermeiden es, den gleichen Code mehrfach hinzuschreiben, falls wir ihn öfter benötigen.

Ein *Funktionsaufruf* besteht aus dem Namen der Funktion, gefolgt von den Argumenten, die wir an die Funktion übergeben wollen. Die Argumente stehen beim Funktionsaufruf nach dem Funktionsnamen in Klammern. Wieviele Argumente von einer Funktion erwartet werden, ist in deren Definition festgelegt. Eine Funktion kann als Ergebnis einen Rückgabewert zurückliefern.

Funktionsaufrufe



Wir haben bisher Operatoren für Grundrechenarten gesehen. Daneben stehen in PYTHON viele weitere mathematische Funktionen bereits zur Verfügung. Bevor wir sie nutzen können, müssen wir ein sog. Modul in unseren Code importieren. Das ist eine Datei, die eine Sammlung von zusammengehörigen Funktionsdefinitionen enthält. Danach können wir die dort enthaltenen Funktionen ebenfalls aufrufen.

Durch die *Komposition* von Funktionen können wir kompliziertere Formeln kompakt hinschreiben und ausrechnen, weil Funktionsaufrufe und Ausdrücke als Argumente für andere Funktionsaufrufe verwendet werden dürfen. Bei der Auswertung ist wieder die Reihenfolge wichtig: 'von innen nach außen'. *Zuerst* wird der Ausdruck im Argument ausgewertet und *danach* die Funktion mit dem resultierenden Wert aufgerufen.

Mathematische Funktionen und Komposition



Mit einer *Funktionsdefinition* erstellen wir eine neue Funktion. Das wird auch *Deklaration* der Funktion genannt. Sie besteht aus einer Kopfzeile (Header) und dem Rumpf (Body). Die Kopfzeile ist immer gleich aufgebaut: Erst das Schlüsselwort **def**, dann der Funktionsname (gleiche Regeln wie bei Variablennamen, siehe oben), gefolgt von den erwarteten Argumenten in Klammern und schließlich ein Doppelpunkt. Der Rumpf muss eingerückt sein (Konvention: genau 4 Leerzeichen!) und besteht aus einer beliebigen Folge von Anweisungen. Die Deklaration wird durch eine Leerzeile beendet.

Eigene Funktionen



Mit der Strukturierung von Code durch Funktionen ändert sich die Reihenfolge, in der die Anweisungen eines Programms ausgeführt werden: Der *Programmablauf* erfolgt nicht mehr nacheinander von oben nach unten. Es geht zwar immer oben bei der ersten Anweisung los, aber Funktionsaufrufe sind 'Umleitungen' im Programmablauf. Bei einem Aufruf springt der Programmablauf in den Body der Funktionsdefinition, führt die dortigen Anweisungen aus, und kehrt dann zur aufrufenden Stelle zurück.

Die Deklaration einer Funktion ändert den Programmablauf nicht. Erst beim Aufruf der Funktion werden die Anweisungen im Body ausgeführt. Durch die Deklaration wird aber der Funktionsname bekannt gemacht, so dass ein nachfolgender Aufruf möglich wird.

$\rightarrow x + 4 = 7$

```
3
4
5  ## Version 1
6  def zeige_text():
7      print('Veronika, der Lenz ist da.')
8      print('Die Mädchen singen trallala.')
9
10 def wiederhole_refrain():
11     zeige_text()
12     zeige_text()
13
14 wiederhole_refrain()
15
```

Um zu verstehen was ein Programm macht, müssen wir die Leserichtung am Programmablauf orientieren!

Wie wir schon gesehen haben, können wir beim Funktionsaufruf *Argumente* an die Funktion übergeben. Dies können Werte, Variablen oder Ausdrücke sein. Bevor die Ausführung der Funktion beginnt, werden die Argumente aber stets zu Werten ausgewertet. Diese Werte werden anschließend innerhalb der Funktion den Variablen zugewiesen, die in der Funktionsdefinition im Header als *Parameter* angegeben sind.

Die Namen von Argumenten einerseits und Parametern andererseits sind unabhängig voneinander. Parameter und weitere Variablen, die innerhalb einer Funktion angelegt werden, existieren nur dort und werden *lokale Variablen* genannt.


Argumente und Parameter



Stapeldiagramme dienen der Veranschaulichung von geschachtelten Funktionsaufrufen. Dabei werden alle Variablen und die zugeordneten Werte wie in einem Zustandsdiagramm dargestellt, aber mit der zusätzliche Information, zu welcher Funktion sie gehören. Dies wird durch einen Rahmen (*Frame*) dargestellt.

Die Anordnung der Frames ist wichtig: Funktionsaufrufe erfolgen von oben nach unten. Der oberste Frame außerhalb jeder Funktion heißt `__main__`, ganz unten steht die aktuell ausgeführte Funktion.

Stapeldiagramme

$$(-3 \times 4) = 7$$


```
163 zeile1 = 'Bing tiddle '
164 zeile2 = 'tiddle bang.'
165 zweimal_concat(zeile1, zeile2)
```

```
158 def zweimal_concat(teil1, teil2):
159     concat = teil1 + teil2
160     print_zweimal(concat)
161
```

```
122 def print_zweimal(peter):
123     print(peter)
124     print(peter)
---
```

Ist die Ausführung eines Funktionsrumpfes beendet, wird auch der zugehörige Eintrag im Frame wieder entfernt und es geht zurück zur aufrufenden Stelle. Während der Aufbau des Stapels von oben nach unten erfolgt, geschieht der Abbau von unten nach oben in umgekehrter Aufrufreihenfolge. Im Fehlerfall gibt PYTHON die Liste der aktuell ausgeführten Funktionen gemäß Stapel zurück, den sog. *Traceback*.

Unsere bisherigen selbst definierten Funktionen haben *keinen Rückgabewert*, sondern sie führen lediglich die Aktionen im Body aus. Wenn wir das Ergebnis eines solchen Funktionsaufruf trotzdem einer Variablen zuweisen, wird der Variable der Wert `None` zugeordnet.

