Funktionale Programmierung verstehen

Konzepte und Entwurfsmuster für guten Code

Kapitel 1
Was ist funktionale Programmierung?

Funktionale Programmierung mag auf den ersten Blick durch Begriffe wie *Funktoren*, *Monoide* und *Monaden* abschreckend wirken, besonders wenn Mathematik nicht zu Ihren Hauptinteressen zählt. Doch keine Sorge, um sich mit FP auseinanderzusetzen, bedarf es keiner mathematischen Expertise.

Die Grundprinzipien der funktionalen Programmierung sind leicht verständlich, solange sie in einer klaren und geradlinigen Art präsentiert werden. Genau das ist das Anliegen dieses Buchs: eine verständliche und praxisnahe Einführung in funktionale Programmierung zu bieten. Vor allem werde ich Ihnen beibringen, wie wir in der funktionalen Programmierung *denken*. Doch warum sollten Sie sich überhaupt mit funktionaler Programmierung befassen?

Stellen Sie sich vor, es ist 22:00 Uhr, und Sie stecken komplett fest beim Versuch, einen Fehler in einem Programm zu beheben, das Sie am nächsten Morgen liefern müssen. Der Fehler hat offenbar mit einer Variablen namens `ratio` zu tun. Das Problem ist, dass sich der Wert dieser Variablen ständig ändert, abhängig vom Zustand des modellierten Systems – der Frust steigt.

Ein anderes Szenario: Die Deadline rückt näher, aber in Ihrem Microservice ist ein schwer nachvollziehbarer Fehler aufgetreten, den Sie ausfindig machen müssen. Das Problem scheint in zwei verschachtelten `for`-Schleifen zu liegen, in denen Variablen auf äußerst komplexe Weise manipuliert werden. Die Logik ist undurchschaubar, und die Lösung will sich einfach nicht zeigen. Hätten Sie doch nur eine Möglichkeit, Programme so zu schreiben, dass sich der Wert von Variablen nicht verändert! Hier kommt funktionale Programmierung ins Spiel.

Variablen, die häufig ihre Werte ändern, sind eine typische Quelle für Programmfehler. Es kann eine Herausforderung sein, den Wert einer solchen Variablen im Blick zu behalten, da er jederzeit mutieren kann.

Aber was genau ist funktionale Programmierung? Was unterscheidet eine funktionale Sprache von einer nicht funktionalen? Tatsächlich hängt dies gewissermaßen davon ab, wie weit Sie gehen möchten. Sie müssen nicht zwangsläufig alle Grundsätze der funktionalen Programmierung befolgen. Der eine Programmierer wird versuchen, alle Prin-

zipien zu befolgen, die andere Entwicklerin wird sich etwas aussuchen. Das bleibt ganz Ihnen überlassen. Funktionale Programmierung ist ein Paradigma, eine Herangehensweise ans Programmieren – eine Art und Weise, die Welt zu analysieren und in Form von Code wieder zusammenzusetzen. Dies betrifft einerseits die Art, wie wir das zu modellierende Universum organisieren, und andererseits unsere Herangehensweise an die Codestrukturierung.

Um das Wesen der funktionalen Programmierung besser zu verstehen, ist es hilfreich, sie im Vergleich mit der imperativen Programmierung und der objektorientierten Programmierung (OOP) zu betrachten. Es existieren weitere Programmierparadigmen, wie etwa die logische Programmierung, aber die genannten drei sind zweifellos die bekanntesten.

Imperative Programmierung stellt die älteste Form dar – die Programmierung vor dem Aufkommen der objektorientierten und der funktionalen Programmierung. Bei der imperativen Programmierung schreiben Sie Funktionen oder Prozeduren, verwenden `for`- und `while`-Schleifen und arbeiten häufig mit Zustandsänderungen. Sprachen wie C oder Pascal sind typische imperative Programmiersprachen. Ein weiteres Paradigma ist die objektorientierte Programmierung (OOP), die derzeit am weitesten verbreitet ist. Hierbei wird die Realität als Ansammlung von Objekten abgebildet. Jedes dieser Objekte besitzt einen eigenen Zustand sowie Methoden, also Aktionen, die ein spezifisches und objektspezifisches Verhalten repräsentieren. Während der Laufzeit des Programms kann sich der Zustand der Objekte ändern. Ein wesentlicher Vorteil dieses Ansatzes ist die Kapselung, wodurch der Zustand und die Methoden eines Objekts auf Codeebene innerhalb des Objekts gebündelt sind. Diese Herangehensweise ist weitaus sinnvoller als die Verteilung des Zustands über den gesamten Code, da die Verwaltung eines variablen Zustands problematisch sein kann. Die Manipulation zahlreicher Variablen, deren Werte sich verändern, erweist sich hierbei als Herausforderung. Im Gegensatz dazu versucht die funktionale Programmierung, veränderbare Zustände zu minimieren oder gänzlich zu umgehen.

Ein grundlegendes Prinzip der funktionalen Programmierung besteht also darin, Zustandsänderungen zu *vermeiden*, statt sie zu *verwalten*.

Dennoch ist es nicht immer möglich, komplett auf Zustandsveränderungen zu verzichten. Daher folgt der typische Ansatz in der funktionalen Programmierung dem Konzept, den Codeabschnitt zu isolieren, der für Zustandsänderungen verantwortlich ist. Falls es nicht möglich ist, sämtliche Zustandsänderungen zu umgehen, besteht zumindest die Möglichkeit, den Code, der solche Veränderungen bewirkt, an einer zentralen Stelle zu bündeln.

1.1 Unveränderlichkeit

Der wichtigste Aspekt in der funktionalen Programmierung ist die *Unveränderlichkeit*. Elemente, die wir nicht in irgendeiner Weise verändern können, gelten als unveränderlich. In der funktionalen Programmierung manifestiert sich diese Unveränderlichkeit auf verschiedene Weisen. Einmal festgelegt, kann der Wert einer Variablen nicht mehr modifiziert werden. Wenn beispielsweise die Variable `x` zu Beginn eines Programms den Wert `3` aufweist, wird dieser für die gesamte Dauer des Programmlaufs beibehalten. Doch bedeutet das, dass wir im funktionalen Programmierstil keine Möglichkeit haben, Veränderungen wie beispielsweise das Älterwerden einer Person darzustellen? Keineswegs – diese Vorstellung wäre schlicht absurd. Es existieren Techniken, die es uns ermöglichen, den Code zu manipulieren, ohne dabei den Zustand zu verändern.

