Einfache Algorithmen mit Python
– Handout zur Vorlesung Diskrete Strukturen –

Bauhaus-Universität Weimar

Stefan Lucks

24. September 2015

1. Einleitung

1.1. Warum Python?

Die Veranstaltung **Diskrete Strukturen** beschäftigt sich mit Strukturen der Diskreten Mathematik, und mit Algorithmen, die über diese operieren. Um Algorithmen nicht nur abstrakt in Pseudocode zu beschreiben, sondern auf einem Computer laufen lassen zu können, verwenden wir die Programmiersprache Python. Die Hauptgründe für den Einsatz von Python sind:

1. Die Python-Syntax, vor allem für Programmieranfänger, sehr einfach zu verstehen.[1]

2. Die Python-Syntax ist auch für Programmieranfänger sehr einfach zu schreiben.

Weitere Gründe sind, dass Python eine verbreitete Programmiersprache, gut dokumentiert und kostenlos auf allen gängigen Systemen installierbar ist. Außerdem hat, wer

[1] Wie in anderen Programmiersprachen, braucht man auch in Python manchmal Phrasen, die für Programmieranfänger unverständlich sind, quasi "magisch". Das fachdidaktische Konzept dieses Handouts – wie das der Veranstaltung Diskrete Strukturen insgesamt – zielt darauf ab, Dinge *verständlich* zu machen, und auf "magische Aspekte", die erst für Fortgeschrittene verständlich werden, so weit wie möglich zu verzichten. Einer der Vorteile von Python ist, dass man derarige Phrasen nicht für die ersten Programmierschritte benötigt.

So würde man eine Unterprogrammdefinition als Beispiel für Anfänger in Java mit **public static void** einleiten. Für Anfänger könnte man statt "public static" auch "abracadabra" schreiben. Der Sinn von "public" und "static" erschließt sich erst Fortgeschrittenen. In Python leitet man eine Unterprogrammdefinition mit **def** ein, was unschwer als Abkürzung von "define" erkennbar ist.

Python kennt, jederzeit ein nützliches Werkzeug um mathematische Aussagen nachzurechnen, Vermutungen zu überprüfen oder Zufallsprozesse zu simulieren.

Schließlich haben die meisten Programmiersprachen zur Repräsentation ganzer Zahlen einen Typ **int** (bzw. **Integer** o. ä.), welcher tatsächlich nur ganze Zahlen bis zu einer bestimmten Größe darstellen kann (oft 2^{32}, oder 2^{64}) und Fließkommazahlen, die zwar größere Werte repräsentieren können, aber nur näherungsweise, nicht genau. Für die Veranstaltung Diskrete Strukturen ist Python besser geeignet, weil die Darstellung "mathematischer" ganzer Zahlen ein fester Teil der Sprache ist (nicht etwa Teil einer ergänzenden Bibliothek). Zum Beispiel ist 2^{200} (exakt, nicht bloß näherungsweise):

```
>>> 2**200
1606938044258990275541962092341162602522202993782792835301376
```

1.2. Dies ist keine Einführung in die Programmierung mit Python!

Es wird nur beschrieben, wie man *einfache Algorithmen in Python implementiert* und einfache Probleme mit Python löst. Mehr wird für die Veranstaltung Diskrete Strukturen nicht gebraucht! Wer mit Python komplexe Probleme lösen will, oder sich überhaupt intensiver mit dieser Programmiersprache auseinandersetzen will, sollte auf ein ordentliches Lehrbuch zurückgreifen – ein Handout wie dieses ist allemal zu kurz. Es gibt viele gute Lehrbücher. ich empfehle

Practical Programming: An Introduction to Computer Science Using Python 3

von Paul Gries, Jennifer Campbell und Jason Montojo.[2]

Insbesondere verzichte ich darauf, jene Aspekte von Python zu vermitteln, die trotz der einfachen Syntax sehr wohl kompliziert sind, zum Beispiel die Unterscheidung zwischen veränderlichen ("mutable") und unveränderlichen Objekten – bzw. das Speichermodell, das der Python-Semantik zugrunde liegt. Die Unterstützung von objektorientierter Programmierung und Vererbung wird nicht beschrieben. Python-Listen werden zwar erwähnt, aber nur mit Bezug auf eine Nutzung analog zu *Arrays* anderer Programmiersprachen usw. Insgesamt wäre eine Liste der Features von Python die in diesem Handout *nicht* beschrieben werden länger als das Handout selbst.

