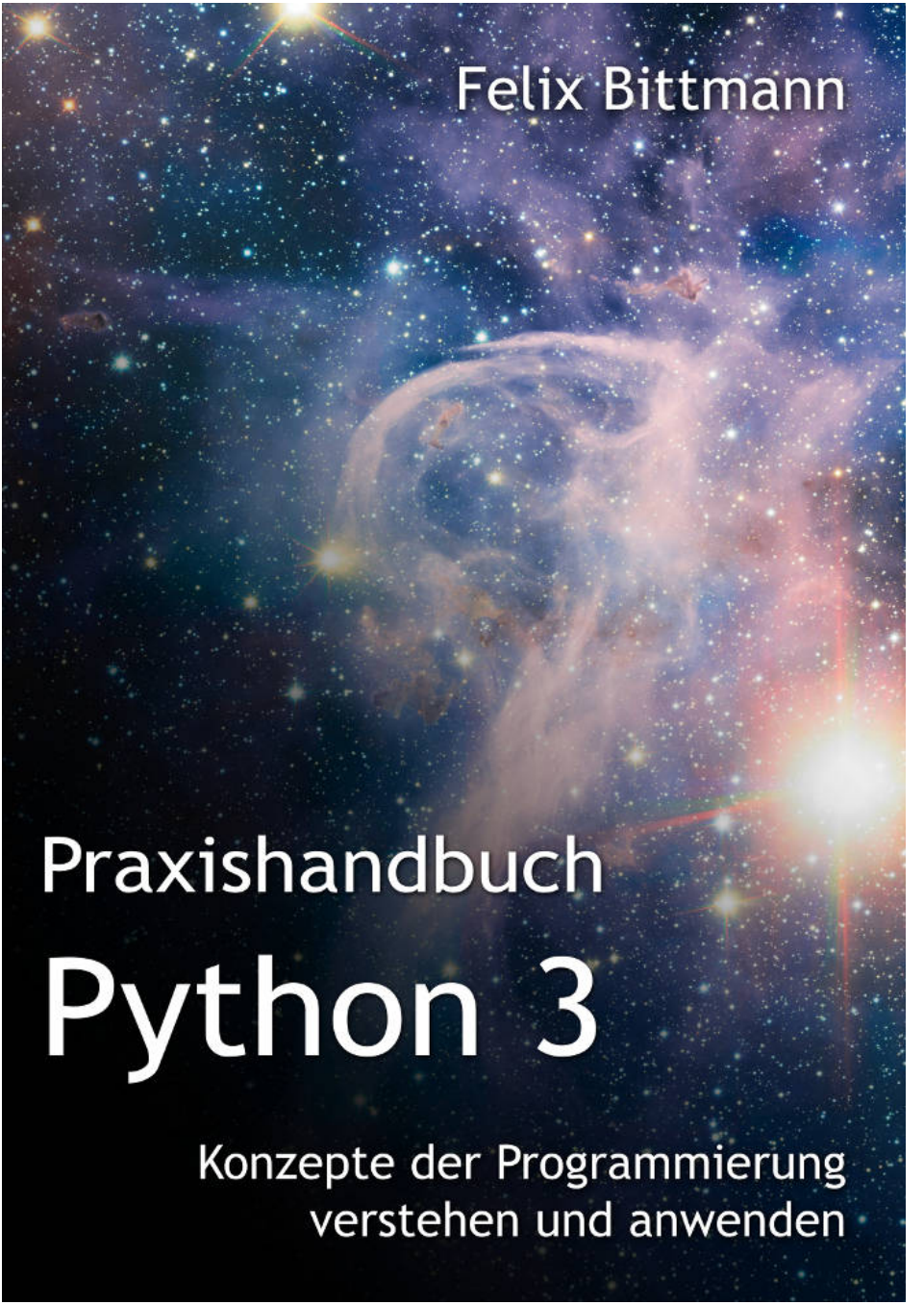
Felix Bittmann
Praxishandbuch
Python 3
Konzepte der Programmierung
verstehen und anwenden

Praxishandbuch Python 3

Das Buch
Wer die Grundlagen von Python beherrscht und jetzt tiefer einsteigen möchte, kommt in diesem Buch auf seine Kosten. Mittels konkreter Anwendungsbeispiele aus verschiedenen Fachgebieten wird aufgezeigt, wie man Python produktiv zur Problemlösung einsetzen kann. Diskutiert werden dabei neben den allgemeinen Lösungsstrategien auch die Spezifika von Python und wie diese gewinnbringend genutzt werden können. Somit veranschaulicht das Buch allgemeine Konzepte der Programmierung, wie beispielsweise Algorithmen, Rekursion und Datenstrukturen, und lehrt problemorientiertes Denken.

Der Autor
Felix Bittmann studierte Soziologie und Geschichte in Freiburg (Breisgau) und Bamberg und ist wissenschaftlicher Mitarbeiter an der Universität Bamberg. Python begleitet ihn seit vielen Jahren als Allzweckwaffe in Beruf und Alltag. Ebenfalls erhältlich sind *Soziologie der Zukunft* (2014) sowie *Stata - A Really Short Introduction* (2019).

Praxishandbuch Python 3

Konzepte der Programmierung verstehen und anwenden

Felix Bittmann

Bibliografische Informationen der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Titelabbildung: Carinanebel. Urheber: *European Southern Observatory* (Wikimedia Commons).

Herstellung und Verlag:
BoD - Books on Demand, Norderstedt. Gesetzt in LaTeX

Mai 2020
ISBN: 978-3751900584

www.felix-bittmann.de
www.github.com/fbittmann/Python

Inhaltsverzeichnis

Einleitung

Warum Python?

Python ist nicht ohne Grund zu einer der beliebtesten Programmiersprachen der Welt aufgestiegen. Eine nutzerfreundliche und intuitive Syntax, eine große und engagierte Community, sowie eine Vielzahl an Modulen und Bibliotheken, die eine zügige und effiziente Umsetzung der geplanten Projekte erlauben, begeistern Anfänger wie Profis gleichermaßen. Somit ist Python einerseits eine ideale erste Sprache, die es Anfängern erlaubt, rasch eigene Ideen umzusetzen und andererseits eine hervorragende Ergänzung für Profis, die etwa auf dem Gebiet der *Data Sciences* Fuß fassen wollen.

Dieses Buch richtet sich an Leserinnen und Leser, die bereits erste Erfahrungen mit Python haben, etwa nach dem Durcharbeiten eines Grundkurses, und nun lernen möchten, wie man Python produktiv und anwendungsbezogen umsetzt. Dabei halten Sie kein klassisches Lehrbuch in Händen, das verschiedene Funktionen bzw. Aspekte der Sprache nacheinander abhandelt, vielmehr werden konkrete Aufgaben und Anwendungen in den Fokus gerückt. Jeder Abschnitt besteht dabei aus einer Aufgabe oder einem Problem, das gelöst werden möchte. Diese sind verschiedenen Fachgebieten entnommen und bilden so eine breite Palette der denkbaren Einsatzgebiete ab. Erklärt wird dabei nicht nur die Lösungsidee,

sondern natürlich auch, wie sich diese mit den spezifischen Tricks und Tools in Python umsetzen lässt.

Voraussetzungen

Damit Sie an diesem Buch Freude haben, sollten Sie über die Grundfunktionen von Python informiert sein und beispielsweise die gängigen Datentypen kennen (Ganzzahlen, Kommazahlen, Strings, Listen, *Dictionaries*). Was ist eine Variable, wie kann ich Elemente zu einer Liste hinzufügen und wie definiere ich eine Funktion? Wenn Sie diese Fragen beantworten können, werden die Aufgaben für Sie lösbar sein. Für Anfänger oder für Personen, die ihre Kenntnisse auffrischen wollen, empfiehlt sich der kompakte Kurs der University of Waterloo.[1]

Philosophie

Die hier gezeigten Aufgaben richten sich an Anfänger, die bisher noch wenig Erfahrung mit Programmierung haben. Sofern bestimmtes Grundwissen bzw. mathematische Kenntnisse zum Lösen der Aufgaben notwendig sind, so werden diese ebenfalls vorgestellt. Die gezeigten Codebeispiele haben nicht den Anspruch, die elegantesten, kürzesten oder schnellen Lösungen zu sein, sondern sollen vielmehr die grundlegenden Ideen und Konzepte der Programmierung vermitteln. Für viele der dargestellten Anwendungen stehen hochspezialisierte und deutlich komplexere Algorithmen bereit, die in einer Produktivumgebung zum Einsatz kommen, jedoch für eine Einführung nicht geeignet sind.

Dabei werden zur Lösung keinerlei Zusatzprogramme oder Module benötigt, die nicht offizieller Teil Pythons sind (Pure Python).

[1] `https://cscircles.cemc.uwaterloo.ca/de/`

Insofern sei darauf hingewiesen, dass manche Module die hier gezeigten Anwendungsgebiete massiv erweitern und dadurch auch vereinfachen (z.B. *NumPy*, *SciPy*, *Pygame*, etc...), jedoch oftmals eine sehr umfassende Dokumentation beinhalten und daher an dieser Stelle nicht vorgestellt werden können.