Betrachten Sie diese einfache `for`-Schleife in Java, die die Zahlen 0 bis 99 ausgibt:

```
for (int i = 0; i < 100; i++) {
    System.out.println( i );
}
```

Listing 1.1 »for«-Schleife in Java

Solche Codekonstrukte sind alltäglich. Möglicherweise fragen Sie sich, wie wir sie auf eine unveränderliche Art ausdrücken können. Offensichtlich liegt der Kern dieses Codes in der Veränderung des Werts der Variablen `i`. Ein gängiger Ansatz in der funktionalen Programmierung ist die Verwendung rekursiver Funktionen – also von Funktionen, die sich selbst aufrufen. Den zuvor gezeigten Code könnten Sie in eine Funktion packen und diese in jeder Iteration für den nächsten Wert von `i` aufrufen. Ein Beispiel dazu:

```
void f(int i) {
    if (i > 99) {
        return;
    }
    else {
        System.out.println( i)
        return f(i+1)
    }
}
f(0)
```

Listing 1.2 Rekursion statt Schleife (Java)

Dieser Code mag zwar etwas ausführlicher sein, doch es erfolgt keine Zustandsveränderung. Wenn Sie bereits ein gewisses Verständnis für funktionale Programmierung besitzen, wissen Sie möglicherweise, dass der Rückgabetyp `void` ein eindeutiger Hinweis auf *Nebeneffekte* ist.

Tipp: nach »void« Ausschau halten

In der funktionalen Programmierung sollten alle Funktionen einen Wert zurückgeben. `void` ist ein sicheres Zeichen für Nebeneffekte.

Zu den Nebeneffekten zählt jegliche Aktivität, die das Programm außerhalb der Funktion beeinflusst – das Schreiben in eine Datei, das Auslösen einer Ausnahme oder die Modifikation einer globalen Variable. Das vorige Codebeispiel veranschaulicht eine Methode, um den Einsatz von zustandsverändernden Operationen zu umgehen. Während Ihrer gesamten Programmierkarriere haben Sie vermutlich Zustandsveränderungen vorgenommen, die Ihnen womöglich unverzichtbar erschienen. Allerdings sollten Sie Folgendes bedenken:

- Das Mutieren von Zuständen fühlt sich sehr natürlich an.
- Zustandsänderungen sind eine der Hauptursachen für komplexe Code-Strukturen.

Die erfreuliche Nachricht lautet: Mit ausreichender Übung und Erfahrung wird Ihnen der Ansatz der funktionalen Programmierung genauso vertraut erscheinen wie andere Programmierparadigmen.

Betrachten wir eine weitere Technik zur Vermeidung von Zustandsänderungen. Angenommen, Sie haben ein Objekt mit einer Eigenschaft oder einem Feld, das sich verändert. Die Frage ist nun, wie Sie diese Situation modellieren können, ohne eine Variable im Code zu modifizieren. Betrachten wir zunächst ein Beispiel in Java:

```
public class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
    this.age = age;
    }

    public static void main(String[] args) {
        Person person = new Person("Carl", 32);
```

```
        //Ein Jahr vergeht
        Person changedPerson = new Person("Carl", 33);
        System.out.println(changedPerson);
    }
}
```

Listing 1.3 Beispiel: Zustandsänderung vermeiden in Java

In der fettgedruckten Zeile sehen Sie: Statt den Wert des Alters im Objekt `Person` zu ändern, erzeugen wir ein neues Objekt und initialisieren den neuen Wert für `age` im Konstruktor.

Sehen wir uns nun ein Python-Beispiel an:

```
class Person:
    def init(self,name,age):
        self.name = name
        self.age = age

    def main():
        person = Person("John",22)
    #Ein Jahr vergeht
    changedPerson = Person("John",23)
```

Listing 1.4 Das gleiche Beispiel in Python

Python-Special

Mehr zu unveränderlichen Datentypen und diesem Listing finden Sie in Abschnitt B.1.

Ein Jahr vergeht, und das muss sich im `Person`-Objekt widerspiegeln. Doch wir sind nicht in der Lage, den Wert von `age` zu modifizieren. Daher erschaffen wir ein weiteres unveränderliches Objekt, in dem die Variable `age` mit 23 initialisiert wird.

Sehen wir uns ein Beispiel in Scala an:

```
case class Person(name: String, age: Int)
val person = Person("Katherine", 25)
val changedPerson = person.copy(age=26)
```

Listing 1.5 Das gleiche Beispiel in Scala

- **Zeile 1:** Eine `case`-Klasse wird deklariert.

- **Zeile 2:** Eine Instanz der Klasse wird erzeugt.
- **Zeile 3:** Eine neue `Person`-Instanz wird erstellt und das Alter auf `26` initialisiert. Dabei erfolgt keinerlei Zustandsveränderung.

Unveränderlichkeit stellt einen der wichtigsten Aspekte der funktionalen Programmierung dar. Die Existenz zahlreicher veränderlicher Zustände in einem Programm führt oft zu einer Vielzahl von Fehlern. Es gestaltet sich schwierig, den Überblick über all die sich wandelnden Werte zu behalten. In diesem Abschnitt haben wir einige Beispiele betrachtet, wie Sie die offensichtliche Anforderung der Zustandsveränderung umgehen können. Zugegeben, diese Herangehensweisen mögen anfangs ungewohnt sein, doch mit ausreichender Übung werden sie Ihnen vielleicht sogar ganz natürlich erscheinen.

1.2 Referenzielle Transparenz

Der nächste entscheidende Grundsatz der funktionalen Programmierung ist die *referenzielle Transparenz*. Ein Ausdruck ist dann referenziell transparent, wenn er an beliebiger Stelle im Code durch seinen Wert ersetzt werden kann. Bei einem ersten Blick auf diese Definition mag es so wirken, als ob dies immer möglich sei. Doch betrachten wir ein einfaches Beispiel einer nicht referenziell transparenten Funktion:

```
today()
```

Listing 1.6 Eine nicht referenziell transparente Funktion in Java

Wenn wir diese Funktion aufrufen und den 29. Dezember 2023 als Ergebnis erhalten, diesen Wert dann im Code einsetzen und die Funktion morgen erneut aufrufen, erhalten wir eine unrichtige Antwort. Infolgedessen ist die Funktion `today` nicht referenziell transparent.

Weitere Beispiele für nicht referenzielle Transparenz sind

- eine Funktion, die eine Zufallszahl zurückgibt – ihr Funktionskörper kann nicht durch einen konstanten Wert ersetzt werden, da dieser nicht vorherbestimmt werden kann,
- eine Funktion, die eine Ausnahme wirft – Ausnahmen werden in der funktionalen Programmierung normalerweise vermieden, worauf ich später eingehen werde.