[2] Siehe `https://archive.org/details/PracticalProgramming`.

1.3. Aufgaben (diesen Text sollten Sie nicht nur lesen!)

Benutzen Sie eine Python-Shell[3] und schreiben Sie Python-Programme![4]

In Abschnitt 1.1 haben Sie bereits gesehen, wie die Interaktion in einer Python-Shell in diesem Handout dargestellt wird. Wir haben interaktiv 2^{200} berechnet und das exakte Ergebnis ausgegeben. IDLE (und andere Python-Shells) geben "`>>> `" (drei "größer"-Zeichen, danach ein Leerzeichen) als "Eingabeprompt" aus, danach die *Eingabe* des Benutzers, in diesem Fall "`2**200`" für 2^{200}. In der nächsten Zeile, ohne Eingabeprompt, steht die *Ausgabe* von Python ("160 . . . 376").

Listings von Programmdateien in Python werden in diesem Handout durch "Listing . . . " (oben) und Zeilennummern (links) kenntlich gemacht. Oft wird eine Datei stückweise in mehreren Fragmenten dargestellt. Die vollständigen Dateien können sie auch auf der Webseite zur Veranstaltung finden. Aber grundsätzlich können Sie die Dateien anhand der Zeilennummern auch aus den hier dargestellten Fragmenten vollständig zusammensetzen. Anhand der Zeilennummern werden Sie feststellen, dass einzelne Zeilen zu fehlen scheinen – dies sind allerdings grundsätzlich nur Leerzeilen. Diese dienen dazu, inhaltlich zusammenhängende Code-Fragmente von anderen Code-Fragmenten abzugrenzen.

1. Rufen Sie Python auf und berechnen Sie 2^{100}, sowie die Summe $2^{100} + 2^{200}$.

2. Schreiben Sie eine Python-Datei "hallo.py" mit dem folgenden Programm:

Listing 1: Zeilen 1–3 aus hallo.py

```
1 def hallo():
2     """Prints some greetings."""
3     print("Hallo Universum!")
```

3. Führen Sie Ihr Programm in Ihrer Python-Shell aus:

```
>>> import hallo
>>> hallo.hallo()
Hallo Universum!
```

4. Was wird angezeigt, wenn Sie die Hilfefunktion von Python aufrufen?

```
>>> help(hallo)
```

[3] Ich selbst verwende meistens die IDLE Umgebung ("Integrated DeveLopment Environment").

[4] Mit IDLE: Im "File"-Menü Unterpunkt "New File", "Open . . . " oder ggf. "Recent File" aufrufen, in dem Fenster, das sich neu öffnet schreiben, und dann dort das "File"-Menü aufrufen um das Ergebnis mit "Save" oder "Save as . . . " zu sichern.

2. Python als “Taschenrechner”

Wie erwähnt kann man mit Python nicht nur Programme schreiben, man kann auch interaktiv damit arbeiten – in etwa wie mit einem (extrem leistungsfähigen) Taschenrechner.

Wenn wir Python aufrufen, werden wir von Python begrüßt, etwa so:

```
Python 3.4.3+ (default, Jun  2 2015, 14:09:35)
[GCC 4.9.2] on linux
Type "copyright", "credits" or "license()" for more information.
```

Wie Sie sehen, arbeite ich zum Zeitpunkt an dem dieses Handout entsteht mit Python 3.4.3. Sie können mit jeder Version 3.x.y arbeiten (kurz: mit Python 3). Die Vorgänger-Version Python 2 ist immer noch weit verbreitet, aber viele der in dieser Einleitung präsentieren Beispiele müssten für Python 2 umgeschrieben werden.