Im Allgemeinen wurde versucht, die Aufgaben (zumindest innerhalb eines Kapitels) so anzuordnen, dass die eher leichten am Anfang stehen und neue Konzepte dort zuerst erklärt werden. Diese werden dann später als bekannt vorausgesetzt und nicht mehr im Detail erläutert. Ansonsten ist es empfehlenswert, unbekannte Befehle, Konzepte, oder Operatoren in der Suchmaschine Ihrer Wahl nachzuschlagen, was auch erfahrene Programmierer im Schnitt alle fünf Minuten machen...

Online Ressourcen, Code und Lösungen

Online finden Sie den gesamten in diesem Buch gezeigten Code, sowie Bugfixes, Updates und die Lösungen aller Aufgaben kostenlos zum Download.

github.com/fbittmann/Python

Wenn Sie mich kontaktieren möchten, freue ich mich über Emails an *kontakt@felix-bittmann.de*. Sofern Sie Änderungen an den hier

gezeigten Codebeispielen vorschlagen möchten, fügen Sie diese bitte direkt online in Github ein, damit sie öffentlich für alle sichtbar sind. Dort können Sie bequem das gesamte Repository herunterladen, Kopien erstellen und gegebenenfalls abändern.

Danksagung

Ein herzliches Dankeschön gilt an dieser Stelle allen Personen, die mich und das Projekt unterstützt haben. Besonders zu nennen sind (in alphabetischer Reihenfolge) Florian Scholze, Jannik Köster und Kurt Bittmann. Simon Wolf hat alle Codebeispiele akribisch durchgesehen und unzählige Verbesserungen, auch zu verschiedenen Algorithmen, angemerkt. Ebenso gilt mein Dank allen Mitarbeitern der *Python Software Foundation*, die der Welt bewiesen haben, dass Programmierung Spaß machen kann. Abschließend möchte ich mich auch bei allen Personen bedanken, die Abbildungen und Grafiken unentgeltlich bei Projekten wie *Wikipedia* oder *Wikimedia Commons* bereitstellen und somit dazu beitragen, die hier gezeigten Beispiele zu illustrieren.

Kapitel 1

Grundlagen

1.1 Terminologie

Funktionsbefehle werden in Python in der Regel durch kurze, englischsprachige Begriffe umgesetzt, wie beispielsweise *True*, *False*, *for*, *while*, usw... Diese werden auch in anderen Sprachen nicht abgeändert. Deshalb werden in diesem Buch viele englischsprachige Begriffe nicht übersetzt, sondern der prägnante englische Begriff beibehalten, etwa für *dictionary* oder *list comprehension*. Achten Sie zudem bei diesen Schlüsselbegriffen auf die korrekte Groß- bzw. Kleinschreibung.

1.2 Installation und Programmierumgebung

Die aktuellste Version von Python lässt sich schnell und unkompliziert von *python.org* herunterladen. Die in diesem Buch gezeigten Beispiele benötigen mindestens **Python 3.6** oder höher. Wenn Sie Linux oder Mac benutzen, stehen die Chancen gut, dass Python bereits vorinstalliert ist. Um zu testen, welche Version Sie aktu-

ell benutzen, öffnen Sie ein Terminal (Linux, Mac) bzw. die Shell (Windows). Dort tippen Sie *python3*, um eine interaktive Session zu starten und bekommen die aktuell genutzte Version angezeigt.[1]

Als Programmierumgebung (Editor bzw. IDE) ist *Geany* sehr empfehlenswert.[2] Diese Open-Source Anwendung ist schnell, übersichtlich und bietet Anfängern und fortgeschrittenen Nutzern eine große Palette an nützlichen Funktionen, ohne überfrachtet zu sein (nur ca. 16 MB im Download). Zudem stehen eine Vielzahl an Themes und Plugins bereit, sodass neue Funktionen bzw. ein anderer Look im Handumdrehen eingefügt werden können. *Geany* ist für Linux, Windows und Mac verfügbar.

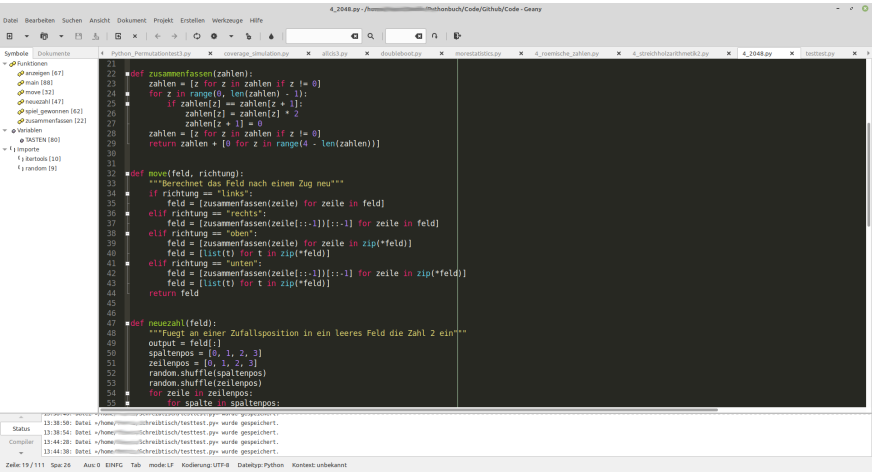

Abbildung 1.1: *Geany* als moderne und offene Programmierumgebung

[1] Falls diese Information nicht direkt beim Aufrufen der Konsole erscheint, tippen Sie: *import sys* und danach *sys.version*.

[2] `geany.org`

1.3 Python Basics

Auf den nächsten Seiten werden in sehr kompakter Form einige Grundlagen wiederholt, die von etwas erfahreneren Nutzern problemlos übersprungen werden können.[3] Im Gegensatz zu den meisten anderen Darstellungen im Buch wird hierbei eine interaktive Session in einer Konsole bzw. dem Terminal dargestellt. Eine mit >>> beginnende Zeile bedeutet daher stets die interaktive Eingabe in die Konsole. Der Output, sofern welcher generiert wird, wird dann in der folgenden Zeile ausgegeben.[4]

```
>>> a = 12
>>> b = 3.141
>>> c = "Banane"
>>> d = [a, b, c]
>>> e = (1.734, 3.822)
>>> f = {3, 8, 99, -4}
>>> g = {"Hallo": 5, "Nein": 4, "Ich": 3, "Rakete": 6}
```

Hier ist *a* eine Ganzzahl (*integer*), *b* eine Kommazahl (*float*), *c* ein String, *d* eine Liste, *e* ein Tuple, *f* ein Set und *g* ein *Dict* (Wörterbuch). Eine Variable wird wie dargestellt über das Gleichheitszeichen definiert. Besondere Vorsicht ist beim Dezimaltrennzeichen gefragt. Python benutzt die angloamerikanische Konvention, weshalb Kommazahlen stets mit einem *Punkt* eingegeben werden müssen, siehe Beispiel *b*. Der Einheitlichkeit wegen wird deshalb im gesamten Buch dieser Konvention gefolgt. Das Komma wird somit als Tausendertrennzeichen verwendet.

[3] Als erste Anlaufstelle für Fragen ist die ausgezeichnete Python Dokumentation sehr empfehlenswert: `docs.python.org/3.6/`

[4] Interaktive Sessions etwa in einer Shell oder dem Terminal eignen sich hervorragend als Spielwiesen und können ohne Weiteres neben dem eigentlichen Editor laufen. So lässt sich schnell testen, ob man noch die richtige Syntax im Kopf hat. Eine Eingabe von *help(Objekt)* liefert eine ausführliche Information zu der gewünschten Funktion, also etwa *help(list)*. Wenn man die Information im Hilfetext gefunden hat, bringt der Tastendruck auf *Q* zurück zur Session.

Bei mathematischen Operationen gilt stets die Abarbeitung nach der Reihenfolge *BEDMAS*: Brackets (Klammern), Exponenten, Division, Multiplikation, Addition, Subtraktion.

Indices und Slices

Listen sind in Python Allzweckwaffen, die fast immer genutzt werden können. Sets, Tuples und Dicts sind oftmals sinnvolle Ergänzungen und können manche Aspekte besser bzw. schneller erledigen, aber ohne Listen ist Python nicht denkbar. In Listen lassen sich beliebige Elemente, also alle Dateitypen und natürlich auch verschachtelte Unterlisten sammeln. Elemente dieser Listen werden über ihren Index aufgerufen. In Python erhält das erste Element einer Liste immer den Index 0.

```
>>> a = [1, 2, 3]
>>> b = ["Hi", 1, "Rot", -6.87, [1, 2, 3, ["Maus"]], 95]
>>> a[0]
1
>>> b[2]
"Rot"
>>> b[4][1]
2
>>> b[-1]
95
>>> len(b)
6
>>> len(b[4])
4
```

Wie gezeigt, können Unterlisten über mehrfache Indices aufgerufen werden. Will man etwa die vorhandene Zahl 2 aufrufen, wählt man zuerst die Unterliste aus, welche den Index 4 hat. Die 2 steht dann an der zweiten Position in dieser Unterliste, hat also den Index 1. Somit erfolgt der Aufruf über b[4][1]. Beim Aufrufen, gleichgültig ob aus einer Liste, einem Tuple, oder einen *Dict*, werden stets eckige Klammern verwendet. Will man ein Element der

Liste von ihrem Ende her aufrufen, nutzt man negative Indices. Das letzte Element einer Liste hat dabei immer den Index -1. Die Anzahl der Elemente in einer Liste wird mittels *len()* abgefragt. Will man eine Liste zerschneiden, also nur Teilstücke verwenden, spricht man von *Slices*.