Wenn wir uns komplett von nicht referenziell transparenten Funktionen distanzieren (und dies ist unser Ziel), scheinen uns möglicherweise gewisse wertvolle Möglichkeiten abhandenzukommen – wir können dann vielleicht bestimmte nützliche Konzepte nicht

mehr ausdrücken. Seien Sie versichert, dass die funktionale Programmierung Mechanismen bereithält, um diese Konzepte oder Ideen dennoch auszudrücken.

Ein verwandter Begriff aus der Literatur über funktionale Programmierung ist die *Reinheit*. Leider herrscht eine gewisse Verwirrung über die Verbindung zwischen Reinheit und referenzieller Transparenz. Zudem existiert keine allgemeingültige Definition dieser Termini. Im Allgemeinen bezeichnen wir eine Funktion als rein, wenn sie frei von jeglichen Nebeneffekten ist und für eine bestimmte Eingabe stets dieselbe Ausgabe generiert. Das bedeutet, wenn die Eingabe `x` ist und die Ausgabe `y`, wird die Funktion bei jeder Anwendung mit `x` als Eingabeparameter stets `y` liefern. Ein Nebeneffekt ist jede Aktivität außerhalb des Funktionskontexts – Beispiele hierfür sind Schreibvorgänge in Dateien oder das Auslösen von Ausnahmen. Ignorieren wir für den Moment das Schreiben in Dateien (und das Auslösen von Ausnahmen ist meist unnötig) und stellen wir uns vor, wie vorteilhaft es wäre, stets dieselbe Ausgabe zu erhalten, ohne dass sich etwas außerhalb der Funktion verändert, wenn wir die Funktion mit denselben Eingabeparametern aufrufen. Genau *das* stellt eine der Errungenschaften der funktionalen Programmierung dar.

In der funktionalen Programmierung streben wir danach, nur reine Funktionen zu verwenden – Funktionen ohne Nebeneffekte, die jedoch die Eigenschaft besitzen, bei identischer Eingabe stets die gleiche Ausgabe zu liefern.

Da es unterschiedliche Interpretationen gibt und die Unterscheidung zwischen referenzieller Transparenz und Reinheit sehr subtil sein kann, behandle ich die beiden Konzepte als gleichbedeutend.

Wie bereits erwähnt, benötigen Sie kein Mathematikstudium, um funktionale Programme zu erstellen. Dennoch ist die funktionale Programmierung in der Mathematik verwurzelt – konkret in den Bereichen Lambda-Kalkül und Kategorientheorie. Die Kategorientheorie steht in enger Verbindung zu Funktionen, und in der Mathematik sind Funktionen rein. Wenn Sie als Programmierer oder Programmiererin einen Ausdruck wie $x = x + 1$ betrachtet, denken Sie: »Ah, der Variablenwert wird erhöht.« Eine Person, die Mathematik betreibt, hingegen sagt: »Nein, das stimmt nicht.«[1]

Wie sieht nun eine unreine Funktion aus?

```
@main def impure():Unit =
  def impureFunction(x: Int): Int =
      import scala.util.Random
      return Random.nextInt(100) + x
```

1 Das war ein Scherz, aber ich habe diese Reaktionen tatsächlich einmal erlebt.

```
println(impureFunction(5))
println(impureFunction(8))
```

Listing 1.7 Eine unreine Funktion in Scala

Die beiden Aufrufe dieser Funktion würden höchstwahrscheinlich auch für denselben Eingabewert unterschiedliche Ausgabewerte liefern. Diese Funktion ist nicht rein. Wie bereits erwähnt, sind mathematische Funktionen rein, und die Programmierung hat von diesem mathematischen Ansatz stark profitiert. Funktionale Programme zeichnen sich durch Klarheit, Reinheit und Eleganz aus. Der Stil der funktionalen Programmierung mag anfangs etwas gewöhnungsbedürftig erscheinen, jedoch werden wir im Verlauf dieses Buchs schrittweise die fundamentalen Konzepte der funktionalen Programmierung durchgehen. Mit der Zeit werden Sie lernen, wie ein funktionaler Programmierer oder eine funktionale Programmiererin zu denken. Ihre Funktionen werden rein sein, Ihr Code wird klarer und sauberer.

Der bedeutendste Vorteil der funktionalen Programmierung liegt jedoch in robusteren und zuverlässigeren Programmen.

An dieser Stelle möchte ich einen wichtigen Punkt ansprechen. Funktionale Programmierung bedeutet nicht einfach das Fehlen bestimmter Merkmale aus der imperativen oder objektorientierten Programmierung. Sie stellt eine besondere Herausforderung dar, da es oft schwierig ist, alle Problemstellungen in der Softwareentwicklung auf funktionale Art auszudrücken. Die Schöpfer der funktionalen Programmierung haben Techniken entwickelt, um funktionale Lösungen für diese Anforderungen zu finden.

1.3 Funktionen höherer Ordnung

In der funktionalen Programmierung stehen Funktionen im Mittelpunkt; wir behandeln sie vorrangig. Das bedeutet, dass wir sie als Argument an andere Funktionen übergeben können und auch als Rückgabewert aus Funktionen erhalten können. Ich möchte erklären, warum Funktionen höherer Ordnung einen bedeutsamen Teil der funktionalen Programmierung darstellen. Ein Hauptziel der funktionalen Programmierung besteht darin, die Essenz der Aufgabenstellung herauszuarbeiten. Das bedeutet, dass wir in der Lage sein sollten, Konzepte in unserer Sprache präzise auszudrücken. Wenn wir zum Beispiel beabsichtigen, jedes Element einer Liste von Ganzzahlen zu quadrieren, sollten wir nicht durch eine Schleife gehen und jedes Element einzeln quadrieren müssen. Stattdessen sollten wir in der Lage sein, eine `square`-Funktion direkt auf jedes Element der Liste gleichzeitig anzuwenden. In vielen Sprachen ermöglicht dies die Verwendung

der `map`-Funktion. Mit ihr können wir auf einer abstrakteren Ebene arbeiten, die den Konzepten einer Funktion höherer Ordnung entspricht. Dieses Thema wird im weiteren Verlauf des Buchs noch sehr wichtig.

Hier ein imperativer Ansatz:

```
def square(nums):
    squared = []
    for i in nums:
        squared.append(i*i)
    return squared
```

Listing 1.8 Eine Liste bearbeiten, imperativer Ansatz (Python)

Python-Special

Schauen Sie sich zu diesem Codebeispiel auch das Listing B.1 im Python-Special an.