Andere Funktionen, wie beispielsweise die mathematischen Funktionen aus `math`, sind *mit Rückgabewert*, d.h. sie liefern das Berechnungsergebnis an die aufrufende Stelle zurück. Das Ergebnis ordnen wir meist einer Variablen zu oder verwenden es in einem zusammengesetzten Ausdruck weiter. Im interaktiven Modus wird ein Rückgabewert einer Funktion unmittelbar ausgegeben, im Skript-Modus dagegen verschwindet er, sofern er nicht in einer Zuweisung oder einem Ausdruck weiterverwendet wird.

Funktionen mit und ohne Rückgabewerte



Wir werden etwas später auch eigene Funktionen mit Rückgabewert schreiben.

Vorteile der Verwendung von Funktionen in Programmen:

- Übersichtliche Strukturierung von Programmen für bessere Lesbarkeit und einfachere Fehlersuche.
- Programme werden kürzer, weil die Wiederholung von gleichem oder ähnlichem Code vermieden werden kann.
- Einzelne Teile eines Programms können unabhängig voneinander entwickelt werden, ggf. von verschiedenen Personen, um sie später zu einem funktionierenden Ganzen zusammenzufügen.
- Funktionen können in mehreren Programmen verwendet werden, müssen aber nur einmal entwickelt und getestet werden, siehe Modul `math`.

4 Gestaltung von Schnittstellen

Wir entwickeln schrittweise einige Funktionen zum Zeichnen geometrischer Objekte und führen dabei wichtige Konzepte der Programmierung ein. Das `turtle`-Modul bietet die notwendigen Grundfunktionen, die wir zum Zeichnen verwenden wollen. Mit der **for**-Anweisung (*Schleife*) können wir einfache Wiederholungen kompakt aufschreiben.

Schildkröte, wiederhole das!



Das Auslagern von Code in eine Funktion ist ein Beispiel für *Datenkapselung* (Encapsulation, auch Information Hiding): Der Zugriff auf die Anweisungen im Body der Funktion ist nur über den Funktionsaufruf möglich. Andererseits kann der gleiche Code für verschiedene Argumente ausgeführt werden.

Datenkapselung



Wenn wir eine Funktion so erweitern, dass sie weiterhin alles das kann, was sie bisher auch schon konnte, aber jetzt eben noch mehr, dann ist das eine *Generalisierung* (Verallgemeinerung). So können wir unseren Code viel öfter einsetzen, auch in Szenarien, an die wir jetzt vielleicht noch nicht denken.

Generalisierung



Generalisierungen erfordern i.d.R. zusätzliche Anpassungen: Der Funktionsname soll weiterhin gut dokumentieren, was die Funktion macht. Deshalb, und wegen der veränderten Anzahl der Argumente, müssen auch die Funktionsaufrufe angepasst werden. Um die Bedeutung der Argumente an der aufrufenden Stelle zu dokumentieren, können wir in PYTHON bei Bedarf *Schlüsselwort-Argumente* verwenden. Sie halten die Argument-Parameter-Zuordnung explizit fest.

Weitere Generalisierung



Die *Schnittstelle* einer Funktion fasst zusammen, wie die Funktion verwendet wird:

- Wie heißen die Parameter?
- Was macht die Funktion?
- Was ist der Rückgabewert?

Interessant ist dabei vor allen Dingen, was mit der Schnittstelle *nicht* beschrieben wird: Es bleibt alleine der Funktion selbst überlassen, *wie* das Resultat berechnet wird. Das erlaubt Änderungen an der Art und Weise der Implementierung, ohne dass die Schnittstelle und damit der Funktionsaufruf geändert werden muss.

Außerdem ist bei der Gestaltung von Schnittstellen zu beachten, dass sie einerseits durch Generalisierung allgemein genug sind, andererseits aber auch 'schmal' bleiben, um die Funktion einfach benutzen zu können.

Gestaltung von Schnittstellen




Oft kommt man mit Generalisierung alleine nicht weiter, wenn neue Funktionen hinzu kommen. Dann muss zuweilen ein größerer Teil des bestehenden Codes neu arrangiert werden, um insgesamt eine gute Lösung mit passenden Schnittstellen und hoher Wiederverwendung zu erreichen: Das *Refactoring*.

Refactoring



Refactoring: Vorher/Nachher-Bild

$$(3x+4) = 7u$$


Die gezeigte Vorgehensweise lässt sich iterieren und ist dann besonders gut geeignet, wenn eine Aufteilung in Funktionen noch nicht von vorne herein klar ist:

1. Beginne mit kleinem Codeschnipsel ohne Funktionen.
2. Sobald es funktionieren, die inhaltlich zusammengehörende Anweisungen in einer Funktion kapseln.
3. Generalisiere die Funktion, um weitere Anforderungen umzusetzen.
4. Wiederhole Schritte 1-3, bis ein funktionierendes Ganzes entsteht.
5. Prüfe zwischendurch regelmäßig auf mögliche Verbesserungen durch Refactoring: Gibt es ähnlichen Code mehrfach?

Dieser *Vorgehensweise* zum Schreiben von Programmen liegen die Konzepte Datenkapselung, Generalisierung und Refactoring zugrunde. Es gibt alternative Vorgehensweisen, wie wir später noch sehen werden.

Ein *Docstring* ist eine mehrzeilige Zeichenkette zwischen Header und Body einer Funktion, mit der die Schnittstelle erklärt wird. Dort wird kurz und knapp beschrieben, was die Funktion macht, welche Parameter erwartet werden und was ggf. zurückgeliefert wird. Wenn die Formulierung des Docstrings schwerfällt, könnte das ein Anzeichen dafür sein, dass die Schnittstelle verbesserungswürdig ist.

Docstrings



Die Schnittstelle ist ein Vertrag zwischen der Funktion und dem Aufrufenden.

- Der Aufrufende willigt ein, die Parameter so bereitzustellen, wie sie in der Schnittstelle beschrieben sind: Das ist die *Vorbedingung*.
- Die Funktion hingegen sichert zu, die dokumentierte Aufgabe zu erledigen, sofern die Vorbedingung erfüllt ist. Das ist die *Nachbedingung*.