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> a[0:3]
[1, 2, 3]
>>> a[2:5]
[3, 4, 5]
>>> a[::2]      #Anfang bis Ende, aber nur jede 2. Zahl
[1, 3, 5, 7]
>>> a[::-1]     #Eine Liste umdrehen
[7, 6, 5, 4, 3, 2, 1]
```

Ein Slice-Befehl hat drei Elemente: den Startindex, den Endindex und den *step*, also die Schrittweite. Der Startindex zählt immer zur neuen Liste, der Endindex wird immer exklusiv betrachtet. Wird kein *step* angegeben, so wird immer 1 angenommen. Wird das Start- oder Endelement weggelassen, so setzt Python hier automatisch das erste bzw. letzte Element ein. Die gleichen Regeln gelten für Strings:

```
>>> w = "Trebuchet"
>>> w[3]
"b"
>>> w[2::2]
"euht"
```

Dictionaries

Dicts sind dann sinnvoll anzuwenden, wenn man eine Struktur wie in einem Wörterbuch oder einem Lexikon nachahmen möchte. Dabei werden Paare aus *keys* (Schlüsseln) und *values* (Werten) erstellt, die sich nicht über einen Index, sondern über die direkte Bezeichnung aufrufen lassen. Auf diese Weise kann man eine sehr

einfache Datenbank erstellen, wie das folgende Beispiel aufzeigt, in dem Autoren ihr Geburtsjahr zugeordnet wird:

```
>>> lebensdaten = {"Dawkins": 1941, "Dostojewski": 1821,
    "Goethe": 1749}
>>> lebensdaten["Goethe"]
1749
>>> lebensdaten["Boyle"] = 1948
>>> lebensdaten
{"Dawkins": 1941, "Dostojewski": 1821, "Goethe": 1749, "
   Boyle": 1948}
```

Der erste Wert (vor dem Doppelpunkt) stellt dabei den *key*, die Information dahinter den *value* dar. Der jeweilige *value* kann über die Eingabe des *keys* abgerufen werden, auch Änderungen bzw. das Einfügen neuer *items* wird auf diese Weise möglich. Anzumerken ist, dass die *keys* unveränderlich (immutable) sein müssen, weshalb sich hier etwa Zahlen (integers, floats), Strings oder Tuples anbieten, nicht jedoch Listen, da diese veränderlich sind. Als *values* sind alle Datentypen möglich. *Dicts* sind, sofern es die Anwendung erlaubt, insofern Listen vorzuziehen, als ein Lookup schneller erfolgt. Eine häufige Aufgabe besteht darin, *Dicts* nach bestimmten Elementen zu durchsuchen. Je nachdem, ob die gewünschte Information in den *keys* oder *values* steckt, bieten sich verschiedene Möglichkeiten an.

```
>>> for key in lebensdaten.keys():
>>>     print(key)
Dawkins
Goethe
Dostojewski
Boyle

>>> for value in lebensdaten.values():
>>>     print(value)
1941
1821
1749
1948
```

```
>>> for key, value in lebensdaten.items():
>>>     print(key, value)
("Dawkins", 1941)
("Dostojewski", 1821)
("Goethe", 1749)
("Boyle", 1948)
```

Besonders die letzte Technik ist oftmals sehr sinnvoll, da man gleichzeitig *keys* und *values* erhält und benutzen kann. Die Reihenfolge, in der die Elemente erscheinen, ist dabei jedoch nicht vorhersagbar, da intern kein Index geführt wird.[5] Natürlich können die Elemente dennoch beim Aufruf sortiert werden, etwa alphabetisch. Möchte man hingegen grundsätzlich eine Ordnung haben, so kann man auf *OrderedDict* aus dem Modul *collections* ausweichen.[6]

Schleifen

In Python werden verschiedene Arten von Schleifen unterschieden. Mit *for* kann man direkt über Elemente eines *iterable / iterator* iterieren, also beispielsweise über einen Bereich (range), eine Liste oder ein Tuple.[7] *While*-Schleifen sind dann nützlich, wenn man vorher nicht genau weiß, wie oft die Schleife durchlaufen werden soll und man diese dynamisch beenden will, wenn eine bestimmte Bedingung eintritt.

```
>>> for i in range(0, 10, 2):
>>>     print(i)
```

[5] Ab Python 3.7 gilt dies nicht mehr, ab dieser Version ist auch bei regulären *Dicts* die Reihenfolge fix.

[6] `https://docs.python.org/3.6/library/collections.html`

[7] In Python ist ein *iterable* ein Objekt, über das man iterieren kann, etwa eine Liste, ein Tuple und dergleichen. Ein *iterator* hingegen ist ein Generator und speichert seinen eigenen internen Zustand ab, was beim nächsten Aufruf berücksichtigt wird. Somit kann nur ein iterator mit *next()* angesprochen werden. Bei der Berechnung von Primzahlen in Kapitel zwei werden wir sehen, wie genau Generatoren funktionieren.

```
0
2
4
6
8

>>> wortliste = ["Hallo", "da", "oben"]
>>> for wort in wortliste:
>>>     print(wort)
'Hallo'
'da'
'oben'

>>> wert = 0
>>> while wert < 64:
>>>     print(wert)
>>>     wert = 2 ** wert
0
1
2
4
16
```

Die erste Schleife gibt die *geraden* Zahlen von 0 bis exklusive 10 aus. Wie bei den Slices ist der erste Wert der Startwert, der zweite Wert der Endwert (wieder exklusive, wird also nicht mehr ausgegeben) und der dritte Wert der *step*. Die Variable *i* ist dabei der Laufindex und kann beliebig benannt werden. Die zweite Schleife iteriert über alle Elemente, die in der Liste gefunden werden. Die dritte Schleife läuft, so lange die Bedingung erfüllt ist. In diesem Beispiel muss *wert* kleiner als 64 sein, damit die Schleife durchlaufen wird. Sie wird nicht mehr gestartet, sobald diese Bedingung *False* wird. Manchmal kann es auch vorkommen, dass eine Bedingung immer *True* bleibt, Schleifen dann ewig laufen und nur mittels Abbruch durch den Nutzer beendet werden können. Deshalb muss in diesem Fall der *wert* bei jedem Durchlauf auch verändert werden. Erreicht die Schleife ihr Ende, so beginnt sie

wieder von vorne, also von oben nach unten.

Will man eine Schleife vorzeitig verlassen, so nutzt man *break*. Die Anweisung *continue* ist dann hilfreich, wenn man ein bestimmtes Element in einer Schleife überspringen möchte, etwa, um Zeit zu sparen oder offensichtliche Fehler zu vermeiden. Sie sorgt dafür, dass man sofort zum Beginn der aktuellen Schleife zurückkehrt und dort die nächste Schleifenausführung beginnt. Mit *pass* hat man zudem einen Platzhalter, der gar nichts tut und dann eingesetzt werden kann, wenn man noch leere Codeblöcke füllen möchte, um Fehlermeldungen zu vermeiden. Sehen wir uns drei Beispiele an:

```
>>> for zahl in range(1, 5):
>>>     print(zahl)
>>>     if zahl == 3:
>>>         break
>>>     print(zahl * 10)
>>> print("Schleife wurde verlassen")
1
10
2
20
3
Schleife wurde verlassen
```

Sobald *break* erreicht wird, wird die Schleife sofort verlassen. Alle nachfolgenden Befehle innerhalb der Schleife werden nicht mehr ausgeführt.

```
>>> for zahl in range(1, 5):
>>>     print(zahl)
>>>     if zahl == 3:
>>>         continue
>>>     print(zahl * 10)
>>> print("Schleife wurde verlassen")
1
10
2
20
3
```

```
4
40
Schleife wurde verlassen
```

Wird *continue* angetroffen, so wird direkt zum Anfang der Schleife gesprungen und der nächste Durchlauf gestartet. Alle nachfolgenden Befehle werden nicht mehr ausgeführt, die Schleife wird allerdings auch nicht beendet.