Und ein funktionaler Ansatz:

```
def square(nums):
    return map(lambda n: n*n, nums)
```

Listing 1.9 Eine Liste bearbeiten, funktionaler Ansatz (Python)

Wie im Abschnitt 2.2.1 gezeigt wird, ist `lambda` eine Möglichkeit, eine *anonyme Funktion* zu erstellen – eine namenlose Funktion, die direkt und ohne vorherige Definition verwendet wird. Die `map`-Funktion erlaubt die Anwendung einer Funktion auf jedes Element einer Liste.

1.4 Lazy Evaluation

Eine weitere Komponente der funktionalen Programmierung ist die *Lazy Evaluation*, auch *verzögerte Auswertung* oder *Bedarfsauswertung*. Dies bedeutet einfach, dass ein Ausdruck erst dann ausgewertet wird, wenn seine Auswertung tatsächlich erforderlich ist. Obwohl dies streng genommen nicht zwingend nötig ist, um eine Sprache als funktional zu bezeichnen, tendieren naturgemäß viele funktionale Sprachen dazu, dieses Prinzip der verzögerten Auswertung zu verwenden. Ein gutes Beispiel hierfür ist die Programmiersprache Haskell, die standardmäßig die Lazy Evaluation nutzt und als prototypische funktionale Programmiersprache angesehen werden kann. Haskell wurde von einem wissenschaftlichen Team entwickelt und geht keinerlei Kompromisse hinsicht-

lich funktionaler Prinzipien ein. Die meisten verbreiteten Programmiersprachen verwenden im Gegensatz dazu die sogenannte *Eager Evaluation*, bei der Ausdrücke sofort ausgewertet werden, sobald sie im Code auftreten. Der folgende Aspekt veranschaulicht die Vorteile der Lazy Evaluation:

Stellen Sie sich vor, Sie möchten Ihre eigene Version der if-Anweisung definieren. Nennen wir diese Funktion `myIf`. Möglicherweise möchten Sie jedem Aufruf der `if`-Anweisung eine Protokollierungszeile hinzufügen. Beim folgenden Versuch könnten Sie jedoch auf ein Problem stoßen:

```
def myIf(condition: Boolean, thenAction: => Unit, elseAction: => Unit): Unit =
    if (condition)
            thenAction
    else elseAction
```

Listing 1.10 Scala

Erkennen Sie das Dilemma? In Programmiersprachen mit *Eager Evaluation*, wie sie in den meisten Fällen üblich ist, erfolgt zuerst die Auswertung aller Argumente eines Funktionsaufrufs. Im Falle von `myIf` werden sowohl die Anweisungen für den `then`-Zweig als auch für den `else`-Zweig ausgewertet, obwohl je nach Bedingung nur eine von beiden tatsächlich benötigt wird. Bei der Verwendung von Lazy Evaluation hingegen würde dieser Ansatz funktionieren. In solchen und ähnlichen Szenarien haben Sie die Möglichkeit, eigene Kontrollstrukturen zu entwerfen, die von der verzögerten Auswertung profitieren.

Lazy Evaluation und Eager Evaluation

Unter Eager Evaluation werden die Parameter einer Funktion sofort ausgewertet, sobald die Funktion aufgerufen wird. Im Gegensatz dazu erfolgt bei der Lazy Evaluation die Auswertung der Parameter erst dann, wenn ihre Ergebnisse tatsächlich benötigt werden.

Ein weiterer Vorteil ist die Performancesteigerung in bestimmten Situationen. Durch die Verwendung von lazy Code wird die Auswertung nur dann durchgeführt, wenn sie tatsächlich benötigt wird. Daher erfolgt die Auswertung oft seltener im Vergleich zur Eager Evaluation. Dies kann zur Beschleunigung des Programms beitragen.

In Scala haben wir die Möglichkeit, benannte Parameter zu verwenden. Im folgenden Codebeispiel werden `then` und `else` erst dann ausgewertet, wenn ihre Auswertung erforderlich ist. Anders ausgedrückt, die Auswertung erfolgt *lazily*. Der nachfolgende Code funktioniert wie erwartet:

```
def myIf(condition: Boolean, thenAction: => Unit, elseAction: => Unit): Unit =
    if (condition)
            thenAction
        else elseAction
```

Listing 1.11 Scala

Lazy Evaluation bietet uns also die Möglichkeit, unsere eigenen Versionen von Operatoren wie `if` oder `while` zu erstellen.

Zusammenfassung:

- Lazy Evaluation ermöglicht es Ihnen, die Kontrollflussstrukturen in Ihrem Code direkt zu definieren, anstatt sie als eingebaute Operatoren in der Sprache zu nutzen. Dank Lazy Evaluation müssen Kontrollflussstrukturen nicht zwangsläufig als Sprachkonstrukte implementiert werden. Stattdessen können Sie sie in Ihrem Code eigenständig definieren.
- Die Leistungsfähigkeit Ihres Programms kann verbessert werden.

1.5 Funktionale Denkweise

Der Fokus dieses Buchs liegt darauf, die Denkweise eines funktionalen Programmierers zu vermitteln. Obwohl es verschiedene Ansätze für die funktionale Programmierung gibt, haben sie dennoch einige grundlegende Aspekte gemeinsam. Beispielsweise werden in der funktionalen Programmierung Zustände nicht verändert. Das bedeutet, dass eine einmal gesetzte Variable niemals wieder verändert wird. Des Weiteren kommen in der funktionalen Programmierung häufig Funktionen höherer Ordnung zum Einsatz. Diese Funktionen akzeptieren andere Funktionen als Parameter und/oder geben selbst Funktionen als Rückgabewert zurück.

Muster

Um sich wirklich in die Denkweise eines funktionalen Programmierers hineinzuversetzen, sollten Sie eine Reihe von Idiomen oder Mustern kennenlernen, die funktionale Codestrukturen ermöglichen.

Es reicht nicht, lediglich zu wissen, dass Sie Variablen nicht mutieren sollten. Wenn Sie nicht wissen, wie Sie dieses Konzept umsetzen können, könnte die Implementierung von Unveränderlichkeit möglicherweise verwirrend erscheinen. Mit anderen Worten,

funktionale Muster sind ein wesentlicher Bestandteil der funktionalen Programmierung.