Über die Schnittstelle werden damit die Zuständigkeiten klar geregelt: Tritt ein Fehler in der Funktion auf, weil die Vorbedingung nicht erfüllt ist, liegt die Verantwortung beim Aufrufenden. Wurde andererseits die Vorbedingung erfüllt, aber die Nachbedingung wird verletzt, ist die Funktion verantwortlich. Das funktioniert nur, wenn die Schnittstelle klar beschrieben ist!

5 Bedingungen und Rekursion

Mit den Operatoren für ganzzahlige Division (*Floor-Division*) und für den Rest bei ganzzahliger Division (*Modulo-Operator*) erhalten wir stets ein Ergebnis vom Typ **int**, sofern beide Argumente diesen Typ haben.

Ganzzahlige Division und Rest



Wir haben bisher Ausdrücke kennengelernt, die während der Ausführung des Programms zu einer Zahl ausgewertet wurden. Bei *Booleschen Ausdrücken* ist das Ergebnis dagegen einer von genau zwei Wahrheitswerten: **True** oder **False**. Solche Ausdrücke werden auch *Bedingungen* genannt und sind vom Typ **bool**.

Der einfachste Operator, der zu einem Wahrheitswert führt, ist der Vergleich **==**, nicht zu verwechseln mit der Zuweisung **=** in PYTHON. Weitere Operatoren haben die gleiche Bedeutung, wie die entsprechenden mathematischen Relations-symbole. Die Schlüsselwörter **and**, **or** und **not** sind *logische Operatoren*, die Boolesche Ausdrücke verknüpfen.

Boolesche Ausdrücke und Operatoren



Sehr häufig möchte man den Programmablauf in Abhängigkeit von einer Bedingung steuern. Dazu dienen die *bedingten Anweisungen*, die es in mehreren Varianten gibt. Bei der *Alternative* wird ein Entweder-oder ausgedrückt. In einer *Verkettung* werden mehrere Bedingungen nacheinander geprüft: Sobald eine **True** ergibt wird der eingerückte Code ausgeführt und die Verkettung beendet. Außerdem dürfen bedingte Anweisungen geschachtelt werden, um so mehrere Bedingungen zu kombinieren.


Bedingte Anweisungen



Das sind Beispiele für zusammengesetzte Anweisungen (*Verbundanweisungen*), die aus Bedingungen und anderen Anweisungen bestehen und zusammen eine neue Anweisung ergeben.

Bedingte Anweisungen werden auch *Verzweigungen* genannt, denn sie führen im Programmablauf zu verschiedenen Zweigen, von denen dann während der Ausführung abhängig von den aktuellen Daten ein Zweig abgelaufen wird.

Verzweigungen durch bedingte Anweisungen

$$(\rightarrow x + 4) = 7$$


Funktionen können andere Funktionen aufrufen, das haben wir schon gesehen. Sie dürfen sich sogar selbst aufrufen, das nennt man *Rekursion*. Selbstaufrufe machen nur Sinn, wenn sich dabei mindestens ein Argument ändert, damit nicht immer wieder dasselbe passiert.

Rekursion



Auch bei Selbstaufrufen können wir den Überblick mit Hilfe eines Stapeldiagramms behalten. Folglich gibt es mehrere Frames mit gleichem Namen im Stapel. Falls keine weiteren rekursiven Aufrufe mehr erfolgen, ist der *Basisfall* erreicht, der auch *Rekursionsanker* genannt wird.

Programmablauf und Stapeldiagramm bei Selbstaufrufen

$$(-3 \times 4) = 7$$

```
163 def countdown(n):
164     if n == 0:
165         print('Bumm!')
166     else:
167         print(n)
168         countdown(n-1)
169
```

```
163 def countdown(n):
164     if n == 0:
165         print('Bumm!')
166     else:
167         print(n)
168         countdown(n-1)
169
```

```
163 def countdown(n):
164     if n == 0:
165         print('Bumm!')
166     else:
167         print(n)
168         countdown(n-1)
169
```

```
163 def countdown(n):
164     if n == 0:
165         print('Bumm!')
166     else:
167         print(n)
168         countdown(n-1)
169
```

Falls der Basisfall nie erreicht wird, setzen sich rekursive Aufrufe unendlich oft fort, zumindest theoretisch. Denn wenn der für den Stapel vorgesehene Speicherplatz verbraucht ist, meldet PYTHON das mit riesigem Traceback. Das ist in der Regel unbeabsichtigt und ein Programmierfehler: Oft fehlt dann der Basisfall oder wird nicht erreicht.

Endlose Rekursion



Um Tastatureingaben vom Benutzer entgegen zu nehmen, gibt es in PYTHON die eingebaute Funktion **input** (), der wir auch eine Eingabeaufforderung mitgeben können. Ggf. müssen wir die Eingabe in den für uns passenden Datentyp umwandeln und mit unerwarteten Eingaben umgehen (dazu später mehr).

Tastatureingaben



6 Funktionen mit Rückgabewert

Mit der **return**-Anweisung können wir Funktionen verlassen, und gleichzeitig einen Wert an die aufrufende Stelle zurückgeben. Dabei sollten wir 'toten Code' (*Dead Code*) vermeiden, der unter keinen Umständen erreicht wird. Außerdem wollen wir stets so programmieren, dass auf jedem Zweig eine **return**-Anweisung mit Rückgabewert passenden Typs durchlaufen wird.

Rückgabewerte



Eine weitere Vorgehensweise bei der Programmierung ist die *inkrementelle Entwicklung*. Dabei ist das Ziel, langwieriges Debugging zu vermeiden:

1. Beginne mit einem lauffähigen Programm und nehme schrittweise kleine Änderungen vor. Teste nach jeder Änderung.
Falls ein Fehler auftritt, liegt die Ursache meist bei der letzten Ergänzung, und ist so leichter zu lokalisieren.
2. Verwende temporäre Variablen für Zwischenwerte, damit sie angezeigt und überprüft werden können.
3. Sobald das Programm funktioniert, entferne den *Scaffolding-Code* (= Variablen für Zwischenwerte, zusätzliche **print**-Anweisungen), solange das Programm dadurch nicht schwerer lesbar wird.