```
>>> for zahl in range(1, 5):
>>>     print(zahl)
>>>     if zahl == 3:
>>>         pass
>>>     print(zahl * 10)
>>> print("Schleife wurde verlassen")
1
10
2
20
3
30
4
40
Schleife wurde verlassen
```

Wird *pass* erreicht, so geschieht nichts. Die Schleife wird nicht verlassen, sondern alle Befehle unter *pass* innerhalb der Schleife werden normal ausgeführt. Dieser Befehl eignet sich daher vor allem als Platzhalter.

Comprehensions

Comprehensions in verschiedenen Variationen sind äußerst nützlich, da sie es erlauben, sich das Schreiben von expliziten Schleifen zu ersparen. Man unterscheidet dabei zwischen list, dict, set und generator comprehensions, wobei die Syntax fast identisch ist. Angenommen man möchte eine Liste aller Zahlen unter 100, die sowohl durch 3 als auch durch 7 teilbar sind. Mit einer comprehension

geht das in einer Zeile:

```
>>> [i for i in range(100) if i % 3 == 0 and i % 7 == 0]
[0, 21, 42, 63, 84]
>>> [i ** 2 for i in (1, 2, 3, 4, 5)]
[1, 4, 9, 16, 25]
```

Die eckigen Klammern geben dabei an, dass wir eine Liste generieren möchten und i ist der Laufindex, der alle Zahlen zwischen 0 und 99 durchläuft. Allerdings haben wir einen Filter eingebaut, der nur Zahlen durchlässt, die die gewünschte Bedingung erfüllen. Das zweite Beispiel zeigt auf, wie wir Zahlen vor dem Eintragen in die Liste noch transformieren können. Auch *if...else* Konstruktionen sind möglich, allerdings ist hier die Syntax etwas anders.

```
>>> [1 if x > 5 else 0 for x in range(10)]
[0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
```

Hierbei erhalten wir auf jeder Position einer Zahl, die größer als 5 ist, eine 1, ansonsten eine 0. If...else steht nun vor dem iterable, da es sich hierbei nicht um einen Filter handelt, sondern um den Tenary-Operator. Sets und Dicts können wir mit der gleichen Syntax erstellen, die Definition erfolgt dann schlichtweg über die Art der Klammern.

```
>>> {i for i in range(10)}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> {wort: len(wort) for wort in ["Was", "machst", "du"
    ]}
{'Was': 3, 'machst': 6, 'du': 2}
```

Comprehensions können durch die gezeigten Möglichkeiten komplex werden, auch deshalb, weil eine Comprehension weitere Comprehensions enthalten kann. Wir benötigen eine Matrix, also eine Liste mit Unterlisten? Das kann so erreicht werden:

```
>>> [[i * j for i in range(4)] for j in range(4)]
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6], [0, 3, 6, 9]]
```

Python arbeitet hier von innen nach außen, erzeugt also erst eine Liste, die vier Produkte aus i und j sammelt. Auf diese Weise

werden vier Listen erstellt, welche dann in einer übergeordneten Liste gesammelt werden. Hierbei sollte man aber vorsichtig sein und einen Mittelweg zwischen Kompaktheit und Lesbarkeit suchen. Eine komplexe Comprehension, die heute unglaublich cool oder elegant wirkt, kann morgen dem Kollegen Kopfschmerzen bereiten (oder auch einem selbst, nachdem man zwei Wochen in Urlaub gewesen ist...). Dieser Vorbehalt trifft etwa auch dann zu, wenn mehrere for-Schleifen in einer Comprehension benutzt werden.

Funktionen

Wenn man eine komplexere Aufgabe in Python lösen möchte, so bietet es sich in der Regel an, verschiedene Funktionen zu erstellen, die unterschiedliche Teilaufgaben lösen. Dies hat zahlreiche Vorteile: beispielsweise können die Teilfunktionen wiederum an anderer Stelle nützlich sein und sich somit leicht in andere Skripte importieren lassen. Zudem können Bugs oder Probleme oftmals schneller gefunden werden, wenn man Funktionen separat testen kann. Kurzum, teile und herrsche!

In Python kann man Funktionen prinzipiell auf zwei Weisen definieren. Die eine Art erfolgt über den Befehl *def()*. Eine Funktion kann dann eine beliebige Anzahl von Argumenten beinhalten, die bei Bedarf auch als Voreinstellungen (defaults) vorgegeben werden können.[8] Probieren wir dies an einer schlichten Addition:

```
>>> def addierer(x, y):
>>>     return x + y
>>> addierer(1, 1)
2
```

Die soeben definierte Funktion *addierer()* hat zwei Argumente, x und y. Diese müssen jeweils vom Nutzer vorgegeben werden.

[8] Auch wenn es nicht ganz korrekt ist, werden *Argument* und *Parameter* in Bezug auf Funktionen hier synonym gebraucht.

Über den Begriff *return* definieren wir, was am Ende zurückgegeben werden soll. Setzen wir kein return statement (oder wird dieses niemals erreicht, etwa bei einer *if...else* Bedingung), so gibt jede Funktion am Ende *None* aus. Dies kann in vielen Fällen ohne Bedeutung sein, etwa, wenn eine Funktion nur etwas in der Konsole anzeigen soll.

```
>>> def gruss(name):
>>>     print("Hallo " + str(name) + "!")
>>> gruss("Python")
"Hallo Python!"
```

Wenn wir *defaults* vorgeben, so können diese bei Bedarf vom Nutzer überschrieben werden, die Variable hat aber ansonsten diesen Standardwert:

```
>>> def potenzieren(x, y = 2):
>>>     return x ** y
>>> potenzieren(3)
9
>>> potenzieren(2, 4)
16
```

Neben *def()* gibt es noch die Möglichkeit, anonyme Funktionen über das Keyword *lambda* zu definieren. Diese Funktionen haben den Vorteil, dass sie sehr kompakt sind und an manchen Stellen „on the fly“ definiert werden können, etwa, wenn eine Liste nach einem bestimmten Schema sortiert werden soll.

```
>>> addierer = lambda x, y: x + y
>>> addierer(2, 2)
4
```

Dabei muss man sich jedoch auf einen Ausdruck beschränken, weshalb komplexe Aufgaben sich nicht für derartige Funktionsdefinitionen eignen.

An dieser Stelle sei noch auf eine Kurzschreibweise hingewiesen, die dann verwendet werden kann, wenn der Wert einer bestehenden Variable verändert werden soll.

```
x = x + 5 <=> x += 5
x = x - 5 <=> x -= 5
x = x * 5 <=> x *= 5
x = x / 5 <=> x /= 5
```

Auf diese Weise lassen sich die verschiedenen Basisoperationen kompakter schreiben.

Interne Checks und Fehlerbehandlung

Wer Programme für Endbenutzer schreiben möchte, wird sehr viel Zeit aufwenden um, sicherzugehen, dass bestimmte Funktionen nur mit den korrekten Eingaben gefüttert werden können. So sollte beispielsweise ein Name keine Ziffern enthalten, eine Email hingegen immer den Klammeraffen @. Auch rein mathematische Funktionen müssen auf diese Weise geschützt werden, sofern sie nur in einem bestimmten Definitionsbereich funktionieren (bitte nicht durch 0 teilen!). Manchmal gibt es allerdings keine expliziten Fehlermeldungen, sondern es werden schlichtweg falsche Ergebnisse produziert. Eine derartige Suche nach Bugs kann dann aufwändig und schwierig sein, weshalb es sich empfiehlt, immer wieder Kontrollpunkte einzubauen, die sicherstellen, dass auch nur „legale“ Werte prozessiert werden. Eine einfache Funktion hierfür ist *assert*. Diese prüft, ob eine bestimmte Bedingung erfüllt ist. Sehen wir uns an, wie wir direkt testen können, ob eine Email-Adresse eine falsche Syntax aufweist:

```
>>> email1 = "maxmuster@mustermail.de"
>>> assert "@" in email1, "Ungueltige Eingabe!"
>>> email2 = "email.email.org"
>>> assert "@" in email2, "Ungueltige Eingabe!"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AssertionError: Ungueltige Eingabe!
```

Während die erste Eingabe korrekt ist und *assert* in diesem Fall schlichtweg gar nichts tut, also auch keine Anzeige auslöst,

beendet Python beim zweiten Beispiel das Programm sofort. Wir sehen, welche Art von Fehler vorliegt, zudem können wir spezifizieren, welche Meldung angezeigt werden soll. Wenn bereits zu Beginn klar ist, welche Eingaben eine Funktion handhaben soll bzw. welche Zahlenwerte oder Bereiche gültig sind, ist es oftmals besser, direkt zu Beginn einen offensichtlichen Fehler zu melden, als zu warten, bis sich dieser irgendwann am Ende in einem seltsamen Ergebnis zeigt. Der Vollständigkeit halber sei darauf hingewiesen, dass assert-Befehle bei verschiedenen Optimierungsverfahren automatisch entfernt werden und daher keine finale Lösung sind, um Nutzereingaben zu kontrollieren. In realen Anwendungen sollten daher ausgefeiltere Tests zum Einsatz kommen.

Wie wir gesehen haben, beendet Python bei einem Fehler die Ausführung des Skripts sofort. In manchen Fällen wollen wir aber auf erwartbare Fehler reagieren können und das Skript fortsetzen. Dies können wir über *try...except* Konstruktionen erreichen. Wir lassen einen Befehl ausführen. Sollte es zu einem Fehler kommen, den wir so erwartet haben, fährt das Programm mit einem bestimmten Verhalten fort und stoppt nicht. Ein Beispiel ist, wenn man einen Index in einer Liste ansprechen will, diese allerdings zu kurz ist und den Index gar nicht mehr enthält.

