KAPITEL 4

Fallstudie: Gestaltung von Schnittstellen

In diesem Kapitel wird eine Fallstudie präsentiert, die zeigt, wie man Funktionen entwickelt, die miteinander arbeiten.

Es führt das `turtle`-Modul ein, mit dessen Hilfe Sie Bilder über Turtle-Grafik erzeugen können. Das `turtle`-Modul ist in den meisten Python-Installationen enthalten, doch wenn Sie Python über PythonAnywhere ausführen, werden Sie die Turtle-Beispiele nicht ausführen können (zumindest ist das so, während diese Zeilen geschrieben werden).

Wenn Sie Python bereits auf Ihrem Computer installiert haben, sollten Sie die Beispiele ausführen können. Anderenfalls ist nun ein guter Zeitpunkt zur Installation. Anweisungen auf Englisch finden Sie unter *http://tinyurl.com/thinkpython2e*.

Codebeispiele aus diesem Kapitel finden Sie unter *https://oreilly.de/9783960091691*, die Datei heißt *polygon.py*.

Das turtle-Modul

Um zu prüfen, ob Sie das `turtle`-Modul besitzen, öffnen Sie den Python-Interpreter und geben Folgendes ein:

```
>>> import turtle
>>> bob = turtle.Turtle()
```

Wenn Sie diesen Code ausführen, sollte sich ein kleines Fenster mit einem kleinen Pfeil (der die Schildkröte repräsentiert) öffnen. Schließen Sie das Fenster.

Erstellen Sie eine Datei mit dem Namen *meinpolygon.py* und tippen Sie den folgenden Code ein:

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

Das `turtle`-Modul (mit einem kleinen *t*) stellt die Funktion `Turtle` (mit einem großen *T*) zur Verfügung, die ein Turtle-Objekt erzeugt, das wir einer Variablen namens `bob` zuweisen. Wenn Sie `bob` ausgeben, erhalten Sie in etwa Folgendes:

```
<turtle.Turtle object at 0xb7bfbf4c>
```

Das bedeutet, dass sich `bob` auf ein Objekt einer Schildkröte (»Turtle«) in `Turtle` bezieht.

`mainloop` weist das Fenster an, darauf zu warten, dass der Benutzer etwas macht. In diesem Fall kann der Benutzer allerdings nicht mehr tun, als das Fenster zu schließen.

Sobald Turtle erzeugt wurde, können Sie eine **Methode** aufrufen, um sie im Fenster zu bewegen. Eine Methode ähnelt einer Funktion, verwendet aber eine etwas andere Syntax. Zum Beispiel können Sie die Schildkröte vorwärts bewegen:

```
bob.fd(100)
```

Die Methode `fd` ist mit dem turtle-Objekt `bob` verknüpft. Der Aufruf einer Methode ist wie eine Anfrage: Wir bitten `bob`, sich vorwärts zu bewegen.

Das Argument für `fd` ist die Distanz in Pixeln, weshalb die eigentliche Größe von Ihrem Display abhängt.

Andere Methoden, die Sie für eine Schildkröte aufrufen können, sind `bk` für rückwärts, `lt` für links und `rt` für rechts. Das Argument für `lt` und `rt` ist ein Winkel in Grad.

Außerdem hält jede Schildkröte einen Stift, der oben oder unten sein kann. Ist der Stift unten, hinterlässt die Schildkröte eine Spur, wenn sie sich bewegt. Die Methoden `pu` und `pd` stehen für »pen up« (Stift hoch) und »pen down« (Stift runter).

Fügen Sie diese Zeilen in das Programm ein, um einen rechten Winkel zu zeichnen (nachdem Sie `bob` erstellt haben und bevor Sie `mainloop` aufrufen):

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

Wenn Sie dieses Programm ausführen, müsste sich `bob` zuerst nach Osten und dann nach Norden bewegen und dabei zwei Linienabschnitte zurücklassen.

Ändern Sie nun das Programm so, dass es ein Quadrat zeichnet. Lassen Sie nicht locker, bis es funktioniert!