Beispiel: *Wieviele Sekunden hat ein Jahr?*

```
>>> jahre = 1
>>> tage = 365*jahre
>>> stunden = 24 * tage
>>> minuten = 60 * stunden
>>> sekunden = 60 * minuten
>>> sekunden
31536000
```

Das war einfach! *Nun suchen wir die Quadratwurzel von 3343.* Genauer, wir suchen die *größte ganze Zahl*, die kleiner oder gleich der Quadratwurzel von 3343 ist.

```
>>> 40*40
1600
>>> 50*50
2500
>>> 60*60
3600
```

Die gesuchte Zahl liegt demzufolge irgendwo zwischen 50 und 60. Im nächsten Schritt könnten wir, z. B., 55 probieren ...

Geht es auch bequemer? Wozu haben wir einen Computer – und Python?

```
>>> x = 1
```

```
>>> while x*x < 3343:
        x = x + 1

>>> print(x)
58
```

Was haben wir hier gemacht?

- Wir haben eine Variable x auf 1 gesetzt.
- Wir haben eine Schleife begonnen, die so lange durchlaufen wird, bis $x^2 > 3343$ ist.
- In jedem Schleifendurchlauf haben wir x um eins erhöht.
- Dann lassen wir die Schleife laufen (leere Eingabe von uns).
- Am Ende geben wir x aus.

Das Ergebnis ist *fast* richtig:

```
>>> print(x*x)
3364
>>> print((x-1)*(x-1))
3249
```

Wir haben die kleinste ganze Zahl, die größer als die gesuchte Quadratwurzel ist, denn wir berechnen so lange x = x + 1 bis die Variable x *zu groß* ist. Also machen wir das letzte x = x + 1 rückgängig:

```
>>> print(x-1)
57
```

Das ist das Ergebnis, das wir gesucht haben!

Um mehrere Quadratwurzeln zu berechnen, möchten wir nicht immer wieder x = 1 und **while** x*x < ...: usw. eingeben. Deshalb ist es sinnvoll ein *Programm* (genauer, eine Funktion) die Quadratwurzeln berechnen kann zu definieren.

```
>>> def sqrt(y):
        x = 1
        while x*x < y:
            x = x + 1
        return x-1
```

```
>>> sqrt(23423423)
4839
```

Stimmt das Ergebnis eigentlich? Berechnet der Algorithmus, den wir geschrieben haben, die richtigen Werte?

Wir verzichten hier auf den Versuch, die Korrektheit unseres Programms formal zu beweisen. Aber um mögliche Fehler zu finden, kann man Programme *testen*.[5]

Ein *Testfall* besteht, im einfachsten Fall, aus

- einer Eingabe für das Programm
- **und** der erwarteten Ausgabe.[6]

Man sollte die Testfälle so wählen, dass *alle Klassen typischer Probleme* und dazu *alle Grenzfälle* (mindestens) einmal getestet werden. Für sqrt sind das

- die Eingabe 0 als Grenzfall, erwartetes Ergebnis 0
- drei Klassen von typischen Eingaben:
  - Eingaben, die etwas kleiner als eine Quadratzahl sind
    (wir wählen 3343*3343-1, erwartetes Ergebnis 3342),
  - Eingaben, die eine Quadratzahl sind
    (wir wählen 3343*3343, erwartetes Ergebnis 3343), und
  - Eingaben, die etwas größer als eine Quadratzahl sind
    (wir wählen 3343*3343+1, erwartetes Ergebnis 3343)

```
>>> sqrt(0)
0
>>> sqrt(3343*3343-1)
3342
>>> sqrt(3343*3343)
3342
>>> sqrt(3343*3343+1)
3343
```

[5] Mit *Testen* kann man zwar nicht nachweisen, dass ein Programm richtig ist, aber ein erfolgreicher Test beweist, dass es falsch ist. Schlägt ein systematischer Test fehl, ist das Programm vielleicht korrekt.
[6] Es ist wichtig, dass man sich *vor dem Test* gehau überlegt, welche Ergebnisse richtig sind.

Hurra! Unser Test ist erfolgreich![7] Die Quadratwurzel von 3343 * 3343 ist 3343, nicht 3342. Tatsächlich liefert unser Programm *immer* falsche Werte, wenn y eine Quadratzahl ist. Denn die Schleife bricht schon ab, wenn $x * y$ gleich y ist, das richtige Ergebnis wäre x, aber unser Unterprogramm gibt $x - 1$ zurück.