```
>>> a = [1, 2, 3]
>>> a[20]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Offenbar hat die Liste nur drei Elemente, weshalb der größte Index 2 ist (Pythons erster Index ist immer 0) und 20 nicht vorkommt. Manchmal können wir dies aber hinnehmen und fahren dann einfach mit der nächsten Liste fort.

```
>>> for liste in matrix:
>>>     try:
>>>         print(liste[20])
>>>     except IndexError:
```

```
>>>         print("Index nicht vorhanden, fahre fort")
```

Wir wollen von jeder Liste in der Datenmatrix das Element mit Index 20 anzeigen lassen. Wir verpacken den Befehl in einen try-Block. Sollte es in diesem Bereich zu einem Fehler kommen (und wir erwarten hier im Moment nur einen IndexError), so macht Python mit den Befehlen unter *except* weiter, also in diesem Fall wird eine Fehlermeldung ausgegeben und die Schleife danach fortgesetzt. Taucht in *try* allerdings ein Fehler auf, den wir nicht explizit bei der Definition von *except* berücksichtigt haben, wird das Skript dennoch stoppen. Sogenannte *catch-all* Statements, die alle Arten von Fehlern ignorieren, sind mit Vorsicht zu behandeln und sollten daher nur bei bestimmten Fällen in Betracht gezogen werden. Dann würde man die Zeile in *except:* abändern.

Module

Manche Funktionen oder Datentypen sind in Python immer direkt verfügbar, wie beispielsweise *len()*, *max()* oder *set()*, um nur einige wenige zu nennen. Andere Funktionen sind zwar ebenfalls offizieller Teil Pythons, allerdings funktional gruppiert und in Module ausgelagert. Dies ist insofern effizient, als nicht immer alle Funktionen direkt verfügbar sein müssen. Zudem verhindert es Probleme mit mehrfach belegten Namen von Variablen oder Funktionen. Möchten wir Elemente aus anderen Modulen benutzen, so müssen wir sie vorher importieren. Die Handhabung ist einfach. Angenommen, man braucht mehr mathematische Funktionen, wie etwa den Cosinus:

```
>>> import math
>>> math.cos(math.pi)
-1.0
```

Zunächst importieren wir das Modul *math*. Man kann nun alle enthaltenen Funktionen nutzen, muss dabei allerdings immer die Bezeichnung des Moduls beim Aufruf mit angeben, damit Python

weiß, aus welchem Modul die Funktion bzw. die Konstante (hier Pi) entnommen werden soll. Dies kann auf Dauer lästig sein, besonders, wenn Modulnamen länger sind. Hierbei gibt es zwei Möglichkeiten. Zunächst kann man den zu verwendenden Modulnamen abkürzen:

```
>>> import itertools as it
>>> list(it.combinations([1, 2], 2))
[(1, 2), (1, 3), (2, 3)]
```

Sofern man nur eine ganz bestimmte Funktion aus einem Modul benötigt, ist folgender Ansatz empfehlenswert:

```
>>> from itertools import combinations
>>> list(combinations([1, 2], 2))
[(1, 2), (1, 3), (2, 3)]
```

Ansonsten kann man auch *alle* Funktionen aus dem Modul übernehmen. Dies ist insofern problematisch, da man in diesem Fall aufpassen muss, dass man nicht versehentlich bereits andere Funktionen geschrieben hat, die die gleiche Bezeichnung aufweisen.

```
>>> from itertools import *
>>> list(combinations([1, 2], 2))
[(1, 2), (1, 3), (2, 3)]
```

Besonders bei längeren Skripten mit vielen importierten Modulen erscheint es ratsam, den Modulzusatz auf jeden Fall beizubehalten (und ggf. sinnvoll und eindeutig abzukürzen), was die Verständlichkeit besonders für nur am Rande involvierte Personen massiv erhöht.

1.4 Grundsätze guter Programmierung

1. In Python nehmen Einrückungen eine herausragende Bedeutung ein und ersetzen in vielen Fällen die aus anderen Sprachen berüchtigten Klammern. Ob man nun zum Einrücken Leerzeichen oder Tabulatoren nutzt, ist unerheblich, solange

man hierbei absolut konsequent ist und diese niemals mischt (was auch mit Fehlern quittiert wird).

2. Alle Variablen bzw. Objekte (und in Python ist mehr oder weniger *alles* ein Objekt) sollten eindeutig und sinnvoll benannt sein. Dabei kann man, je nach eigenen Vorlieben, einen bestimmten Stil wählen, etwa *Panellinks* (Pascal Case), *panelLinks* (Camel Case), oder *panel_links* (Snake Case). Wichtiger ist dann Konsistenz. Für Laufindices oder temporäre Variablen scheint es aber nicht unbedingt erforderlich sehr viel Zeit auf Namensgebung zu verwenden. Einbuchstabige Variablennamen sollten jedoch nur in sehr eng begrenzten Codeblöcken oder in Comprehensions genutzt werden.

3. Funktionen sollten meistens *genau eine* Aufgabe erfüllen. Erkennt man, dass eine Funktion sehr viele Grenzfälle berücksichtigen muss bzw. durch viele *if...else* Anweisungen teilweise sehr verschiedene Aufgaben übernimmt, könnte es sinnvoll sein, die Funktion in mehrere Teilfunktionen aufzuspalten. Zudem sollte man eine Aufgabe nie an zwei verschiedenen Stellen separat lösen, also Code duplizieren, sondern *eine* Funktion definieren und bei Bedarf aufrufen. Dies erleichtert die Wartung ungemein. Ansonsten müsste man im schlimmsten Fall Bugs am Ende an zig Stellen beseitigen, was unbedingt zu vermeiden ist. Zudem besteht die Grundregel, dass eine Funktion nur *einen* bestimmten Datentyp ausgeben sollte. Eine Funktion, die demnach Zahlen prozessiert, sollte auch nur Zahlen ausgeben. Tritt allerdings ein Fehler auf, so sollte nicht *False* erzeugt werden (was ein Bool-Wert und keine Zahl ist), sondern eine Exception ausgelöst werden, also eine explizite Fehlermeldung.

4. Python ist eine anwendungsorientierte Sprache, die dazu er-

funden wurde, um Aufgaben zügig und effizient zu erledigen.[9] Somit sollte man sich vor jedem Projekt fragen, welchen Aufwand man in die Rahmenparameter stecken möchte. Müssen zunächst fünf Klassen definiert werden oder reichen am Ende auch zwei Funktionen? Natürlich hängt dies vor allem von der Komplexität und den Projektumständen ab. Größere und auf längere Zeit angelegte Projekte verdienen am Anfang sicher mehr Aufmerksamkeit, damit man später offen und flexibel ist. Besonders wenn man mit anderen Personen zusammenarbeitet, sollte man zunächst einen gemeinsamen Stil festlegen und sich, wie bereits erwähnt, auf Richtlinien und stilistische Vorgaben einigen.

5. Die Lesbarkeit des Codes lebt auch von den Abständen zwischen den einzelnen Zeichen bzw. Bestandteilen. So kann man etwa *x=(5+5)*2* schreiben, oder aber auch *x = (5 + 5) * 2*. Wieder gibt es hierbei keine festen Regeln und man kann seinem Instinkt folgen, solange man diesen dann auch konsequent beibehält. Der Rest des Buchs hält sich dabei an einen Stil, der Leerzeichen zwischen viele Zeichen einfügt, aber auch hier gibt es Ausnahmen (etwa bei Klammern oder beim Potenzieren, um nur einige zu nennen).

6. Neben der sinnvollen Benennung von Objekten gehört eine Dokumentation, besonders bei komplexeren Projekten, zu den absoluten Basics, die man nicht ignorieren sollte. Oft ist man selbst die Person, die den Code später wieder verstehen muss. Eine saubere Dokumentation, die gerne auch knapp und prägnant sein darf, ist daher ein Geschenk an das zukünftige Ich. Dazu eignen sich etwa *Docstrings* (“““Hier der Text“““), die in Python eine besondere Rolle einnehmen und etwa eine Beschreibung von Funktionen erlauben (und so von

[9] *import this*, siehe auch Seite 239.

Python auch ausgelesen werden können). In den Beispielen dieses Buches sind Funktionen oftmals wenig dokumentiert, was daran liegt, dass sie im Fließtext ausführlich erklärt werden und die Codebeispiele nicht aufgeblasen werden sollen. Für kürzere Kommentare in einer Zeile wird das Rautenzeichen verwendet (#).