Einfache Wiederholung

Höchstwahrscheinlich haben Sie ungefähr Folgendes geschrieben:

```
bob.fd(100)
bob.lt(90)
```

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

Prägnanter können wir dasselbe mit einer for-Anweisung erreichen. Fügen Sie die folgenden Zeilen in *meinpolygon.py* ein und führen Sie das Skript erneut aus:

```
for i in range(4):
    print('Hallo!')
```

Nun sollten Sie in etwa Folgendes sehen:

```
Hallo!
Hallo!
Hallo!
Hallo!
```

Das ist die einfachste Einsatzmöglichkeit einer for-Anweisung. Mehr dazu erfahren Sie später. Das sollte aber bereits ausreichen, damit Sie Ihr Programm zum Zeichnen des Quadrats neu schreiben können. Bleiben Sie so lange dran, bis es funktioniert!

Hier sehen Sie eine for-Anweisung, die ein Quadrat zeichnet:

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

Die Syntax einer for-Anweisung ist einer Funktionsdefinition recht ähnlich. Sie hat einen Header, der mit einem Doppelpunkt endet, sowie einen eingerückten Body. Auch hier kann der Body wieder eine beliebige Anzahl von Anweisungen enthalten.

Eine for-Anweisung wird manchmal auch als **Schleife** bezeichnet, weil das Programm den Body in einer Schleife durchläuft. In diesem Fall wird der Body viermal ausgeführt.

Diese Version unterscheidet sich genau genommen ein klein wenig von dem bisherigen Code, weil sich die Schildkröte nach der letzten Seite des Quadrats noch einmal zusätzlich dreht. Diese Drehung braucht ein wenig mehr Zeit, vereinfacht aber den Code, wenn wir bei jedem Durchlauf durch die Schleife immer dasselbe tun. Außerdem landet die Schildkröte so auch wieder in der ursprünglichen Position und zeigt in die Ausgangsrichtung.

Übungen

Es folgt eine Reihe von Übungen mit TurtleWorld. Sie sollen natürlich Spaß machen, haben aber auch einen Sinn. Denken Sie darüber nach, welcher Sinn das jeweils sein könnte, während Sie an den Übungen arbeiten.

Die folgenden Abschnitte enthalten auch Lösungen für die Übungen. Blättern Sie aber nicht vor, bevor Sie mit den Übungen fertig sind (oder wenigstens versucht haben, sie erfolgreich zu absolvieren).

1. Schreiben Sie eine Funktion mit dem Namen `quadrat`, die eine Schildkröte als Parameter `t` erwartet. Die Funktion soll diese Schildkröte verwenden, um ein Quadrat zu zeichnen.

   Schreiben Sie eine Funktion, die `bob` als Argument an `quadrat` übergibt, und führen Sie das Programm erneut aus.
2. Fügen Sie einen zusätzlichen Parameter mit dem Namen `laenge` in `quadrat` ein. Ändern Sie den Body so, dass die Kantenlänge durch `laenge` bestimmt wird, und ändern Sie den Funktionsaufruf so, dass ein zweites Argument übergeben wird. Führen Sie das Programm erneut aus. Testen Sie es mit verschiedenen Werten für `laenge`.
3. Machen Sie eine Kopie von `quadrat` und ändern Sie den Namen in `polygon`. Fügen Sie einen zusätzlichen Parameter `n` ein und ändern Sie den Body so, dass ein gleichseitiges Polygon mit `n` Seiten gezeichnet wird. Tipp: Die Außenwinkel eines gleichseitigen n-seitigen Polygons betragen 360/n Grad.
4. Schreiben Sie eine Funktion mit dem Namen `kreis`, die eine Schildkröte `t` und einen Radius `r` als Parameter erwartet und einen ungefähren Kreis zeichnet, indem sie `polygon` mit einer entsprechenden Länge und Anzahl von Seiten aufruft. Testen Sie die Funktion mit mehreren Werten für `r`.

   Tipp: Ermitteln Sie den Umfang des Kreises und vergewissern Sie sich, dass `laenge * n = umfang`.