Möglicherweise haben Sie gehört, dass Entwurfsmuster in der funktionalen Programmierung nicht so bedeutsam sind wie in der objektorientierten Programmierung. Dies ist ein Missverständnis. Richtig ist, dass der Begriff *Entwurfsmuster (*oder *Design Pattern)* im funktionalen Kontext etwas anderes bedeutet als die Entwurfsmuster der *Gang of Four*, die in dem Buch »Entwurfsmuster« von Erich Gamma et al. (Addison-Wesley) behandelt werden[2]. Die Gang-of-Four-Muster (wie Prototyp, Stellvertreter und Fliegengewicht) wurden im Rahmen der objektorientierten Programmierung entwickelt. Sie können zwar größtenteils im funktionalen Stil umgesetzt werden und sind für den Entwurf von Programmen nützlich, jedoch sind sie nicht unbedingt funktional. Man könnte sagen, die Gang-of-Four-Muster sind funktional neutral. Jedoch existiert eine separate, explizit funktionale Musterkategorie. Diese Entwurfsmuster, wie beispielsweise die Funktor- und Monaden-Muster, sind aus Ideen der Kategorientheorie abgeleitet. Wir werden uns in Kapitel 3, »Kategorientheorie und Entwurfsmuster«, genauer mit ihnen beschäftigen.

1.6 Die Vorteile der funktionalen Programmierung

Die Vorteile der funktionalen Programmierung werden immer deutlicher erkennbar. Sie unterstützt unser Bestreben nach fehlerfreiem – oder zumindest möglichst fehlerfreiem – Code. Doch wie erreichen wir dieses Ziel? Indem wir unsere Denkweise ändern und die Welt nicht mehr als Ansammlung von Objekten betrachten, von denen jedes seinen eigenen veränderlichen Zustand besitzt, sowie Prozessen, die diesen Zustand verändern. Durch diesen Perspektivwechsel erkennen wir den Zustand als Problemquelle.

Zustandsüberwachung

Veränderliche Zustände müssen wir überwachen, was zu erhöhtem Steuerungsaufwand und größerer Fehleranfälligkeit führt. Diese Probleme werden durch die funktionale Programmierung gelöst.

Der Mensch kann nur eine begrenzte Komplexität bewältigen – irgendwann beginnt er, fehlerhaften Code zu schreiben. Sie könnten nun einwenden: »Aber die Welt besteht nun einmal aus Objekten. Diese Objekte haben Zustände, und diese Zustände ändern

2 Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2014). *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. mitp. ISBN: 978-3826697005.

sich im Laufe der Zeit! Daher ist es richtig, die Welt auf diese Weise zu modellieren. So ist eben die Realität!« Das stimmt – dennoch können wir beginnen, die Welt in funktionalen Konzepten zu erfassen (und zu modellieren). Aktuell ist es wichtig zu erkennen, dass wir tatsächlich noch nicht wissen, wie man fehlerfreie Software schreibt. Ich kenne einen Informatikprofessor, der seine Einführung in die Programmierung mit den Worten begann: »Die Menschheit weiß noch nicht, wie man programmiert.«

Das mag etwas dramatisch klingen, trifft jedoch ins Schwarze. In der Regel überschreiten Projekte ihr Budget und dauern wesentlich länger als ursprünglich geplant. Die Ursache ist ihre Komplexität. Programmieren ist die Kunst, Wissenschaft und Technik, mit dieser Komplexität umzugehen. Die funktionale Programmierung stellt uns Instrumente zur Verfügung, die es uns ermöglichen, die Komplexität zu reduzieren und zu kontrollieren. Zu diesen Werkzeugen gehören Unveränderlichkeit, referenzielle Transparenz und Funktionen höherer Ordnung, um nur einige zu nennen. Wenn Sie diese Instrumente beherrschen, schreiben Sie weniger fehleranfälligen Code.

1.6.1 Funktionale Programmierung kann die Produktivität steigern

Funktionale Programmierung ist also ein Programmierparadigma. Welche anderen Paradigmen gibt es?

Das wohl bekannteste ist die *objektorientierte Programmierung* oder *OOP*. Obwohl derzeit mehr Code im OOP-Stil existiert, ist bei vielen Entwicklerinnen und Entwicklern ein deutlicher Trend in Richtung funktionaler Programmierung zu erkennen. Die zukünftige Entwicklung bleibt abzuwarten. Es könnte sein, dass ein hybrider Ansatz, der beide Paradigmen verknüpft, zur Norm wird. Möglicherweise wird sich die funktionale Programmierung aber auch einer zunehmenden Beliebtheit erfreuen.

Wenn Sie beispielsweise bereits in Java, C#, C++ oder Python programmiert haben, sind Sie vermutlich mit der objektorientierten Programmierung vertraut. Hier wird die Welt als Ansammlung von Objekten modelliert, von denen jedes einen individuellen Zustand und ein eigenes Verhalten besitzt. OOP bringt zahlreiche Vorteile mit sich, wie Abstraktion, Kapselung und Vererbung. Trotz dieser klaren Vorzüge gehen jedoch oft ein hoher Kostenaufwand und Überstunden mit der Codeentwicklung einher. Vor dem Aufkommen der funktionalen Programmierung und der OOP gab es die imperative Programmierung. Zwar mag diese auf den ersten Blick der funktionalen Programmierung ähneln, jedoch wird bei genauerer Betrachtung deutlich, dass Zustände veränderlich, Funktionen nicht referenziell transparent sind und imperative Sprachen nicht zwangsläufig Funktionen höherer Ordnung einbeziehen. Beispiele für imperative Programmiersprachen sind C und Pascal.

Kapitel 4
Funktionale Datenstrukturen

Datenstrukturen gehören gemeinsam mit Algorithmen zu den Konzepten, die Sie als Informatikstudent oder Programmiererin beherrschen müssen. Genau wie das *Funktionsmuster* ist auch die *funktionale Datenstruktur* kein etabliertes Konzept mit einer einheitlichen Definition im Codekanon.

In diesem Buch bezeichne ich zwei Konzepte als funktionale Datenstrukturen:

- Strukturen, die in Funktionsmustern verwendet werden, wie beispielsweise `Option`, `Either`, `Try`, `List`. Diese Strukturen sind Monaden.
- Herkömmliche Datenstrukturen, die auf funktionale Weise implementiert sind, wodurch ihr Zustand unveränderlich bleibt, wie beispielsweise verkettete Listen.