7. Wenn Sie bisher wenig Erfahrung mit Versionsverwaltung oder Archivierung haben, lohnt es sich bei mittelgroßen bis größeren Projekten, etwas Zeit in diese Aspekte zu investieren. Somit erspart man sich das Anlegen von zahlreichen Dokumenten (testfunktion1, testfunktion2, testfunktion_final, testfunktion_final2,...) und kann diese auch zur Datensicherung (Backups) oder für die Zusammenarbeit mit anderen Personen nutzen. Als Basistools dienen etwa die Programme *git* oder *bazaar*. Möchte man dann auch online arbeiten, kann man diese dann beispielsweise mit *Github* kombinieren.

8. *Debuggen*, also das Aufspüren von Fehlern im Programmcode, nimmt meistens einen erheblichen Raum im Prozess des Programmierens ein. Ein unschätzbarer Vorteil von Python ist, dass Fehlermeldungen sehr detailliert und hilfreich sind und daher zumindest sehr präzise Anhaltspunkte geben, was schiefgelaufen ist. Dies betrifft oftmals banale Fehler, wie das Vergessen einer Klammer oder eines Doppelpunkts. Bei unbekannten Meldungen lohnt es sich, die Fehlermeldung online zu recherchieren und die letzten Zeilen vor und nach der von Python gemeldeten Stelle genau zu prüfen.

9. Keine Regel ohne Ausnahmen. Die hier vorgestellten Leitsätze sollen genau das sein: Anleitungen bzw. Hilfen, keine starren Gesetze. Es kann gute Gründe geben, von diesen abzuweichen. Wenn es diese allerdings nicht gibt stellt sich die Frage, ob man gerade nur etwas faul oder müde ist, was dann

wiederum ein guter Grund für eine Pause ist.

10. Wer generell Empfehlungen sucht, was Stil oder Konventionen angeht, kann sich an den offiziellen Style Guide *PEP8* wenden, der regelmäßig aktualisiert wird.[10]

1.5 Angewandte Problemlösung

Das Vorgehen in diesem Buch ist problemorientiert, es geht also darum, eine bestimmte Aufgabe zu lösen und ein Ergebnis zu finden. Es ist daher weniger ein reines Lehrbuch, sondern hat den Anspruch, auch im Alltag die Problemlösefähigkeiten zu trainieren, da es sich um realistische Szenarien handelt. Der Chef legt Ihnen eine Aufgabe vor und kümmert sich danach nicht mehr darum, wie sie diese lösen, sofern sie am Ende korrekt erledigt wurde. Es ist dann an Ihnen herauszufinden wie Sie das Problem meistern können. Insofern eignet sich Python für viele Aufgaben hervorragend, da es umfassende Werkzeuge und Anwendungen mitbringt, sodass für viele Szenarien bereits vorgefertigte Funktionen bzw. Module vorhanden sind, die sich mit wenig Aufwand auf die konkreten Aufgaben anpassen lassen. Zudem ist die Rechenleistung moderner Computer oftmals so groß, dass auch recht komplexe Probleme ohne die Kenntnis einer analytischen Methode zu bearbeiten sind, etwa, indem stur alle nur denkbaren Lösungsansätze probiert werden (Brute-Force) oder aber Simulationen herangezogen werden können, um Näherungslösungen zu finden, die in Anwendungsszenarien oftmals völlig ausreichend sind. Wie genau kann nun ein idealisierter Ansatz aussehen, um eine konkrete Aufgabe zu bewältigen?

Zunächst ist es wichtig, das vorliegende Problem zu verstehen und sich einen Überblick zu verschaffen. Habe ich ähnliche Aufga-

[10] `pep8.org`

ben bereits in der Vergangenheit gelöst? Gibt es verwandte Ansätze, die ich kenne? Somit sollte man versuchen, Unbekanntes auf Bekanntes zurückzuführen. Dank Suchmaschinen und Wikipedia ist es zudem häufig der Fall, dass die vorliegende Aufgabe bereits von anderen Personen gelöst wurde und der Lösungsansatz bzw. sogar der konkrete Algorithmus, im besten Fall mit der Implementierung in Python, schlichtweg kopiert werden kann. Dies klappt in sehr vielen Fällen und minimiert dadurch den eigenen Aufwand. Insofern sei an dieser Stelle gesagt, dass es natürlich nicht der Sinn dieses Buches ist, die Aufgaben sofort online nachzuschlagen und die Lösung zu kopieren (das würde höchstens Ihre Recherchefähigkeiten trainieren...). Ich empfehle daher, die Aufgaben in aller Ruhe zu bedenken und als Denkübungen bzw. Rätsel zu betrachten. Kommt man nicht sofort auf eine Lösung kann es sinnvoll sein, einfach mit der nächsten Aufgabe weiterzumachen und später auf die ungelöste zurückzukommen. Das menschliche Gehirn arbeitet unentwegt und auch ohne, dass uns dies bewusst ist, an Lösungen weiter, was dann zu den berühmten Gedankenblitzen führen kann. Dennoch ist es natürlich legitim, nach ausreichender Bedenkzeit Lösungsideen und Ansätze nachzuschlagen, damit man diese danach eigenständig in Python umsetzen kann.

Hat man insofern einen Plan vor Augen, wie das Problem gelöst werden kann, ist eine Umsetzung in Python oftmals die leichtere Aufgabe. Wie bereits zuvor beschrieben, sollte man dann die ganze Aufgabe in Teilschritte zerlegen und sich überlegen, was man davon wie abarbeiten kann. Dank Funktionen ist eine sinnvolle und übersichtliche Strukturierung einfach. Dann sollte man diese Funktionen Schritt für Schritt implementieren und sich zunächst weniger um Perfektion sorgen. Oftmals ist es schon ausreichend, ein erstes Ergebnis oder eine Abschätzung zu erhalten (zumindest für den Chef). Sofern man an der Umsetzung der Lösungsidee in Python Probleme hat ist es oftmals ratsam ein Lehrbuch zu konsultieren oder online nachzuschlagen. In Python muss das sprich-

wörtliche Rad fast nie selbst erfunden werden, die umfassenden Bibliotheken beinhalten oftmals bereits die gewünschte Funktion. Dies beschleunigt die Entwicklung und verhindert zudem Fehler, da die offiziellen Funktionen über Jahre hinweg auf Herz und Nieren geprüft wurden und man davon ausgehen kann, dass diese frei von Bugs sind. Für manche Probleme stehen auch externe Erweiterungen bereit, die die Funktionen von Python massiv erweitern. Wenn man komplexere Probleme angeht bzw. dauerhaft auf diesen Gebieten arbeitet kann es sehr sinnvoll sein, sich diese im Detail anzuschauen.

Ist der Code erstellt, so wird es häufig bei den ersten Tests passieren, dass er nicht so funktioniert, wie man sich das vorgestellt hat. Zu Beginn sind oftmals Syntaxfehler häufig, sodass überhaupt kein Ergebnis erscheint, sondern das Skript den Dienst mit einer Fehlermeldung quittiert. Vergessene Zeichen, falsch geschriebene Variablen oder inkorrekt spezifizierte Funktionen sind hierbei häufige Stolpersteine, die sich jedoch meistens rasch beseitigen lassen. Schwieriger sind dann Logikfehler aufzuspüren, die etwa dann auftreten, wenn das Skript syntaktisch korrekt durchläuft, aber keine oder eine offenbar falsche Lösung erscheint. Hierbei kann es dann wiederum an einem banalen Tippfehler liegen, etwa wenn eine falsche Variable in eine Funktion eingeht, aber auch Logikfehler, die eher am erdachten Algorithmus liegen. Bei diesem Prozess des Debuggens sollte man dann die Einzelfunktionen auf Korrektheit testen und sich einfache Beispielfälle suchen und diese Schritt für Schritt durchgehen. Hierbei kann man schlichtweg über das Einsetzen von print-statements nachsehen, welchen Wert Variablen beispielsweise an bestimmten Stellen haben, was unheimlich hilfreich sein kann. Auch kann man ein Skript über interne Pausierungsanweisungen (*time.sleep()*) „in Zeitlupe“ ausführen lassen und so etwa oft durchlaufene Schleifen Schritt für Schritt nachvollziehen. Zwar ist diese Methodik in Fachkreise mitunter verpönt, aber für kleinere Anwendungen wie die hier gezeigten absolut valide. Python

selbst liefert mit *pdb* eine sehr umfassende Möglichkeit des Debuggens, welche extrem hilfreich sein kann.[11] Die Idee hierbei ist, den Code sozusagen Zeile für Zeile ausführen zu lassen und den aktuellen Wert bestimmter Variablen dabei interaktiv nachverfolgen zu können. Komplexere Entwicklungsumgebungen liefern zudem in vielen Fällen zahlreiche weitere Tools zum Debuggen mit, die sich allerdings an fortgeschrittene Nutzer richten. Zum besseren Verständnis des eigenen Codes bzw. der erdachten Algorithmen kann es hilfreich sein, diese Schritt für Schritt einem Kollegen[12] zu erklären, was oft dazu führt, dass man Unklarheiten präzise formulieren muss. Dies kann dem eigenen Verständnis helfen.