   Noch ein Tipp: Wenn `bob` Ihnen zu langsam ist, können Sie das ändern, indem Sie `bob.delay` anpassen. Dadurch legen Sie die Zeit zwischen den einzelnen Bewegungen in Sekunden fest. Mit `bob.delay = 0.01` wird er die Beine in die Hand nehmen müssen.
5. Schreiben Sie eine allgemeinere Version von `kreis` mit dem Namen `bogen`, die einen zusätzlichen Parameter `winkel` erwartet, mit dem Sie festlegen können, welcher Teil eines Kreises gezeichnet werden soll. `winkel` wird in Grad angegeben, sodass bei `winkel = 360` ein vollständiger Kreis gezeichnet wird.

Datenkapselung

In der ersten Übung sollten Sie den Code zum Zeichnen des Quadrats in eine Funktionsdefinition schreiben und anschließend die Funktion aufrufen, wobei Sie die Schildkröte als Parameter übergeben. Hier eine mögliche Lösung:

```
def quadrat(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

quadrat(bob)
```

Die Anweisungen ganz innen – `fd` und `lt` – wurden zweimal eingerückt, um zu kennzeichnen, dass sie innerhalb der `for`-Schleife stehen, die sich wiederum innerhalb der Funktionsdefinition befindet. Die nächste Zeile, `quadrat(bob)`, ist wieder linksbündig, wodurch sowohl das Ende der `for`-Schleife als auch das der Funktionsdefinition gekennzeichnet werden.

Innerhalb der Funktion bezieht sich `t` auf dieselbe Schildkröte wie `bob`, entsprechend hat `t.lt(90)` denselben Effekt wie `bob.lt(90)`. Aber warum rufen wir dann nicht den Parameter `bob` auf?

Weil `t` auf diese Weise eine beliebige Schildkröte sein kann, nicht nur `bob`. So können Sie auch eine zweite Schildkröte erstellen und als Argument an `quadrat` übergeben:

```
alice=turtle.Turtle()
quadrat(alice)
```

Wenn Sie eine Codezeile in eine Funktion auslagern, nennt man das **Datenkapselung**. Einer der Vorteile der Datenkapselung besteht darin, dass der entsprechende Codeteil einen Namen erhält, was gleichzeitig auch der Dokumentation des Codes dient. Und wenn Sie einen bestimmten Code mehrmals verwenden möchten, ist es wesentlich einfacher, eine Funktion mehrfach aufzurufen, als deren Body mehrmals zu kopieren und einzufügen.

Generalisierung

Der nächste Schritt besteht darin, `quadrat` um den Parameter `laenge` zu erweitern. Hier eine mögliche Lösung:

```
def quadrat(t, laenge):
    for i in range(4):
        t.fd(length)
        t.lt(90)

quadrat(bob, 100)
```

Die Erweiterung einer Funktion um einen Parameter nennt man **Generalisierung**, weil dadurch die Funktion verallgemeinert wird. In der vorherigen Version hatte das Quadrat immer dieselbe Größe. In dieser Version kann es eine beliebige Größe haben.

Der nächste Schritt ist ebenfalls eine Generalisierung. Anstatt Quadrate zu zeichnen, kann `polygon` regelmäßige Polygone mit einer beliebigen Anzahl von Seiten zeichnen. Hier eine mögliche Lösung:

```
def polygon(t, n, laenge):
    winkel = 360.0 / n
    for i in range(n):
        t.fd(laenge)
        t.lt(winkel)

polygon(bob, 7, 70)
```

Dieses Beispiel zeichnet ein siebenseitiges Polygon mit einer Seitenlänge von 70.

Wenn Sie Python 2 nutzen, kann der Wert von `winkel` fehlerhaft sein, weil es sich um eine Integerdivision handelt. Eine einfache Lösung besteht darin, `winkel = 360.0 / n` zu berechnen. Da der Zähler eine Fließkommazahl ist, ist auch das Ergebnis eine Fließkommazahl.

Wenn Sie mehr als ein numerisches Argument übergeben, kann es leicht passieren, dass Sie vergessen, was die einzelnen Argumente bedeuten und in welcher Reihenfolge Sie sie angeben müssen.

Es ist daher zulässig – und manchmal auch durchaus hilfreich –, die Namen der Parameter in der Argumentenliste mit anzugeben:

```
polygon(bob, n=7, laenge=70)
```

Solche Argumente bezeichnet man als **Schlüsselwort-Argumente**, weil sie die Parameternamen als »Schlüsselwörter« mit angeben (nicht zu verwechseln mit Python-Schlüsselwörtern wie `while` und `def`).