Beispiel: Entfernung in der Ebene

$$(-3x + 4) = 7$$

Inkrementelle Entwicklung



Wir haben inzwischen schon mehrfach Funktionen aus anderen Funktionen heraus aufgerufen. Dies nennt man *Funktionskomposition* und bedeutet die Hintereinanderausführung (auch Verkettung) von Funktionen. Der Aufruf von Funktionen mit booleschem Rückgabewert ist besonders nützlich an Stellen, an denen eine umfangreichere Bedingung gebraucht wird.

Komposition und boolesche Funktionen



Wir kennen zwar erst einen Bruchteil von PYTHON, aber bereits mit diesen Sprachmitteln kann man alles das berechnen, was man mit Computern überhaupt berechnen kann! ² Das ist ein Resultat der *Theoretischen Informatik* und wird dadurch ausgedrückt, dass unser bisheriges PYTHON-Fragment als *Turing-vollständig* bezeichnet wird (benannt nach Alan M. Turing, 1912 - 1954). Wir hören in späteren Modulen mehr zur Ausdrucksstärke von Programmiersprachen und dem Begriff der Berechenbarkeit.


Insbesondere die Möglichkeit der Rekursion ist sehr ausdrucksstark, wenn es um Berechnungen geht. Falls eine mathematische Funktion bereits rekursiv, d.h. basierend auf Vorgängerwerten derselben Funktion, definiert ist, dann können wir deren Funktionswerte mit Hilfe der Rekursion in Programmiersprachen unmittelbar ausrechnen. Wir müssen dann lediglich die mathematischen Notationen in die Programmiersprache übertragen.

Mehr Rekursion



²Genau genommen genügt sogar noch weniger!

Programmablauf und Stapeldiagramm mit Rückgabewert

$(3 \times 4) = 7u$ 

```
199 def fakultaet(n):  
200     if n == 0:  
201         return 1  
202     else:  
203         ergebnis = n * fakultaet(n-1)  
204         return ergebnis  
205
```

```
199 def fakultaet(n):  
200     if n == 0:  
201         return 1  
202     else:  
203         ergebnis = n * fakultaet(n-1)  
204         return ergebnis  
205
```

```
199 def fakultaet(n):  
200     if n == 0:  
201         return 1  
202     else:  
203         ergebnis = n * fakultaet(n-1)  
204         return ergebnis  
205
```

```
199 def fakultaet(n):  
200     if n == 0:  
201         return 1  
202     else:  
203         ergebnis = n * fakultaet(n-1)  
204         return ergebnis  
205
```

Sehr oft vertrauen wir beim Lesen von Code darauf, dass wir beim Aufruf eingebauter Funktionen oder von Funktionen, die wir aus anderen Modulen importieren, ein korrektes Ergebnis erhalten, und zwar ohne dass wir uns die zugrunde liegende Implementierung nochmals ansehen.

Das sollten wir uns auch angewöhnen, wenn wir den Code einer rekursiven Funktion lesen und prüfen: *Angenommen* der Selbstaufruf liefert das korrekte Ergebnis für $n-1$, ist dann der Rückgabewert für n richtig? Zusammen mit einem korrekten Basisfall ergibt dieser *Vertrauensvorschuss* eine insgesamt korrekte rekursive Implementierung.³

Ein Beispiel mit Vertrauen



Manchmal entstehen Fehler bei der Verwendung von Funktionen, weil die Vorbedingungen nicht ausreichend dokumentiert sind (Docstring!). Dann liegt die Verantwortung bei der Funktion: Die Argumente müssen geprüft und ggf. abgefangen werden. Das übernehmen *Wächter* zu Beginn des Bodys. Sie tragen allerdings zur Laufzeit bei, obwohl sie bei den weiteren Selbstaufrufen nicht notwendig wären.

Typprüfung




³Mit diesem Konzept kann man sogar *beweisen*, dass eine rekursive Implementierung korrekt ist. Dabei wird die Beweisform der vollständigen Induktion verwendet. Darüber erfahren wir mehr in anderen Modulen.

7 Iteration

Genau genommen ist eine Variable in PYTHON eine *Referenz* (Verweis) auf ein Objekt, deshalb sprechen wir in Zustandsdiagrammen von einer 'Zuordnung' und zeichnen einen Pfeil zwischen Variablenname und Objekt. Das hilft uns nachzuvollziehen, was bei mehreren Zuweisungen hintereinander passiert.

Mehrere Zuweisungen

$$(-3x + 4) = 7u$$


Zuweisungen und Speicherorte



Die erste Zuweisung eines Wertes an eine Variable heißt *Initialisierung*. Sehr häufig werden Variablen nach der Initialisierung *aktualisiert*, d.h. der neuer Wert ist abhängig vom bisherigen Wert. Eine Erhöhung um 1 heißt *Inkrement*, eine Subtraktion von 1 ist ein *Dekrement*.

int-Werte sind unveränderbar (*immutable*) und jeder Wert existiert im Speicher höchstens einmal. Das gilt auch für weitere Datentypen wie z.B. **float** und **str**. Davon zu unterscheiden sind veränderbare Datentypen (*mutable*), die wir später kennenlernen werden.

Initialisierung und Aktualisierung



Zuweisungen und Speicherorte



Bisher haben wir gleiche oder ähnliche Anweisungen wiederholt, indem wir Rekursion oder die **for**-Anweisung (später mehr dazu) verwendet haben. Das sind zwei Arten der *Iteration*, eine weitere ist die **while**-Anweisung.

while-Anweisung



Programmablauf:

1. Werte die Bedingung im Header aus.
2. Falls Ergebnis `False`, gehe zur nächsten Anweisung nach der **while**-Anweisung.
3. Falls Ergebnis `True`, führe den Body aus und gehe danach zu Schritt 1.

$$x = x + 4 = 7$$

Die Schleifenbedingung wird zu Beginn und nach jedem Durchlauf *erneut ausgewertet*. Erst wenn die Bedingung irgendwann nicht mehr erfüllt ist, wird die Schleife beendet (sie *terminiert*). Es könnte sein, dass dieser Fall nie eintritt. Dann haben wir eine *Endlosschleife* programmiert, was in den meisten Fällen unerwünscht ist.

Ob eine Endlosschleife auftreten kann oder nicht, ist manchmal nicht so einfach festzustellen. Mit der **break**-Anweisung können wir Schleifen vorzeitig verlassen.