Wir schreiben also nun **while** x∗x <= y: statt **while** x∗x < y:

```
>>> def sqrt(y):
        x = 1
        while x*x <= y:
            x = x + 1
        return x-1
```

Nun rufen wir unsere Testfälle noch einmal auf.

```
>>> sqrt(0)
0
>>> sqrt(3343*3343-1)
3342
>>> sqrt(3343*3343)
3343
>>> sqrt(3343*3343+1)
3343
```

Diesmal schlägt der Test fehl – das Programm scheint korrekt zu sein.

Einige Elemente von Python:

- *Variablen*, die Namen haben (sogenannte "Bezeichner") jahre, tage, ... x,
- die *Zuweisung* von Werten an Variablen: jahre = 1, tage = 365∗Jahre, ... x = 1, x = x + 1
- *arithmetische Ausdrücke*: 365∗Jahre, ..., x∗x, x+1,
- *logische Ausdrücke*: (x∗x < 3343),
- eine *Schleife*, die mit dem *Schlüsselwort* **while** beginnt und
- der Aufruf eines *vordefinierten Unterprogramms* print.

[7] Das bedeutet natürlich, dass unser Programm nachgewiesenermaßen fehlerhaft ist.

Bezeichner sind Namen für in einem Programm auftretende Dinge, unter anderem für Variablen und Unterprogramme, bestehen aus Groß- und Kleinbuchstaben, dem Unterstrich, und Ziffern. Eine Variable kann nicht mit einer Ziffer beginnen:

- Mögliche Bezeichner sind “name”, “Name”, “ein_name”, “name1”, “ name_1”, ...
- Nicht möglich sind “na me”, “1_name”, “na?me”.

Eine *Zuweisung* wie z = a + 2 ∗ b besteht aus

- einem Variablennamen (hier z),
- gefolgt von einem Gleichheitszeichen (hier =),
- gefolgt von einem Ausdruck (hier a + 2 ∗ b).

Bei der Zuweisung wird zuerst der Wert des Ausdrucks berechnet, dann wird die Variable auf diesen Wert gesetzt. Der Variablenname muss nicht zuvor bekannt sein (d. h., Python braucht keine “Variablendeklarationen”, wie andere Programmiersprachen). Hat die Variable bereits einen Wert, kann sie im Ausdruck verwendet werden, wie in x=x+1.

Aus den Bezeichnern, den Operatoren für die Grundrechenarten und Klammern kann man *arithmetische Ausdrücke* konstruieren. Man beachte, dass es zwei verschiedene Divisionen gibt: “/” für die Division von Fließkommazahlen (Näherungen von reellen Zahlen) und “//” für die ganzzahlige Division ohne Rest. Den Rest der ganzzahligen Division erhält man mit “%”. Zum Beispiel ist 3343 geteilt durch 57 gleich 58, Rest 37:

```
>>> 3343 / 57
58.64912280701754
>>> 3343 // 57
58
>>> 3343 % 57
37
>>> 57 * 58 + 37
3343
>>> (57 * 58) + 37
3343
>>> 57 * (58 + 37)
5415
```

Wie man sieht, werden gemischte Ausdrücke wie in der Mathematik üblich ausgewertet (“Punktrechnung vor Strichrechnung”, ...)

Logische Ausdrücke erlauben das Vergleichen von Werten. Man beachte, dass das Gleich-

heitsszeichen nicht seine mathematische Bedeutung hat; es kennzeichnet ja Zuweisungen. Für den Test auf Gleicheit dient das doppelte Gleichheitszeichen[8] "==":

```
>>> x = 3
>>> y = 2
>>> print(x > y, x >= y, x = y, x <= y, x < y)
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>> print(x > y, x >= y, x == y, x <= y, x < y)
True True False False False
>>> x = 2
>>> print(x > x, x >= x, x == x, x <= x, x < x)
False True True True False
```

Komplexe logische Ausdrücke kann man mit **and**, **or**, **not** und Klammern realisieren:

```
>>> x = 3
>>> y = 2
>>> z = x > y or (x == y and x < y)
>>> print(z)
True
>>> print((not z) or (x < y))
False
```

Eine **while**-Schleife besteht

- aus dem Schlüsselwort **while**,
- gefolgt von einem logischen Ausdruck, der wahr (**True**) falsch (**False**) sein kann,
- gefolgt von einem Doppelpunkt,
- gefolgt von dem Schleifenrumpf, der aus beliebig vielen Zeilen bestehen kann.