Durch diese Syntax ist das Programm besser lesbar. Außerdem veranschaulicht dieses Beispiel, wie Argumente und Parameter funktionieren: Wenn Sie eine Funktion aufrufen, werden die übergebenen Argumente den entsprechenden Parametern zugewiesen.

Gestaltung von Schnittstellen

Im nächsten Schritt zeichnen Sie einen `kreis` mit dem Radius `r` als Parameter. Hier sehen Sie eine einfache Lösung, die mit `polygon` ein fünfzigseitiges Polygon zeichnet:

```
import math

def kreis(t, r):
    umfang = 2 * math.pi * r
    n = 50
    laenge = umfang / n
    polygon(t, n, laenge)
```

In der ersten Zeile wird der Umfang des Kreises mit Radius `r` über die Formel $2\pi r$ berechnet. Da wir `math.pi` verwenden, müssen wir `math` importieren. Der Konvention nach müssen `import`-Anweisungen am Anfang des Skripts stehen.

`n` ist die Anzahl der Liniensegmente für die Annäherung an den Kreis. `laenge` ist die Länge der einzelnen Linien. Entsprechend zeichnet `polygon` ein fünfzigseitiges Polygon als Annäherung an einen Kreis mit Radius `r`.

Eine Begrenzung dieser Lösung liegt darin, dass `n` eine Konstante ist. Für sehr große Kreise sind die Liniensegmente zu lang, und bei sehr kleinen Kreisen verschwenden wir Zeit, indem wir sehr kleine Kreissegmente zeichnen. Eine mögliche Lösung besteht darin, die Funktion zu generalisieren und `n` als Parameter entgegenzunehmen.

Dadurch hätten die Benutzer (wer auch immer `kreis` aufruft) mehr Kontrolle, aber die Schnittstelle wäre dadurch weniger übersichtlich.

Die **Schnittstelle** einer Funktion fasst zusammen, wie sie verwendet wird: Wie heißen die Parameter? Was macht die Funktion? Und was ist der Rückgabewert? Eine Schnittstelle ist dann übersichtlich, wenn sie »so einfach wie möglich, aber nicht einfacher ist« (Einstein).

In diesem Beispiel gehört `r` zur Schnittstelle, weil es den zu zeichnenden Kreis bestimmt. `n` ist dagegen nicht ganz zutreffend, weil es sich mehr auf die Einzelheiten dazu bezieht, wie der Kreis gezeichnet werden soll.

Anstatt die Schnittstelle unübersichtlicher zu machen, wählen wir für `n` besser einen Wert, der vom `umfang` abhängt:

```
def kreis(t, r):
    umfang = 2 * math.pi * r
    n = int(umfang / 3) + 1
    laenge = umfang / n
    polygon(t, n, laenge)
```

Nun entspricht die Anzahl der Segmente (ungefähr) `umfang / 3`, wodurch die Länge jedes Segments (ungefähr) 3 beträgt. Das ist klein genug, damit der Kreis hübsch aussieht, und groß genug, um Kreise beliebiger Größe effizient und angemessen zu zeichnen.

Refactoring

Als ich `kreis` geschrieben habe, konnte ich `polygon` wiederverwenden, weil ein Polygon mit beliebig vielen Seiten eine gute Annäherung an einen Kreis ist. Aber `bogen` ist nicht ganz so kooperativ. Wir können weder `polygon` noch `kreis` verwenden, um einen Bogen zu zeichnen.

Eine Alternative besteht darin, mit einer Kopie von `polygon` zu beginnen und sie in einen `bogen` umzuwandeln. Das Ergebnis könnte folgendermaßen aussehen:

```
def bogen(t, r, winkel):
    bogen_laenge = 2 * math.pi * r * winkel / 360
    n = int(bogen_laenge / 3) + 1
    schritt_laenge = bogen_laenge / n
    schritt_winkel = float(winkel) / n

    for i in range(n):
        t.fd(schritt_laenge)
        t.lt(schritt_winkel)
```