Weitere Beispiele



Das Newton-Verfahren ist ein Beispiel für einen *Algorithmus*: Eine eindeutige Handlungsvorschrift zur Lösung eines Problems. Es ist wichtig, dass wir die beiden Begriffe strikt auseinander halten:

- Ein *Problem* beschreibt präzise, WAS zu berechnen ist (hier: die Quadratwurzel einer Zahl).
- Ein *Algorithmus* beantwortet die Frage, WIE wir das Problem lösen können (hier: Iteration der Formel).

In der Informatik geht es oft um beides: Zunächst muss das Problem präzise formuliert werden, um dann einen Algorithmus zur Lösung des Problems zu entwickeln und zu programmieren. ⁴

Newton: Stoppkriterium

$$(-3x+4) = f(x)$$

⁴Manche Probleme sind so schwierig, dass es keine Algorithmen gibt, die sie lösen können. Wir haben uns eben gefragt, ob eine **while**-Schleife terminiert. Es gibt in der Theoretischen Informatik einen Beweis für die Aussage, dass kein Algorithmus existiert, der diese Frage für alle Programme korrekt beantwortet.

8 Strings

Ein String (Zeichenkette) ist eine *Folge* von Zeichen, auf dessen Bestandteile wir mit dem Klammeroperator unter Angabe eines *Index* (Position) einzeln zugreifen können. Die Nummern der Positionen beginnt bei Null, so dass der Index des letzten Symbols in der Folge nicht gleich der Anzahl der Zeichen in einem String ist. Mit einer Schleife können wir leicht über Strings iterieren und diese so *traversieren*.

$\text{f}[3 \times 4] = \text{fu}$

Strings und Zugriff per Index



Per *Slicing* erhalten wir ein Teilstück einer Zeichenkette. Das ist immer nur ein lesender Zugriff, denn in PYTHON sind auch die Strings *unveränderbar*. Eine Änderung an einem String bedeutet, dass wir einen neuen String an einem anderen Speicherort erzeugen.

Slicing und Unveränderbarkeit



Slicing und Zustandsdiagramm

$\text{f}[3 \times 4] = \text{fu}$

Wir demonstrieren zwei Verarbeitungsmuster, die häufig vorkommen: Bei einer *Suche* wird per Schleife über einen Bereich iteriert und sobald das Gesuchte gefunden ist, wird die Schleife beendet. Mit einem *Zähler* iterieren wir über den gesamten Bereich bis zum Schluß und merken uns, wie oft ein bestimmtes Ereignis auftritt.

Suche und Zähler



Eine *Methode* ist ähnlich einer eingebauten Funktion: Sie erwartet Argumente und liefert etwas zurück. Zusätzlich ist sie an ein Objekt gebunden, das als Argument dient (die Methode wird 'auf' dem Objekt aufgerufen). Für Strings stehen in PYTHON bereits viele nützliche Methoden zur Verfügung. Hinzu kommen diverse (relationale) Operatoren.

String-Methoden und -Operatoren



Indexfehler sind manchmal schwer zu finden

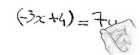


9 Wortspiele

Übungen 9-1 bis 9-6



Indexbereiche und Rekursion




```
198 # Alternative 2: Rekursion
199 def ist_alphabetisch(wort):
200     if len(wort) <= 1:
201         return True
202     if wort[0] > wort[1]:
203         return False
204     return ist_alphabetisch(wort[1:])
205
```

```
198 # Alternative 2: Rekursion
199 def ist_alphabetisch(wort):
200     if len(wort) <= 1:
201         return True
202     if wort[0] > wort[1]:
203         return False
204     return ist_alphabetisch(wort[1:])
205
```

```
198 # Alternative 2: Rekursion
199 def ist_alphabetisch(wort):
200     if len(wort) <= 1:
201         return True
202     if wort[0] > wort[1]:
203         return False
204     return ist_alphabetisch(wort[1:])
205
```

Eine besonders elegante Vorgehensweise zur Lösung eines Problems ist der Rückgriff auf eine bereits vorhandene Lösung für ein anderes, aber ähnliches Problem. Das ist nicht nur effizient, weil wir weniger Code schreiben müssen. Sondern wenn wir im Laufe der Zeit einen besseren Algorithmus für ein Problem finden, dann profitieren auch gleich alle Probleme davon, die auf dieses reduziert wurden.

Reduktion⁵

$$f(3x+4) = f_u$$


Testfälle: lange/kurze Eingaben, positiv/negativ, Sonderfälle⁶



“Durch Testen kann man stets nur die Anwesenheit, nie aber die Abwesenheit von Fehlern beweisen.” (Edsger W. Dijkstra, 1972)

⁵Das allgemeiner Konzept der *Reduktion* von einem Problem auf ein anderes ist grundlegend und tritt in der Informatik an verschiedenen Stellen auf, z.B. um die Komplexität von Problemen zu vergleichen. Insbesondere ist eine Reduktion möglich, ohne dass Algorithmen zur Lösung der beteiligten Probleme bekannt sein oder existieren müssen. Das wird in späteren Modulen nochmals aufgegriffen.

⁶Die systematische Erstellung von Testfällen, die einen bestimmten Grad der *Überdeckung* eines Programms gewährleisten, wird in späteren Modulen ausführlicher behandelt. Dazu gehört auch die effiziente Verwaltung von sehr vielen Testfällen und deren Sollergebnissen, so dass diese automatisiert und wiederholt ausgeführt werden können.

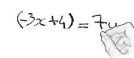
10 Listen

Eine *Liste* (Typ **list**) ist eine Folge von Werten, den sog. *Elementen* der Liste. Der Zugriff erfolgt wie bei Strings per Klammeroperator und Index $0, 1, \dots$. Die Elemente können im Unterschied zu Strings von beliebigen Typs sein. Außerdem können wir Listen ändern (sie sind *mutable*) und wir dürfen Listen *verschachteln*, d.h. ein Element einer Liste kann wieder eine Liste sein.

Listen anlegen, verschachteln und ändern



Zustandsdiagramm für Listen



Mit der **for**-Anweisung iterieren wir über die Liste und durchlaufen alle Elemente. Alternativ können wir mit Hilfe von **len** und **range** den Index auch explizit verwenden. Das *Slicing* kennen wir schon von Strings und funktioniert genauso. Listen-Methoden ändern die Listen, auf denen sie aufgerufen werden, und liefern lediglich `None` zurück.