Ist der Code auf diese Weise von offensichtlichen Bugs befreit, sodass er die korrekte Lösung liefert, scheint es oftmals sinnvoll, diesen noch weiter zu verbessern. Besonders bei größeren Projekten mit längerer Laufzeit lohnt es sich, diesen zu optimieren (Refactoring). Dabei versucht man die Lesbarkeit, Verständlichkeit und Geschwindigkeit zu verbessern. Diese Aufgabe kann man insofern entspannt angehen, da man bereits eine funktionierende Lösung geliefert hat. Es folgt nun also die „Kür“, um noch mehr herauszuholen. So kann man versuchen, umständliche Codeabschnitte, die mitunter auch als „Spaghetticode“ bezeichnet werden, sauberer aufzuschreiben. Wie genau dies aussieht hängt natürlich von der jeweiligen Struktur bzw. Funktionalität ab. Hier ist dann immer ein Kompromiss zwischen Kompaktheit und Lesbarkeit zu finden. Auch das Einfügen von Kommentaren, Docstrings und anderen Hinweisen, die man in der Eile vergessen hat, erscheint jetzt absolut empfehlenswert. Letztlich kann man auch die Performance optimieren und überlegen, an welchen Stellen der Code unnötig langsam ist und Verbesserungspotential besteht. Zur Messung sollte man dann verschiedene Optimierungsstrategie anwenden, wie wir sie im Verlauf des Buches noch kennenlernen werden.

[11] `https://docs.python.org/3.6/library/pdb.html`

[12] Ansonsten tut es oftmals auch ein Haustier oder eine Gummiente...

Kapitel 2

Zahlenspiele und Numerik

2.1 Einleitung

In der Mathematik wird eine (un)endliche Auflistung von fortlaufend nummerierten Objekten als Folge bezeichnet. Dabei spielen Folgen auch im Alltag eine wichtige Rolle, beispielsweise, wenn man an die Fibonacci-Folge oder die Folge der Primzahlen denkt.

2.2 Fibonacci

Die Fibonacci-Folge, die nicht nur in der Mathematik bereits seit Jahrtausenden bekannt ist, sondern auch in der Natur immer wieder in Populationsgesetzen oder der Anordnung von Objekten vorkommt, ist über ein rekursives Bildungsgesetz definiert. Die ersten beiden Folgenglieder sind jeweils 1 ($f_1 = f_2 = 1$). Die weiteren Folgenglieder ($i \geq 3$) sind definiert über

$$f_i = f_{i-1} + f_{i-2} \tag{2.1}$$

In Worten ausgedrückt: Das nächste Folgenglied ist die Summe der beiden vorhergehenden. Die ersten 10 Elemente der Folge

lauten somit $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$.

Da bisher nur eine rekursive Definition der Folge vorliegt, müssen zur Berechnung des n-ten Gliedes sämtliche vorhergehenden Glieder berechnet werden. Zunächst werden wir verschiedene Lösungsansätze besprechen. Wir arbeiten uns somit vom Startpunkt aus nach vorne, speichern die neuen Glieder ab und verwenden diese wiederum als Ausgangspunkt für die nachfolgenden. Auf diese Weise können wir beliebig viele Elemente der Folge generieren. Ein sehr einfacher Ansatz könnte so aussehen:

```
1 def fibonacci(n):
2     assert n > 0
3     a, b  = 1, 1
4     for i in range(n):
5         print(a)
6         a, b = b, a + b
```

Wir lassen uns durch diese Funktion alle Folgenglieder bis einschließlich n ausgeben. Dabei definieren wir, dass das erste Folgenglied immer den Index 1 hat, weshalb wir mit *assert* die Eingabe daraufhin prüfen. *a* ist das vorletzte, *b* das letzte bekannte Folgenglied. Diese Variablen werden zu Beginn mit 1 initialisiert. Hier nutzen wir eine Abkürzung in Python und sparen uns so eine Zeile (Tuple-Assignment). Am Ende der Schleife benutzen wir einen ähnlichen Trick und vermeiden somit die Nutzung einer Hilfsvariable, um einen Wert zwischenspeichern zu müssen. Falls wir die Folgenglieder allerdings abspeichern wollen, um sie beispielsweise in einer anderen Funktion zu benutzen, so nutzen wir nicht *print()*, sondern legen sie beispielsweise in einer Liste ab.

```
>>> fibonacci(10)
1
1
2
3
5
8
13
```

```
21
34
55
```

Das Ergebnis stimmt. Programmieren wir nun eine Version mittels Listen.

```
1  def fibonacci2(n):
2      glieder = [1, 1]
3      for i in range(n):
4          glieder.append(glieder[-1] + glieder[-2])
5      return glieder[:-2]
```

In diesem Beispiel verzichten wir auf die Prüfung der Eingabe und korrigieren das Offset um die zwei Anfangsglieder am Ende, indem wir die beiden letzten Elemente der Folge nicht mit ausgeben lassen. Wir starten mit einer Liste, die die beiden Startwerte enthält. Mit *append()* fügen wir am Ende der Liste das jeweils neu berechnete Glied an.

Wie anfangs festgestellt, ist die Fibonacci-Folge rekursiv definiert. Warum also nicht auch bei der Programmierung auf Rekursion zurückgreifen? Meistens zeichnen sich rekursiv definierte Funktionen durch eine gewisse Eleganz aus, können jedoch bei komplexeren Aufgaben schwierig zu verstehen sein. Ein weiterer Nachteil ist, dass die Selbstaufrufung der Funktion stets einen gewissen Overhead mit sich bringt, weshalb die Berechnung im Vergleich zu einer der oben gezeigten Lösungen mitunter mehr Ressourcen benötigt. Zudem ist in Python die Rekursionstiefe nicht unendlich, sondern wird durch einen Parameter gesteuert bzw. angepasst. Wird diese Anzahl überschritten, bricht die Berechnung mit einer Fehlermeldung ab. Sollen demnach *sehr* viele Folgenglieder berechnet werden, ist eine Programmierung mittels Rekursion daher nicht optimal. Abgemildert werden kann dieses Problem allerdings durch eine Speicherung der bereits berechneten Glieder. Somit wird nach jeder Berechnung das Folgenglied gespeichert und die Anzahl der tatsächlich durchgeführten Rekursionen reduziert sich erheblich.

```
1  def fibonacci3(n):
2      glieder = {1:1, 2:1}
3      def inner(n):
4          if n not in glieder:
5              folgeglied = inner(n-1) + inner(n-2)
6              glieder[n] = folgeglied
7          return glieder[n]
8      return inner(n)
```

In diesem Beispiel, in dem wieder nur die n-te Fibonacci-Zahl berechnet wird, nutzen wir ein *Dict* zur Abspeicherung der bereits berechneten Glieder. Somit schaut die Funktion erst nach, ob ein Wert schon vorhanden ist, falls nicht, berechnet es diesen rekursiv und fügt ihn anschließend dem *Dict* hinzu. Wichtig ist es, dass hier *inner()* angelegt wird, damit beim Selbstaufruf das *Dict* nicht immer wieder leer erzeugt wird, was bedeutet, dass es nicht genutzt wird wie geplant. Es wird also in der Rekursion selbst dann nur die innere Funktion immer wieder aufgerufen, während die bereits bekannten Werte im *Dict* gespeichert bleiben.

Aufgaben

1. Programmieren Sie eine Funktion, welche die ersten 5,000 Fibonacci-Zahlen in einer Liste speichert und lassen Sie sich diese ausgeben.

2. Tatsächlich kann man die n-te Fibonacci-Zahl auch ohne Rekursion berechnen, indem man die Formel von Moivre-Binet anwendet. [1] Lassen Sie sich die 1000. Fibonacci-Zahl mit dieser Funktion berechnen und vergleichen Sie das Ergebnis mit der anderen Funktion. Was stellen Sie fest? Was ist schiefgelaufen?

[1] `https://de.wikipedia.org/wiki/Fibonacci-Folge#Formel_von_Moivre-Binet`

$$f_i = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - \left(\frac{1-\sqrt{5}}{2} \right)^i \right) \tag{2.2}$$

3. Programmieren Sie die gestellte Aufgabe auf mindestens zwei verschiedene Weisen und vergleichen Sie die Geschwindigkeit. Tipp: Mit der Funktion *time.time()* oder *time.monotonic()* können Sie die Zeitdifferenz zwischen Funktionsaufruf und Abschluss berechnen und somit die Zeit stoppen.

4. Der Quotient zweier aufeinander folgender Fibonacci-Zahlen nähert sich dem Goldenen Schnitt[2] an, wenn n gegen unendlich geht. Berechnen Sie den Quotienten für die Folgeglieder 10^1, 10^2, 10^3, 10^4 und 10^5 und die jeweilige prozentuale Abweichung zum tatsächlichen Wert auf fünf Nachkommastellen genau.