Die zweite Hälfte dieser Funktion sieht wie `polygon` aus, aber wir können `polygon` nicht verwenden, ohne die Schnittstelle zu ändern. Wir könnten zwar `polygon` so verallgemeinern, dass die Funktion einen Winkel als drittes Argument erwartet. Aber dann wäre `polygon` kein passender Name mehr! Verwenden wir lieber die allgemeinere Funktion `polylinie`:

```
def polylinie(t, n, laenge, winkel):
    for i in range(n):
        t.fd(laenge)
        t.lt(winkel)
```

Nun können wir `polygon` und `bogen` so umschreiben, dass sie `polylinie` verwenden:

```
def polygon(t, n, laenge):
    winkel = 360.0 / n
    polylinie(t, n, laenge, winkel)

def bogen(t, r, winkel):
    bogen_laenge = 2 * math.pi * r * winkel / 360
    n = int(bogen_laenge / 3) + 1
    schritt_laenge = bogen_laenge / n
    schritt_winkel = float(winkel) / n
    polylinie(t, n, schritt_laenge, schritt_winkel)
```

Zum Abschluss können wir `kreis` noch so umschreiben, dass die Funktion `bogen` verwendet wird:

```
def kreis(t, r):
    bogen(t, r, 360)
```

Den Vorgang, ein Programm neu zu arrangieren, um Funktionsschnittstellen zu verbessern und die Wiederverwendung von Code zu erleichtern, nennt man **Refactoring**. In diesem Fall haben wir festgestellt, dass `bogen` und `polygon` ähnlichen Code enthalten haben, deshalb haben wir ihn in die Funktion `polylinie` »ausgeklammert«.

Mit einer umsichtigen Vorausplanung hätten wir vielleicht zuerst `polylinie` geschrieben und uns das Refactoring gespart. Aber oft wissen Sie am Anfang eines Projekts nicht genug, um alle Schnittstellen entsprechend zu entwerfen. Sobald Sie mit dem Code angefangen haben, verstehen Sie die Probleme besser. Manchmal ist Refactoring ein Zeichen dafür, dass Sie etwas gelernt haben.

Entwicklungsplan

Ein **Entwicklungsplan** ist ein Verfahren zum Schreiben von Programmen. Die beiden Ansätze, die wir in dieser Fallstudie herangezogen haben, waren »Datenkapselung« und »Generalisierung«. Die Schritte dieses Verfahrens lauten:

1. Beginnen Sie mit einem kleinen Programm ohne Funktionsdefinitionen.
2. Sobald das Programm funktioniert, kapseln Sie es in eine Funktion und geben ihr einen Namen.
3. Generalisieren Sie die Funktion durch entsprechende Parameter.
4. Wiederholen Sie die Schritte 1 bis 3, bis Sie eine Reihe entsprechender Funktionen haben. Kopieren Sie den funktionierenden Code und fügen Sie ihn ein, um sich das erneute Tippen (und das erneute Debugging) zu ersparen.

5. Suchen Sie nach Möglichkeiten, das Programm durch Refactoring zu verbessern. Wenn Sie beispielsweise an mehreren Stellen ähnlichen Code verwenden, sollten Sie darüber nachdenken, diesen in eine entsprechende allgemeinere Funktion auszulagern.

Dieses Verfahren hat auch Nachteile (Alternativen dazu sehen wir uns später an), kann aber sehr nützlich sein, wenn Sie nicht von vornherein wissen, wie Sie das Programm in Funktionen aufteilen können. Bei diesem Ansatz gestalten Sie das Programm immer wieder um, während Sie daran arbeiten.

Docstring

Ein **Docstring** ist ein String am Anfang einer Funktion, der die Schnittstelle erklärt (»doc« steht dabei für Dokumentation). Hier ein Beispiel:

```
def polylinie(t, n, laenge, winkel):
    """Zeichnet n Liniensegmente.
    t: Turtle-Objekt
    n: Anzahl der Liniensegmente
    laenge: Länge der einzelnen Segmente
    winkel: Winkel zwischen den Segmenten in Grad
    """
    for i in range(n):
        t.fd(laenge)
        t.lt(winkel)
```

Dieser Docstring steht in drei Anführungszeichen hintereinander. So etwas bezeichnet man auch als mehrzeiligen String, weil er mehr als eine Zeile umfassen kann.