Iteration, Operationen, Slicing und Listen-Methoden



Typische Aufgaben rund um Listen umfassen das Reduzieren, Filtern oder Umwandeln aller Elemente einer Liste. Für das Löschen von Elementen stehen verschiedene Möglichkeiten zur Verfügung, je nachdem ob der Index oder das Element bekannt sind. Listen und Strings scheinen sehr ähnlich, haben aber verschiedenen Typ. Sie können leicht ineinander umgewandelt werden.

Reduzieren, Filtern, Umwandeln und Löschen, Vergleich mit Strings



Wir haben schon erwähnt, dass es die Werte von unveränderbaren Datentypen im Speicher nur einmal gibt (z.B. **int** und **str**). Bei veränderbaren Datentypen können die gleichen Werte dagegen mehrfach vorkommen (z.B. **list**). Um entscheiden zu können, wann zwei Dinge 'gleich' sind, müssen wir daher genauer zwischen *Objekten* und *Werten* unterscheiden: In PYTHON ist alles ein Objekt und ein Objekt hat einen Wert.

Wenn zwei Objekte den gleichen Wert haben und dieser Wert ist von ...

- ... unveränderbarem Typ: Dann handelt es sich um dasselbe Objekt (= identische Stelle im Speicher), denn der Wert des Objekts existiert nur einmal.
- ... veränderbarem Typ: Dann sind die Objekte nicht identisch. Ändert sich der Wert des einen Objekts, dann hat das keine Auswirkungen auf den Wert des anderen Objekts.

Verweisen zwei Variablen auf dasselbe Objekt, spricht man von *Aliasing*, denn das Objekt ist dann über mehrere Namen gleichermaßen ansprechbar.

Objekte und Werte

$$\mapsto x+4 = 7u$$

Gleich oder nicht gleich



Beim Aufruf einer Funktion in PYTHON erfolgt die Übergabe der Argumente an die Parameter der Funktion stets als sog. *Call by Object Reference*: Nach der Übergabe referenziert der korrespondierende Parameter auf dasselbe Objekt wie das Argument. Wenn der Wert dieses Objekts innerhalb der Funktion geändert wird, hat dies Auswirkungen an der aufrufenden Stelle, sofern es sich um einen änderbaren Datentyp handelt.

Verweist ein Argument beim Funktionsaufruf auf einen Wert von

- ... unveränderbarem Typ: Dann referenziert das Argument auch nach dem Ende der Ausführung der Funktion auf denselben Wert wie vorher.
- ... veränderbarem Typ: Dann liegt eine Form von Aliasing vor, denn sowohl über das Argument (an der aufrufenden Stelle) als auch über den Parameter (in der Funktion) wird dasselbe Objekt referenziert. Ändert die Funktion während der Ausführung den Wert dieses Objekts, dann ändert sich dadurch auch der Wert, auf den das Argument nach Beendigung der Funktion verweist.

Daher ist wichtig zu unterscheiden zwischen Operationen, die Objekte verändern, und Operationen, die Kopien der Objekte erzeugen. Und natürlich sollte man wissen, ob man es mit einem veränderbaren oder unveränderbaren Datentyp zu tun hat.

Funktionsaufrufe mit Listen



Zugehörige Zustandsdiagramme

$$f(x+4) = 7u$$

11 Dictionaries

Ein *Dictionary* (Typ **dict**) ist eine Menge von Schlüssel-Wert-Paaren. Im Unterschied zu Listen kann der Schlüsselwert von irgendeinem unveränderbaren Typ sein. Der Zugriff erfolgt über den Schlüsselwert per Klammeroperator. Dictionaries sind *änderbar*, aber im Unterschied zu Listen sind die Schlüssel-Wert-Paare nicht geordnet, sondern in einer Menge zusammengefasst.

Dictionaries anlegen und ändern



Um einen Wert per **in**-Operator in einer Liste zu suchen, müssen wir die Liste vollständig durchlaufen, d.h. die Dauer der Suche ist abhängig von der Anzahl der Elemente in der Liste. Suchen wir dagegen einen Schlüsselwert per **in**-Operator in einem Dictionary, wird aus dem Schlüssel ein *Hashwert* ermittelt (deshalb muss dieser unveränderbar sein) und wir können unabhängig von der Anzahl der Elemente im Dictionary nachsehen, ob der Wert vorhanden ist oder nicht. Das geht i.d.R. schneller. Außerdem ist die Flexibilität beim Typ der Schlüsselwerte oft von Vorteil.

Implementierung eines Histogramms



Dictionaries eignen sich besonders gut, um *endliche Funktionen* zu repräsentieren, die wir aus der Mathematik kennen: Jedem Argument (Schlüssel) wird ein Funktionswert (Wert) zugeordnet. Die endliche Menge aller Schlüssel im Dictionary ist dann der *Definitionsereich* der Funktion, die endliche Menge aller Werte im Dictionary ist der *Wertebereich* der Funktion.

Da dieselben Werte mehrfach vorkommen können, ist die Umkehrrelation nicht unbedingt wieder eine Funktion im mathematischen Sinne. Wir können sie aber trotzdem als Dictionary repräsentieren, wenn wir einen geeigneten Datentyp wählen.

Mit der **raise**-Anweisung können wir selbst Ausnahmen auslösen, inkl. Traceback und (eigener) Fehlermeldung.

Iteration und inverse Suche




Zugehörige Zustandsdiagramme

$$f(x+4) = f(x)$$

Eine weitere nützliche Anwendung von Dictionaries ist die Verwendung als globale *LookUp-Tabelle*, um die mehrfache Berechnung gleicher Aufgaben zu vermeiden, wie sie häufig bei rekursiven Implementierungen auftritt. Die Schlüssel-Wert-Paare bestehen dann aus dem Parameter des rekursiven Aufrufs (Schlüssel) und dem Berechnungsergebnis (Wert). Falls das Ergebnis schon einmal ermittelt wurde, ersetzen wir jeden weiteren rekursiven Aufruf durch den in der Tabelle gespeicherten Wert.