In diesem Kapitel befassen wir uns mit der ersten Art von funktionalen Datenstrukturen und anschließend kurz mit einigen Überlegungen zu normalen Datenstrukturen, die auf funktionale Weise implementiert wurden. Bevor wir uns mit bestimmten Datenstrukturen beschäftigen, möchte ich eine Idee über funktionale Datenstrukturen erwähnen, die in der Literatur diskutiert wurde. Dazu zunächst ein Zitat von Alan Perlis[1]:

> *Es ist besser, 100 Funktionen auf einer Datenstruktur anzuwenden, als 10 Funktionen auf 10 Datenstrukturen einzusetzen.*

Erfahrene funktionale Programmierer nutzen häufig eine begrenzte Auswahl an Datenstrukturen: zum Beispiel verkettete Listen, Arrays und Hashtabellen sowie Strukturen wie `Option`, `Either` und `Try`. Mit diesen beschäftigen wir uns im Folgenden. Meiner Ansicht nach liegt dem genannten Zitat die Idee zugrunde, dass eine geringere Vielfalt an Datenstrukturen zu einem kohärenteren und einfacheren Code führt. Sehen wir uns nun einige besonders funktionale Datenstrukturen näher an. Wir beginnen mit `Option`.

1 *https://oreil.ly/5lY7y*

4.1 Die Option-Datenstruktur

Ich verwende das Wort *Option* als sprachneutralen Begriff (statt als Bestandteil einer bestimmten Programmiersprache). Verschiedene Programmiersprachen bieten eine Version dieser Datenstruktur an, und ich werde einige Beispiele hierzu aufzeigen. Lassen Sie uns jedoch zunächst das Konzept konkretisieren. Idealerweise sollten wir mit dem `null`-Konstrukt beginnen. `null` repräsentiert einen möglicherweise nicht vorhandenen Wert, eine optionale Ausprägung oder eine Situation, in der der Wert unbekannt ist. Ein Problem im Zusammenhang mit `null` besteht darin, dass ein Programm eine Nullzeiger-Ausnahme (`NullPointerException`) auslöst, wenn eine Variable ein Objekt enthält, aber der Wert dieser Variablen `null` ist und für dieses Objekt eine Methode aufgerufen wird. Ausnahmen unterbrechen die referenzielle Transparenz und sind daher im Kontext des funktionalen Programmierens unerwünscht. Tony Hoare, der Urheber der Null-Referenz `null`, nannte diese seinen »Milliarden-Dollar-Fehler«:

> *Ich bezeichne ihn als meinen Milliarden-Dollar-Fehler. Es war die Einführung der Null-Referenz im Jahr 1965. Zu jener Zeit entwickelte ich das erste umfassende Typsystem für Referenzen in einer objektorientierten Sprache (ALGOL W). Mein Ziel war es, eine uneingeschränkt sichere Verwendung von Referenzen sicherzustellen, wobei die Überprüfung automatisch durch den Compiler erfolgt. Doch der Versuchung, eine Null-Referenz einzuführen, konnte ich nicht widerstehen, einfach weil ihre Implementierung so einfach war. Dies führte zu zahllosen Fehlern, Schwachstellen und Systemabstürzen, die in den letzten vierzig Jahren vermutlich Schäden und Probleme in Höhe von einer Milliarde Dollar verursacht haben.*[2]

Doch wie gehen wir ohne die Verwendung von `null` mit einem Datenelement um, das optional ist oder keinen Wert haben darf? Nun, eines der Hauptprobleme von `null` besteht darin, dass es keine spezifische Typzuweisung aufweist. Wie wäre es, wenn wir einen Typ schaffen könnten, der den Fall eines fehlenden oder optionalen Werts repräsentiert? Dies ist in vielen Programmiersprachen bereits möglich. Falls es in einer Programmiersprache nicht vorgesehen ist, ist es dennoch nicht schwer, dies umzusetzen. Es gibt verschiedene Wege, auf diesen Typ zu verweisen, jedoch verwenden die meisten Varianten das Konzept der Option. Ein bekanntes Beispiel dafür ist `Option` in Java. In Python gibt es ein Objekt, das in manchen Aspekten `Option` ähnelt, jedoch nicht die volle Funktionalität aufweist: `None`. Dennoch ist es möglich, in Python ein Konzept ähnlich der Option zu implementieren. Die Grundidee hinter `Option` ist folgende: Stellen Sie sich

2 Hoare, Tony. »Null References: The Billion Dollar Mistake.« Historically Bad Ideas. Vortrag auf der QCon, 2009, *https://oreil.ly/sQSnH*.

vor, wir haben eine Methode, die eine `id` als Parameter erhält und ein `Customer`-Objekt zurückgibt. Wie reagieren wir, wenn es keinen `Customer` mit der angegebenen `id` gibt?

Vorsicht mit Ausnahmen

Es wäre zwar möglich, eine Ausnahme auszulösen, wenn `Customer` nicht gefunden wird. Jedoch widerspricht dies den funktionalen Prinzipien, da die referenzielle Transparenz verletzt würde. Wenn eine Funktion eine Ausnahme auslöst, ist nicht mehr gewährleistet, dass dieselbe Eingabe stets dieselbe Ausgabe ergibt.

Selbst wenn Sie Ausnahmen mögen, ist dieses Beispiel kein angemessener Anwendungsfall für eine solche. Eine Ausnahme sollte nicht auftreten, geschieht aber dennoch. Es gibt kaum Grund zur Annahme, dass eine vom System generierte `id` tatsächlich einem existierenden `Customer` entspricht. Diese Situation ist keineswegs ungewöhnlich und könnte sogar als erwartet betrachtet werden. Wie also könnten wir ohne eine Ausnahme mit dieser Situation umgehen? Hier kommt das Option-Konstrukt ins Spiel. Mit »Option« meine ich eine allgemeine Datenstruktur und nicht ein Konstrukt in einer bestimmten Programmiersprache. Eine Option besteht im Wesentlichen aus zwei Teilen: einem Container, der einen Wert beinhaltet, wenn ein gültiger Wert auf irgendeine Art und Weise berechnet oder erhalten wurde, sowie einem zweiten Teil, der den Fall eines fehlenden gültigen Werts abdeckt. In einer Sprache ohne Option kann in dieser Situation stattdessen oft der Typ `null` verwendet werden. Ein Beispiel in Scala verdeutlicht dies. Betrachten wir erneut die Situation, in der ein Kunde anhand einer `id` abgerufen werden soll. In Scala können wir folgendermaßen vorgehen:

```
def getCustomer(id: Int): Option[Customer] = {
    // Kunde aus der Datenbank abrufen
}
```

Listing 4.1 Nochmal einen Kunden anhand einer ID abrufen (Scala)

Diese Funktion kann zwei verschiedene Rückgabewerte liefern. Wenn `id` einem Kunden entspricht, gibt die Methode zurück:

```
Some(customer)
```

Existiert kein `Customer` für die angegebene `id`, wird zurückgegeben:

```
None
```