Das ist kurz und knapp, enthält aber die wesentlichen Informationen für jemanden, der diese Funktion verwenden möchte. Der Docstring erklärt exakt, was die Funktion macht (ohne auf Einzelheiten einzugehen), welche Auswirkungen die jeweiligen Parameter auf das Verhalten der Funktion haben und welcher Typ jeweils erwartet wird (falls das nicht offensichtlich ist).

Diese Art der Dokumentation ist ein wichtiger Teil der Gestaltung von Schnittstellen. Eine gut durchdachte Schnittstelle sollte einfach zu erklären sein. Sollten Sie Schwierigkeiten haben, eine Ihrer Funktionen zu beschreiben, könnte das ein Hinweis darauf sein, dass die Schnittstelle verbesserungsbedürftig ist.

Debugging

Eine Schnittstelle ist wie ein Vertrag zwischen einer Funktion und dem Aufrufenden. Der Aufrufende stimmt zu, bestimmte Parameter zur Verfügung zu stellen, und die Funktion willigt ein, eine bestimmte Aufgabe zu erfüllen.

`polylinie` benötigt beispielsweise vier Argumente: `t` muss eine Turtle sein, `n` ist die Anzahl der Liniensegmente und muss daher ein Integer sein. `laenge` muss eine positive Zahl sein, und `winkel` muss eine Zahl sein, die sich in Grad auswerten lässt.

Diese Anforderungen nennt man **Vorbedingungen**, weil sie erfüllt sein müssen, bevor die Funktion mit der Ausführung beginnen kann.

Die Bedingungen gegen Ende der Funktion heißen entsprechend **Nachbedingungen**. Zu den Nachbedingungen gehören der gewünschte Effekt der Funktion (beispielsweise das Zeichnen von Liniensegmenten) sowie jegliche Nebeneffekte (Bewegungen der Schildkröte oder andere Änderungen in der jeweiligen Welt).

Vorbedingungen unterliegen der Verantwortung des Aufrufenden. Falls der Aufrufende eine (korrekt dokumentierte!) Vorbedingung nicht erfüllt und deshalb die Funktion nicht korrekt arbeitet, liegt der Fehler beim Aufrufenden, nicht bei der Funktion.

Wenn die Vorbedingungen erfüllt sind und die Nachbedingungen nicht, liegt der Fehler in der Funktion.

Glossar

Methode
: Eine mit einem Objekt verknüpfte Methode, die mittels Punktnotation aufgerufen wird

Schleife
: Teil eines Programms, der wiederholt ausgeführt wird

Datenkapselung
: Vorgang, bei dem eine Folge von Anweisungen in eine Funktionsdefinition umgewandelt wird

Generalisierung
: Verfahren, um etwas unnötig Spezifisches (etwa eine Zahl) durch etwas Allgemeineres (etwa eine Variable oder einen Parameter) zu ersetzen

Schlüsselwort-Argument
: Argument, das den Namen des Parameters als »Schlüsselwort« enthält

Schnittstelle
: Beschreibung, wie eine Funktion zu verwenden ist, einschließlich der Namen und Beschreibungen der Argumente sowie des Rückgabewerts

Refactoring
: Vorgang, um die Funktionsschnittstellen und andere Qualitäten eines Programms zu verbessern

Entwicklungsplan
: Verfahren zum Schreiben von Programmen

Docstring
: String in einer Funktionsdefinition, der die Schnittstelle der Funktion dokumentiert

Vorbedingung
: Bedingung, die vom Aufrufenden erfüllt werden muss, bevor eine Funktion ausgeführt werden kann

Nachbedingung
: Anforderung, die von einer Funktion erfüllt werden muss, bevor sie beendet wird

Übungen

Übung 4-1

Den Code für dieses Kapitel finden Sie in der Beispieldatei *polygon.py* (*http://downloads.oreilly.de/9783960091691*).