Eine **while**-Schleife kann beliebig oft ausgeführt werden – auch gar nicht (also null-mal). Genauer gesagt, die Schleife wird so lange durchlaufen, bis der logische Ausdruck hinter **while** falsch (**False**) ist.

Dagegen steht bei der **for**-Schleife sofort fest, wie oft sie durchlaufen wird: Nach

for Variable **in** range(Start, Ziel):

[8] Das ist eine häufige Fehlerquelle, die Python leider mit vielen anderen Programmiersprachen teilt.

wird der Schleifenrumpf (Start-Ziel)-mal durchlaufen; range(Ziel) ist eine Kurzform für range(0, Ziel) Dabei nimmt Variable die Werte Start, Start+1, ..., Ziel-1 an.[9]

```
>>> for i in range (2, 4):
        print(i)

2
3
>>> for i in range(4, 4):
        print(i)

>>> for i in range (4):
        print(i)

0
1
2
3
```

Mit dem *Unterprogramm* print können wir unsere Ergebnisse ausgeben. Der Bezeichner print ist *vordefiniert. Vordefinierte Bezeichner* kann man in Python neues Objekt zuweisen. Zum Beispiel kann man print eine Zahl zuweisen. Das ist aber eine schlechte Idee. Erstens wird macht man es sich selbst und anderen schwierig, das Programm irgendwann zu lesen und seine Funktion zu verstehen, zweitens hat jenes Unterprogramm zum Ausgeben von Werten keinen Namen mehr. Wie kann man es dann aufrufen?

```
>>> print(3)
3
>>> print = 3
>>> print
3
>>> print(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Einige Schlüsselwörter wie **while** sind reserviert, und als Bezeichner verboten:

```
>>> while = 3
  File "<stdin>", line 1
    while = 3
          ^
```

[9] Kein Tippfehler! Die Variable durchläuft ein halboffenes Intervall. Der Start ist im Intervall enthalten, das Ziel nicht. Das ist eine andere Fehlerquelle, die Python mit vielen Programmiersprachen teilt.

```
SyntaxError: invalid syntax
```

In diesem Handout werden die reservierten Schlüsselwörter **fett** dargestellt.

Es gibt nur wenige reservierte Schlüsselwörter. Ihre Liste erhält man mit der Hilfe-Funktion von Python. Man ruft zuerst das vordefinierte Unterprogramm `help()` auf und gibt dann `keywords` ein:

```
help> keywords

Here is a list of the Python keywords.
Enter any keyword to get more help.

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

Aufgaben:

1. Das Unterprogramm cqrt ("cube-root") berechne die dritte Wurzel (Kubikwurzel), abgerundet zur nächsten ganzen Zahl. Geben Sie vier geeignete Testfälle an.

2. Schreiben Sie ein das Unterprogrammm cqrt. Überzeugen Sie sich davon, dass es Ihre Tests besteht.

3. Benutzen Sie das Unterprogramm, um die Kubikwurzel von 3343 zu berechnen.

3. Ein Python-Modul

Natürlich wollen wir unsere Funktion sqrt nicht jedesmal neu eintippen wenn wir eine Quadratwurzel zu berechnen haben. Stattdessen öffnen wir eine Datei `sqrt.py` und schreiben unsere Funktion dort auf.

Listing 2: Zeilen 1–11 aus sqrt.py

```
1  """module sqrt: Functions to compute the square-root."""
2
3  def sqrt(y):
4      """
5      Computes the largest integer x with x*x <= y.
6      Requires y to be an integer >= 0.
7      """
8      x = 1
9      while x*x <= y:  # previous version with x*x < y was buggy!
10         x = x + 1
11     return x-1
```