5. Berechnen Sie die Summe der Kehrwerte der ersten 5,000 Fibonacci-Zahlen.

6. Dem Zeckendorf-Theorem zufolge kann jede natürliche Zahl als die Summe voneinander verschiedener, nicht direkt aufeinanderfolgender Fibonacci-Zahlen ausgedrückt werden. Die Zahl 6 ist beispielsweise die Summe der fünften und zweiten Fibonacci-Zahl (5+1). Erstellen Sie eine Funktion, die als Eingabe eine natürliche Zahl akzeptiert und als Ausgabe die Zerlegung in Fibonacci-Zahlen vornimmt. Tipp: eine Beschreibung des Algorithmus finden Sie online.[3]

7. In der letzten hier gezeigten Funktion, *fibonacci3()*, nutzen wir eine innere Funktion, damit die bereits bekannten Werte beim Selbstaufruf nicht wieder überschrieben werden. Wie

[2] $\Phi = \frac{1+\sqrt{5}}{2} \approx 1{,}6180339887$

[3] `https://cp-algorithms.com/algebra/fibonacci-numbers.html`

kann man dies anders lösen, ohne eine innere Funktion zu benutzen? Tipp: es geht auch ohne globale Variablen.

Nachtrag: Rekursion verstehen

Wer bisher noch nie mit Rekursion zu tun hatte kann es mitunter sehr schwierig finden, die hier gezeigten Beispiele zu verstehen. Besonders wenn die Programme länger und komplexer werden ist es in der Tat mitunter anspruchsvoll, die Struktur nachzuvollziehen. Insofern soll an dieser Stelle noch einmal die Grundlogik von Rekursion dargestellt werden, da sie in vielen nachfolgenden Aufgaben benutzt wird. Die zentrale Idee der Technik ist es, dass man eine Funktion schreibt, die eine bestimmte Aufgabe erfüllt, und dies tut, indem Sie das Problem leicht verändert und sich selbst neu aufruft. Dies mag komisch erscheinen, aber ja, eine Funktion kann sich selbst aufrufen. Wie beim Baron Münchhausen, der sich selbst am Schopf aus dem Sumpf zieht, läuft es ab, aber in der Realität und ohne Flunkerei. Damit das klappt sind zwei zentrale Aspekte zu beachten: erstens muss es einen *base case* geben, also den Fall, bei dem keine Selbstaufrufung mehr stattfindet. Würde dies nicht der Fall sein, so gäbe es offensichtlich kein Ende und wir hätten einen unendlichen Regress, der nicht berechnet werden kann. Insofern erscheint es oftmals hilfreich, zuerst zu definieren, wann das „Ende“ der Rekursion erreicht ist. Zweitens muss klar sein, dass bei der Selbstaufrufung das zu lösende Problem nicht *identisch* mit dem vorherigen sein kann, da ansonsten ebenfalls ein unendlicher Regress entsteht. Klassischerweise wird daher das Argument der Funktion inkrementiert oder dekrementiert.

Sehen wir uns ein weiteres Beispiel an. In der Mathematik ist die Fakultät einer Zahl folgendermaßen definiert:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n = \prod_{k=1}^{n} k \tag{2.3}$$

Somit ist die Fakultät von 5 gleich 120 ($1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$). Diese Formel ist durch Rekursion umzusetzen. Wir sehen bereits an der Definition, dass immer von 1 hochgezählt wird, bis die Zahl n erreicht wird. Umgekehrt können wir von der Zahl n herunterzählen, bis wir 1 erreichen. Somit ist garantiert, dass wir alle dazwischenliegenden Zahlen erfassen. Somit sollte 1 unser *base case* sein. Weiterhin wird klar, dass immer die genau gleiche Operation durchgeführt wird (Multiplikation), nur die Argumente ändern sich. Will man also die Fakultät von n, so muss man zuerst alle Zahlen, die kleiner sind als n, miteinander multiplizieren. Die gleiche Regel gilt für n-1, usw... Setzen wir nun die Funktion um und schauen uns an, wie sich die Funktion selbst aufrufen kann.

```
1 def fak(n):
2     print("Berechne Fakultaet von:", n)
3     if n == 1:              #Base Case
4         print("Returnwert: ", 1)
5         return 1
6     else:                   #Selbstaufrufung notwendig
7         ergebnis = n * fak(n - 1)
8         print("Returnwert: ", ergebnis)
9         return ergebnis
```

Hier haben wir verschiedene Debugging-Hilfen eingebaut, damit wir der Funktion sozusagen bei der Arbeit zusehen können. Wir rufen die Funktion testweise mit dem Argument 3 auf. Hier lassen wir uns dann direkt anzeigen, mit welchem Argument die Funktion gerade aufgerufen wird. Intern wird dann zuerst geprüft: Ist der *base case* bereits erreicht? Offensichtlich nicht, denn 3 ist ungleich 1. Es kommt also die else-Bedigungung zum Tragen. Wir berechnen nun *ergebnis*. Hier wird jetzt die Rekursion aktiv. Wir multiplizieren das aktuelle n mit dem Ergebnis der Funktion für das nächst kleinere n, also n-1. Hier wird klar, was damit gemeint ist, wenn sich die Argumente verändern müssen. Statt n rufen wir die Funktion nun mit n-1 auf. Dies erscheint sinnvoll, da n eine positive ganze Zahl und offenbar größer als 1 ist, sonst wäre die

Funktion bereits beendet. Wir müssen daher versuchen, uns dem *base case* anzunähern, was durch Subtraktion erreicht wird. In diesem Moment wird auch deutlich, dass die nächste Zeile (Zeile 8) nicht erreicht wird, da zunächst erst der *neue* Funktionsaufruf von vorne durchlaufen muss. Rufen wir nun unsere Testfunktion auf und prüfen, was passiert:

```
>>> Fak(3)
Berechne Fakultaet von: 3
Berechne Fakultaet von: 2
Berechne Fakultaet von: 1
Returnwert:  1
Returnwert:  2
Returnwert:  6
6
```

Wir sehen deutlich, was geschieht. Wir rufen die Funktion zunächst mit 3 auf (Aufruf von „außen“). Hier wird die Eingabe in der if...else Bedingung überprüft. Da die Eingabe nicht 1 ist, kommt die else-Bedingung zum Tragen und es wird eine Selbstaufrufung mit n-1 (also 2) eingeleitet. Danach wiederholt sich das Spiel mit 2, was wieder zu einer Selbstaufrufung führt. Damit erreichen wir den *base case* und die if-Bedingung wird das erste Mal in der „innersten“ Funktion erreicht. Es erfolgt der erste Return (1), der nun an die zweit-innerste Funktion zurückgegeben wird. Nun kann auch diese Funktion abschließen und liefert wiederum ihr Ergebnis zurück. Wäre der *base case* größer als die Startzahl, so könnte man sich die Struktur als einen „Rekursionsturm“ vorstellen. Im gezeigten Beispiel ist es eher so, als würde man immer weiter nach unten gehen, bis man das tiefste Stockwerk erreicht und dann anfangen, das dort gefundene Ergebnis Stockwerk für Stockwerk nach oben durchzureichen. Folgende Abbildung zeigt das Vorgehen noch einmal schematisch auf (Abbildung 2.1).

Es wird ersichtlich, wie zunächst keine Funktion eine Rückgabe liefert, sondern immer neue Funktionsinstanzen erzeugt werden, was so lange geschieht, bis der *base case* erreicht wird. Erst dann

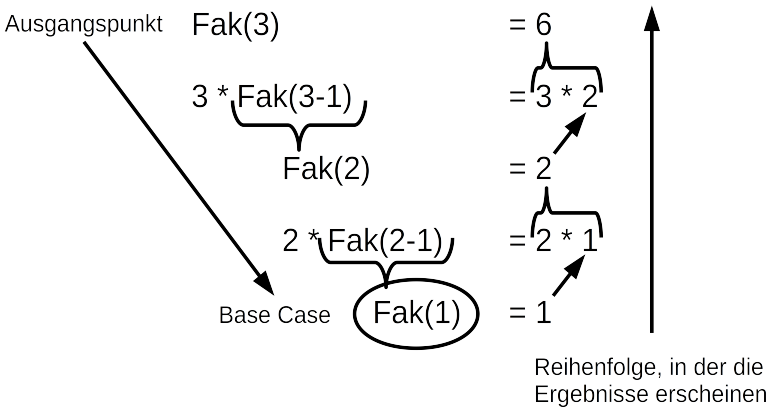

Abbildung 2.1: Schema der vorgestellten Rekursion

werden, sozusagen von unten nach oben, Ausgaben erzeugt. Dabei gilt, dass die Ausgabe einer Funktion umso später erfolgt, je früher sie aufgerufen wurde. Fak(3) wurde offenbar zuerst aufgerufen, hat aber als letztes eine Ausgabe. Zur Übung kann man versuchen, ähnliche Rechenoperationen wie beispielsweise die Multiplikation oder das Potenzieren auf die gleiche Weise zu berechnen. Die Struktur ist analog, man muss sich nur überlegen, wann genau die Rekursion enden muss und was die jeweiligen Rechenoperationen eigentlich bedeuten, bzw. welche elementaren Rechenregeln angewandt werden.

2.3 Primzahlen

Primzahlen faszinieren die Menschheit nicht nur seit Jahrtausenden, sondern haben auch einen praktischen Anwendungsbezug, bei-