Dieses Konzept heißt *Memoization* und wird häufig mit Rekursion kombiniert, um die Einfachheit einer rekursiven Implementierung beibehalten zu können und dabei gleichzeitig Redundanz zu vermeiden. Auch hier ist es günstig, dass wir beim Typ für die Schlüsselwerte nicht wie bei Listen an einen strikten Indexbereich gebunden sind.

Aufrufbaum und Redundanz

$$f(3 \times 4) = 70$$


Rekursion mit Memoization



Im vorherigen Beispiel haben wir innerhalb einer Funktion auf eine Variable zugegriffen, der dort zuvor kein Wert zugeordnet wurde. Stattdessen wurde die Variable im äußeren Frame `__main__` per Zuweisung deklariert und damit zur *globalen Variable*. In PYTHON gelten die folgenden Regeln:

- Variablen, die innerhalb einer Funktion nur referenziert werden (d.h. es erfolgt keine Zuweisung), sind implizit global. Fehlt eine Zuweisung auch in `__main__`, gibt es eine Fehlermeldung.
- Variablen, an die *irgendwo* innerhalb einer Funktion eine Zuweisung erfolgt oder als Parameter auftreten, sind *lokale Variablen* der Funktion und existieren nur während deren Ausführung. Soll es sich trotz Zuweisung innerhalb der Funktion um eine globale Variable handeln, müssen wir dies mit dem Schlüsselwort **global** explizit in der Funktion angeben.

Globale und lokale Variablen



Zustandsdiagramme

$$(-3 \times 4) = 7$$

```

248  bekannt = {0: 0, 1: 1}
249
250  # keine Zuweisung an bekannt
251  def Beispiel5():
252      bekannt[2] = 1
253
254  Beispiel5()
255  print(bekannt)

```

```

260  bekannt = {0: 0, 1: 1}
261
262  def Beispiel6():
263      bekannt = {}
264      bekannt[2] = 100
265      print(bekannt)
266
267  Beispiel6()
268  print(bekannt)

```

12 Tupel

Ein *Tupel* (Typ **tuple**) ist eine Sequenz (Folge) von Werten, ebenso wie Strings und Listen. Sequenzen haben viele Eigenschaften gemeinsam, z.B. den Indexbereich, den Zugriff auf die Elemente per Klammer-Operator, die Iteration per **for**-Anweisung, das Slicing und auch die Bedeutung der relationalen Operatoren.

Mit Listen haben Tupel gemein, dass ihre Elemente beliebigen Typs sein können, mit Strings teilen sie die Eigenschaft, dass sie unveränderbar sind. Tupel werden u.a. überall da verwendet, wo man eigentlich eine Liste bräuchte, aber der Datentyp unveränderbar sein muss.

Tupel anlegen, ändern und vergleichen



Wir können Tupel verwenden, um mehrere Werte gleichzeitig genauso vielen Variablen zuzuweisen (*Tupel-Zuweisung*). Jede Funktion liefert nur einen Rückgabewert, aber wir können bei Bedarf mehrere Werte in einem Tupel verpacken. Und wir verwenden Tupel zusammen mit dem *****-Operator, wenn wir die Anzahl der Argumente beim Funktionsaufruf variabel halten wollen.

Tupel-Zuweisungen, Rückgabewerte und Argumente



Ein *Iterator* ist ein Objekt, über das wir mit der **for**-Anweisung iterieren und so alle Elemente der zugrundeliegenden Sequenz durchlaufen können. Wenn wir über zwei Listen gleichzeitig iterieren wollen, können wir diese zunächst mit der eingebauten **zip**-Funktion zu einem solchen Iterator zusammenfassen und anschließend die korrespondierenden Elemente aus beiden Listen durchlaufen. Es gibt vielfältige Möglichkeiten um Listen, Tupel und Dictionaries zu kombinieren.

Kombination von Listen, Tupeln und Dictionaries



Zustandsdiagramme für Tupel

$$f(x) = 3x + 4 = 7u$$

Listen, Tupel und Dictionaries sind Beispiele für *einfache Datenstrukturen*, die wir je nach Einsatzzweck anhand ihrer Eigenschaften auswählen: Wenn wir in PYTHON beispielsweise sehr viele Strings verändern müssen, aber nicht immer wieder neue Strings anlegen wollen, dann bieten sich Listen aus Buchstaben als Alternative an. In einem anderen Fall möchten wir vielleicht beim Aufruf einer Funktion Seiteneffekte durch Aliasing unbedingt vermeiden, also verwenden wir als Argument lieber ein unveränderbares Tupel statt einer veränderbaren Liste.

Zusammengesetzte Datenstrukturen bestehen typischerweise aus Kombinationen einfacher Datenstrukturen und werden je nach Bedarf konstruiert, z.B. Listen aus Listen als Repräsentation einer Matrix, oder Dictionaries mit Werten vom Typ **dict**, um in einem Adressverzeichnis zu dem Vor- und Nachname (Schlüssel) die Adresse in strukturierter Form abzulegen.⁷

Es gibt allerdings wie immer auch eine Kehrseite der Medaille: Je komplizierter die Struktur der Daten, desto schwieriger wird unter Umständen die Fehlersuche.

⁷Sie werden in späteren Modulen noch viele Datenstrukturen und deren Eigenschaften kennenlernen.

13 Datenstrukturen

Beispiele zur Auswahl und Verwendung von Datenstrukturen.

Worthistogramm für "Die Buddenbrooks": Übungen 13-1 und 13-2



Wir können auch benutzerdefinierte Funktionen mit *optionalen Parametern* versehen. Sie stehen in der Liste der Parameter im Header der Funktion hinter den erforderlichen Parametern. Ein optionaler Parameter wird dadurch gekennzeichnet, dass ein *Standardwert* im Header angegeben ist.

Übungen 13-3 und 13-4



Alle Berechnungen sind grundsätzlich *deterministisch*, d.h. bei gleichen Eingaben (und gleichen sonstigen Rahmenbedingungen) erhalten wir dieselben Berechnungsergebnisse.⁸ Manchmal möchten wir absichtlich *Zufall* in die Berechnungen einbauen, z.B. bei der Implementierung von Spielen. Über das PYTHON-Modul `random` stehen uns dazu Algorithmen zur Verfügung, die u.a. *Pseudozufallszahlen* generieren. Die nennen wir ab jetzt vereinfachend 'Zufallszahlen'.