Damit können geeignete Maßnahmen für diese Situation ergriffen werden. Falls `Some (customer)` zurückgegeben wird, kann der Kunde auf verschiedene Weisen aus dem

Some-Container extrahiert werden. Ein gängiger Weg ist die Verwendung der Pattern-Matching-Funktion in Scala:

```
getCustomer(17) match {
    case Some(customer) => //etwas mit customer tun
    case None => // Maßnahmen zur Behandlung dieser Situation ergreifen
}
```

Listing 4.2 Auf »Some« und »None« reagieren mit »match« (Scala)

Ein anderer, vielleicht gebräuchlicherer Ansatz ist die Verwendung einer Funktion höherer Ordnung wie `map`, `flatMap` oder `filter`. Ein Beispiel:

```
val customer = getCustomer(17) map { customer => customer.firstName}
```

Wenn ein Wert gefunden wurde, enthält `customer` den Wert `Some("Peter")`. Wurde kein `customer` gefunden, wird `None` zurückgegeben, was vom Programmierer entsprechend abgefangen werden kann.

»Some« und »None« sind Typen

Es ist wichtig zu verstehen, dass `Some` und `None` Typen sind und der Compiler daher Fehler in diesem Kontext erkennen kann – anders als bei `null`, das erst zur Laufzeit (nicht zur Kompilierzeit) auftritt.

Betrachten wir nun eine Liste von `Option`-Objekten, die entweder `Some(customer)` oder `None` enthalten. Und nehmen wir an, Sie wollten diese Liste in eine Kundenliste umwandeln und dabei die `None`-Werte ignorieren. Nehmen wir weiter an, `getCustomers` gibt eine Liste von Optionen von `Customer`-Objekten zurück. Das `Customer`-Objekt könnte durch eine `case class` repräsentiert werden:

```
case class Customer(id: Int, firstName: String)
```

Die Liste der `Customer`-Optionen könnte so aussehen:

```
val customers = List(Some(Customer(1,"Bob")), None, Some(Customer(33, "Lee")),
   None, Some(Customer(5, "Rhonda")))
```

In diesem Fall könnten wir folgendermaßen vorgehen:

```
customers.flatten
```

Das Ergebnis wäre:

```
List(Customer(1, "Bob"), Customer(33, "Lee"), Customer(5, "Rhonda"))
```

Nun betrachten wir einige Beispiele in Java:

```
     Optional<User> optUser = findUserById(123);

     optUser.ifPresent(user -> {
          System.out.println("User's name = "
     + user.getName());})
}
```

Listing 4.3 Beispiele in Java

In diesem Beispiel gibt `findUserById` ein `Optional` eines `User`-Objekts zurück, wenn eines für die jeweilige `id` gefunden wird. Andernfalls wird der Code in den geschweiften Klammern nicht ausgeführt. Eine ältere Methode wäre gewesen, dass `findUserById null` zurückgibt, wenn kein `User` gefunden wurde. Das Problem dabei ist, dass eine `NullPointerException` auftritt, wenn eine Methode auf das `User`-Objekt aufgerufen wird und tatsächlich `null` zutrifft. Beim vorangehenden Code mit `Optional` wird keine Ausnahme ausgelöst.

In Python existiert keine `Option`-Klasse. Es gibt zwar das `None`-Objekt, doch dieses deckt nur die eine Hälfte des Option-Konzepts ab. Dennoch kann es nützlich sein, wie im folgenden Beispiel:

```
def getUser(id):
    #Nutzer aus Datenbank abrufen

    #bei Fehlschlag
    return None
```

Listing 4.4 Das »None«-Objekt in Python

Dann können Sie bei der Verwendung dieser Funktion wie folgt vorgehen:

```
if getUser(789) is not None:
   #Aktion durchführen
```

Sie könnten anmerken, dass dies stark nach einer Nullprüfung aussieht, und ich stimme Ihnen zu. Wenn Sie wirklich möchten, könnten Sie in Python eine `Option`-Klasse erstellen. Eine Möglichkeit sehen Sie hier:

```
class Option:
    def get_or_else(self, default):
        return self.value if isinstance(self, Some) else default
class Some(Option):
    def __init__(self, value):
        self.value = value
class Nothing(Option):
    pass
```

Listing 4.5 Eine »Option«-Klasse in Python

Ich habe `Nothing` gewählt, weil Python bereits ein `None`-Objekt enthält. Das ist nicht besonders »pythonisch«, bietet jedoch eine Möglichkeit, die Aufgabe zu bewältigen.

In C# existiert ebenfalls eine Version dieses Konzepts. Zwar handelt es sich dabei nicht direkt um eine `Option`-Struktur, jedoch um eine Konstruktion, um das Problem der `null`-Werte bei bestimmten Datentypen zu behandeln: Mit dem Typ `Nullable` können wir explizit darstellen, dass eine Variable einen `null`-Wert annehmen kann. Für einen gegebenen Typ, beispielsweise `int`, kann der `Nullable`-Typ verwendet werden:

```
Nullable<int>
```

Listing 4.6 Syntax für »Nullable«-Typen in C#

Es handelt sich hierbei um einen generischen Typ. Eine kurze Schreibweise dafür ist:

```
int?
```

Ein Beispiel für die Verwendung:

```
Nullable<int> n1 = new Nullable<int>(10);
Nullable<int> n2 = null;

if (n1.HasValue) {
   process(n1.Value);
} else {
  // n1 enthält einen null-Wert.
}
```

Listing 4.7 Verwendung eines »Nullable«-Typen in C#

Dies ist eine Möglichkeit, `null`-Werte sicher zu behandeln und eine `NullPointerException` zu vermeiden.