1. Zeichnen Sie ein Stapeldiagramm, das den Zustand des Programms bei der Ausführung von `kreis(bob, radius)` darstellt. Die Berechnungen können Sie entweder von Hand durchführen oder Sie können entsprechende `print`-Aufrufe in den Code einfügen.
2. Die Version von `bogen` im Abschnitt »Refactoring« auf Seite 57 ist nicht allzu genau, weil die lineare Annäherung an einen Kreis niemals einen echten Kreis ergibt. Als Konsequenz davon landet die Schildkröte einige Einheiten von der korrekten Position entfernt. Meine Lösung zeigt eine Möglichkeit, den Effekt dieser Abweichung zu reduzieren. Lesen Sie den Code und schauen Sie, ob er für Sie Sinn ergibt. Wenn Sie ein Diagramm zeichnen, finden Sie vielleicht heraus, wie er funktioniert.

Übung 4-2

Schreiben Sie eine halbwegs allgemeine Sammlung von Funktionen, die Blumen wie die in Abbildung 4-1 zeichnen können.

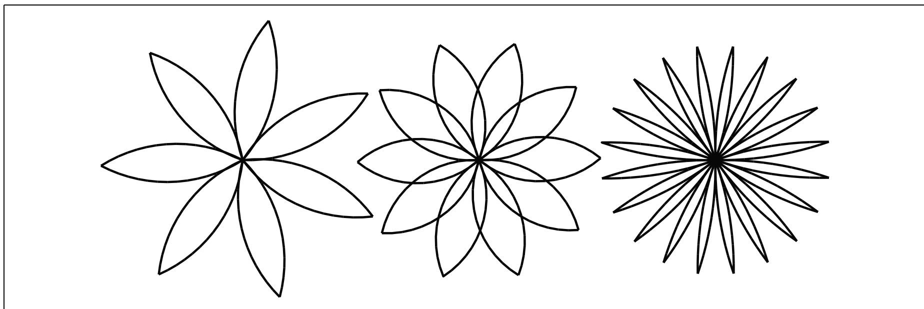

Abbildung 4-1: Turtle-Blumen

Lösung (*https://oreilly.de/9783960091691*): *blumen.py*, benötigt außerdem *polygon.py*

Übung 4-3

Schreiben Sie eine angemessen allgemeine Sammlung von Funktionen, die Formen wie die in Abbildung 4-2 zeichnen kann.

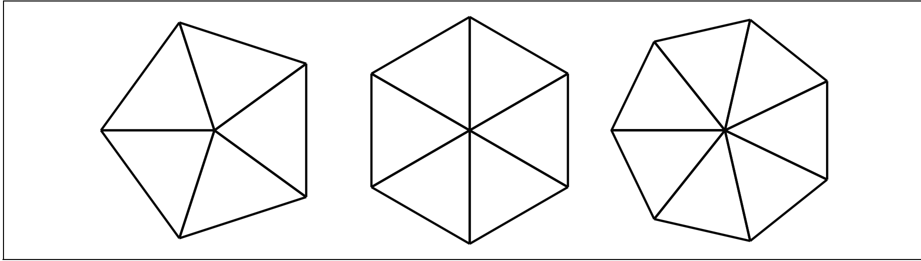

Abbildung 4-2: Turtle-Kuchen

Lösung: *kuchen.py*

Übung 4-4

Die Buchstaben des Alphabets können aus einer überschaubaren Anzahl grundlegender Elemente aufgebaut werden, wie etwa vertikalen und horizontalen Linien sowie einigen Kurven. Gestalten Sie eine Schrift, die mit einer minimalen Anzahl grundlegender Elemente gezeichnet werden kann, und schreiben Sie die Funktionen, die die Buchstaben des Alphabets zeichnen.

Schreiben Sie jeweils eine Funktion für jeden Buchstaben mit den Namen `zeichne_a`, `zeichne_b` usw. und legen Sie die Funktionen in einer Datei mit dem Namen *buchstaben.py* ab. Die Datei *schreibmaschine.py* enthält eine »Schildkrötenschreibmaschine«, mit der Sie Ihre Funktionen testen können.

Lösung: *buchstaben.py*, benötigt außerdem *polygon.py*

Übung 4-5

Informieren Sie sich über Spiralen unter *http://de.wikipedia.org/wiki/Spirale*. Schreiben Sie dann ein Programm, das eine archimedische Spirale zeichnet (oder einen der anderen Typen).

Lösung: *spirale.py*