Zufallszahlen und Übung 13-5



⁸Manchmal scheint dies nicht zu stimmen: Tatsächlich liegt ein nicht reproduzierbares Verhalten häufig daran, dass komplexe Rahmenbedingungen nicht identisch vorliegen.

[aus Kap. 12] Ein weiterer Standarddatentyp in PYTHON ist **set**, mit dem wir endliche Mengen repräsentieren können. Er hat die Eigenschaften, die wir von Mengen aus der Mathematik kennen: Jedes Element kommt höchstens einmal vor und es gibt keine feste Reihenfolge der Elemente.

Mengen selbst sind änderbar, aber die Elemente in einer Menge müssen unveränderbar sein (weil viele **set**-Operationen wieder auf Hashing basieren). Um Mengen von Mengen zu repräsentieren, steht der spezielle unveränderbare Typ **frozenset** zur Verfügung.

Mengen und Übung 13-6



Typische Kriterien für die Auswahl von Datenstrukturen:

- *Einfachheit* der Implementierung: Identifiziere die notwendigen Operationen auf den Daten zur Lösung der anstehenden Aufgabe. Mit welcher Datenstruktur können diese Operationen am einfachsten implementiert werden?
- *Effizienz*: Häufig verwendete Operationen sollten auf der ausgewählten Datenstruktur möglichst effizient ausgeführt werden können. Bei mehreren Möglichkeiten beginnt man mit einer Datenstruktur und vergleicht die Laufzeiten dann im *Benchmarking* mit den Alternativen.
- *Speicherplatz*: Redundante Ablage von Informationen gilt es zu vermeiden. Dadurch könnte es nicht nur zur Engpässen im Speicher kommen, sondern das führt auch zu mehrfachem Änderungsaufwand an verschiedenen Stellen. Andererseits liegt häufig ein *Trade-off* zwischen Laufzeit und Speicherverbrauch vor: Redundante Daten z.B. in LookUp-Tabellen stehen schnell zur Verfügung.

14 Dateien

Bisher können wir keine Daten über mehrere Berechnungen hinweg speichern. Um Daten *persistent* vorzuhalten, müssen wir sie dauerhaft aus einem Programm heraus auf Speichermedien ablegen und bei Bedarf von dort wieder einlesen. Eine einfache Form ist das Speichern (und Lesen) von Informationen in Form von Textdateien im Dateisystem des Rechners.

Dabei brauchen wir auch Möglichkeiten, um verschiedene Ablageorte zu wählen und um Daten in die Textform zu bringen. Neben den *f-Strings* gibt es in PYTHON sehr viele Möglichkeiten der Formatierung von Zeichenketten.

Textdateien lesen und schreiben



Wir haben schon häufig Fehlermeldungen gesehen: Dann wird die Ausführung des Programms beendet und die Meldung samt Traceback ausgegeben. Uns bleibt dann nichts anderes übrig, als das Programm so zu ändern, dass kein Fehler mehr auftritt.

Insbesondere beim Lesen und Schreiben von Dateien sind Fehler nicht immer vorhersehbar, und vorher auf alle Eventualitäten zu prüfen ist sehr aufwändig. Mit der **try/except**-Anweisung können wir Fehler innerhalb des Programmablaufs behandeln.

Ausnahmen abfangen und behandeln



*Datenbanken*⁹ sind Systeme, die auf die persistente Speicherung von strukturierten Daten unterschiedlicher Formate spezialisiert sind. Wir können sie aus einem Programm heraus über eine *Schnittstelle* ansprechen und u.a. Daten lesen und schreiben. Die Daten sind i.d.R. ähnlich strukturiert wie in einem Dictionary: Einem Schlüsselwert sind weitere Daten zugeordnet.

Bevor Datenstrukturen in Datenbanken gespeichert werden können, muss i.d.R. eine *Serialisierung* stattfinden. Bei der hier verwendeten Datenbank ist vorgegeben, dass Schlüssel und Werte vom Typ `bytes` sein müssen. Also wandeln wir unsere strukturierten Daten zunächst in einen Byte-Strom um, und zwar so, dass beim späteren Einlesen die Struktur wieder hergestellt werden kann (*Deserialisierung*).

Datenbanken



Beim Import von selbst entwickelten Modulen wollen wir i.d.R. die dort deklarierten Funktionen verwenden. Dazu wird die importierte Datei ausgeführt, insbesondere die Deklarationen. Während der Entwicklung entsteht im Modul aber auch Code zum Testen dieser Funktionen, der ohne weitere Vorkehrungen bei jedem Import ebenfalls ausgeführt würde.

PYTHON-Module als Skript oder als Import



⁹Dazu gibt es ein separates Modul.

19 (Einige) weitere PYTHON-Features

Code wird sehr viel häufiger gelesen als geschrieben (nach **Guido van Rossum**, Erfinder von PYTHON), daher ist die *Lesbarkeit* von Code ein wichtiges Ziel beim Programmieren. Wie jede andere Programmiersprache erlaubt auch PYTHON viele Freiheitsgrad, um Code zu schreiben. Wichtige Hinweise, was man besser tun oder lassen sollte, finden sich im *Python Enhancement Proposal* 8 von Guido van Rossum:

Style Guide for Python Code

Wir stellen noch einige PYTHON-Features vor, die die Lesbarkeit von Code erhöhen können, wenn sie sinnvoll angewendet werden.

- Mit *bedingten Ausdrücken* können wir Fallunterscheidungen direkt in Ausdrücke einbauen.
- *List Comprehension* erlaubt eine kompakte Schreibweise für das Erzeugen und Filtern von Listen (existiert auch als *Set/Tuple/Dictionary Comprehension*).
- *Generator-Ausdrücke* erzeugen während der Iteration das jeweils nächste Element erst auf Anforderung.
- Und die mehrstelligen logischen Operatoren **all** und **any** als Verallgemeinerungen der zweistelligen **and** bzw. **or** sind häufig besser lesbar, als das Iterieren über mehrere Boolesche Werte.

Weitere Features