4.2 Die Try-Datenstruktur

Das `Option`-Konstrukt ist zwar nützlich und stellt eine Verbesserung gegenüber `null` dar, aber es bietet im Gegensatz zu Ausnahmen keinen Hinweis darauf, warum kein gültiger Wert vorhanden ist. Optionen sind in vielen Situationen eine großartige Lösung, aber manchmal benötigen Sie Informationen darüber, was schiefgelaufen ist. Das `Try`-Konstrukt geht auf dieses Problem ein. So wie es bei `Option` `Some` und `None` gibt, gibt es bei `Try` `Success` und `Failure`. `Failure` umschließt die geworfene Ausnahme. Zwar versuchen wir in der funktionalen Programmierung, Ausnahmen so weit wie möglich zu vermeiden, jedoch ist das nicht immer praktikabel. `Try` ist eine Lösung für dieses Problem. Hier etwas Code:

```
def divide(a: Float, b: Float): Try[Float] = Try(a/b)
```

Listing 4.8 Ausnahmen vermeiden mit »Try« (Scala)

Daraufhin könnten wir folgendermaßen vorgehen:

```
divide(3, 0) match {
   case Success(result) => // Aktionen mit dem Ergebnis durchführen
   case Failure(ex) => println(ex.getMessage)
}
```

Listing 4.9 Zur Auswertung von »Try« gibt es »Success« und »Failure«. (Scala)

Ein Beispiel für `Try` in einer `for`-Abstraktion:

```
def toInt(s: String): Try[Int] = Try(Integer.parseInt(s.trim))

val y = for {
    a <- toInt("9")
    b <- toInt("3.5")
    c <- toInt("6")
} yield a + b + c
```

Listing 4.10 Auch eine »for«-Schleife kann mit »Failure« umgehen. (Scala)

Beachten Sie, wie gut der Fehlerfall behandelt wird. `Y` enthält entweder ein `Success`, das die Summe von `a`, `b` und `c` umhüllt, oder ein `Failure` mit einer Ausnahme.

B.6 Codebeispiele zu Kapitel 4, »Funktionale Datenstrukturen«

Ich gehe nun darauf ein, wie Sie einige der wichtigsten Scala-Typen in Python abbilden können. Hier werden Sie meist auf Drittbibliotheken zurückgreifen.

B.6.1 Scala-Option wird Python-Maybe

Um die Beispiele von Abschnitt 4.1 über die Datenstruktur »Option« in Python nachbilden zu können, hilft es zu wissen, welche Typen die einzelnen Bestandteile der Listings haben. Das hilft dann auch zu verstehen, wie Sie mit Monaden arbeiten.

Hier ist das Scala-Fragment:

```
val customer = getCustomer(17) map { customer => customer.firstName}
```

Wie im Text erwähnt, liefert `getCustomer()` ein `Option[Customer]` zurück. Das `map`-Ergebnis ist dann ein `Option[String]`. Man beachte, dass die Funktion für `map` direkt von `Customer` nach `String` geht – also ohne `Option`, denn das ist ja der *Behälter*.

In purem Python würden wir mit `None` arbeiten, was aber keine Monade ist. Fallunterscheidungen würden sich durch den Code ziehen. Sie können natürlich mit den Implementierungen wie in Listing 4.4 und Listing 4.5 arbeiten. Eine komplette Implementierung finden Sie mit `Maybe` in `returns`:

```
from collections import namedtuple
from returns.maybe import Maybe, Some, Nothing

Customer = namedtuple("Customer", ["first_name"])

def get_customer(id: int) -> Maybe[Customer]:
    return Some(Customer(f"Peter")) if id == 2 else Nothing

some_customer = get_customer(2).map(lambda customer: customer.first_name)
no_customer = get_customer(17).map(lambda customer: customer.first_name)
```

Listing B.7 »Maybe«, »Some« und »Nothing« aus dem Modul »returns«

B.6.2 Scala-List wird Python-Stream

Python-Listen sind leider keine Monaden, aber Sie können mit `streams.stream` aus dem Modul `pyxtension` oder `streamerate` eine Monade daraus machen. Leider hat `stream` keine `flatten()`-Methode, also verwenden Sie dessen `filter()`-Methode. Da `Maybe` und

stream nicht voneinander wissen, müssen Sie beim Filtern mit einem entsprechenden Prädikat nachhelfen:

```
from collections import namedtuple
from returns.maybe import Maybe, Some, Nothing
from pyxtension.streams import stream

Customer = namedtuple("Customer", ["first_name"])

customers = ( Some(Customer("Bob")), Nothing, Some(Customer("Lee")), Nothing,
Some(Customer("Rhonda")) )

result = stream(customers).filter(lambda x: x != Nothing).toList()
```

Listing B.8 Aus einer Python-Liste kann man eine Monade machen.

Das ist länger, als das eingebaute `filter(lambda..., customers)` zu benutzen, aber sobald Sie mehrere Operationen auf der `stream`-Monade ausführen, lohnt sich das Einpacken.

B.6.3 Scala-Try zu Python-Result

Einen Datentyp wie `Try`, der automatisch Exceptions abfängt, haben wir in Python nicht. Aber das Modul `returns` hilft mit `Result` und dem `@safe`-Dekorator aus.

```
from returns.result import Result, Success, Failure, safe

@safe
def divide(a: float, b: float) -> float:
    return a / b

match divide(3, 0):
    case Success(result):
        return result
    case Failure(error):
        print(error)
```

Listing B.9 Den Fehlerfall abfangen mit den Mitteln aus dem Modul »returns.result«

Eigentlich wirft `3 / 0` eine Exception. Die könnten wir in einem `try-except`-Block abfangen und entweder `Success(a/b)` oder `Failure(...)` zurückgeben. Das macht `@safe` für uns. Die dekorierte Funktion hat in Wahrheit als Rückgabetyp `Result[float, Exception]`.

Das zweite Beispiel für `Try` ist in Python nicht einfach nachzubilden. Es gibt natürlich in Python ein ähnlich mächtiges `for` wie in Scala, und es arbeitet auch mit `Result` zusammen, da dieses `__iter__()` implementiert:

```
from returns.result import Result, Success, Failure, safe

@safe
def to_int(s: str) -> int:
    return int(s)

st = next(
    a + b + c
    for a in to_int("9")
    for b in to_int("3")
    for c in to_int("6")
)
```

Listing B.10 Noch mal »returns.result« im Einsatz

Das klappt, weil in `a`, `b` und `c` der umgewandelte `int` steckt, denn `__iter__()` packt es aus der `Success`-Monade aus. Stand da aber zum Beispiel `to_int("3.5")`, dann ist das Ergebnis eine `Failure`-Monade, deren `iter()` die eingeschlossene Exception zurückliefert. Und die kann mit `a + b + c` natürlich nur scheitern.

Es gibt mehrere Möglichkeiten, dieses Problem funktional zu lösen. Da wir hier sowieso schon mit `Result` aus `returns` arbeiten, können wir die `do()`-Methode verwenden, die es in `returns` für jede Monade gibt:

```
st = Result.do(
    a + b + c
    for a in to_int("9")
    for b in to_int("3.5")
    for c in to_int("6")
)
```

Der Generator ist derselbe wie in dem `for`-Beispiel zuvor. `do()` ruft intern `next()` auf und fängt die Exception. Das Ergebnis ist dann eine `Failure`-Monade mit der eingepackten Exception. Trat keine Exception auf, ist das Ergebnis eine `Success`-Monade mit dem gültigen Wert.