Die Texte zwischen den dreifachen Anführungszeichen sind *Docstrings*. Der Docstring in der ersten Zeile dokumentiert, was das Modul als Ganzes leistet. Der Docstring in Zeile 4–6 die Leistung des Unterprogramms sqrt. Speziell sollte man angeben, was eine Funktion für den Aufrufer leistet ("computes ..."), und welche Vorbedingung beim Aufruf der Funktion beachtet werden muss ("requires ..."). Texte, die mit einer Raute ("#") beginnen, werden bis zum Zeilenende von Python ignoriert. Es sind Kommentare.[10]

Ein Modul kann man mit **import** laden. Dann kann man die im Modul definierten Funktionen aufrufen – oder sich auch den Inhalt des Docstrings anzeigen lassen. Für uns bedeutet dies, dass wir nun jederzeit das Modul sqrt importieren und durch den Aufruf von sqrt.sqrt(n) die (abgerundete) Quadratwurzel einer Zahl n berechnen können.

```
>>> import sqrt
>>> help(sqrt.sqrt)
doc
Help on function sqrt in module sqrt:

sqrt(y)
    computes the largest integer x with x*x <= y
    requires y to be an integer >= 0

>>> sqrt.sqrt(3343)
57
```

Die Dopplung sqrt.sqrt ergibt sich daraus, dass Modul und Funktion beide den gleichen Namen haben. Rufen Sie zum Vergleich einmal help(sqrt) auf!

[10] Kommentare sind ein wichiger Teil der Programmdokumentation, obwohl sie vom Python-Übersetzer ignoriert werden, also semantisch wirkungslos sind. Unkommentierte Programme sind selten verständlich, und es ist oft schwierig, Fehler zu beseitigen. Allerdings sind Kommentare wie Medikamente: "zu viel" kann genau so schaden wie "zu wenig".

Natürlich wollen wir sqrt.sqrt auch testen – das sollten wir jetzt und nach jeder Änderung tun. Tests sollten umfassend und ohne großen Arbeitsaufwand wiederholbar sein. Deshalb sollte man nach Möglichkeit die Testfälle (Eingaben und erwartete Ausgaben) *einmal* festlegen und dann automatisch vom Computer durchführen lassen.

Python hat eine elegante Möglichkeit, das Testen zu automatisieren: Das Modul doctest. Wir ergänzen unser Modul sqrt wie folgt:

Listing 3: Zeilen 13–24 aus sqrt.py

```
13 """
14 >>> import sqrt
15 >>> sqrt.sqrt(0)
16 0
17 >>> sqrt.sqrt(3343*3343)
18 3343
19 >>> sqrt.sqrt(3343*3343-1)
20 3342
21 >>> sqrt.sqrt(3343*3343+1)
22 3343
23 >>>
24 """
```

Das sieht aus wie eine interaktive Python-Sitzung mit unseren Tests. Tatsächlich steht es aber im Programm. Man achte auf die drei Anführungszeichen vor und nach den Testfällen! Nun benutzen wir die Funktion doctest.testfile um unser Modul zu testen. Im Prinzip simuliert doctest die interaktive Python-Sitzung, die wir beim manuellen Testen hätten, und vergleicht ob bei den (durch den Eingabeprompt >>> gekennzeichneten) Eingaben genau die erwarteten Ausgaben erzeugt werden. Deshalb ist auch der leere Eingabeprompt in Zeile 23 wichtig. Ohne den Eingabepromt wüsste doctest nicht, in welcher Zeile die Antwort auf >>> sqrt.sqrt (3343*3343+1) in Zeile 22 endet. Es würde irrtümlich einen Fehler melden.

```
>>> import doctest
>>> doctest.testfile("sqrt.py")
TestResults(failed=0, attempted=5)
```

Die Antwort ist, dass fünf Tests durchgeführt wurden, und dass kein Fehler aufgetreten ist.[11] Funktionieren unsere Tests überhaut? Am einfachsten findet man das heraus, indem man doctest mit einem fehlerhaften Programm testet. Deshalb führen wir den Fehler in Zeile 8 wieder ein:

[11] "Fünf Testfälle? Wir haben doch nur vier Testfälle spezifiziert!" Stimmt, aber doctest betrachtet die Eingabe ">>> **import** sqrt" auch als Teil eines Testfalls, bei dem das erwartete Ergebnis darin besteht, dass nichts ausgegeben wird.

```
8       while x*x < y: # ERROR! Change this back to x*x <= y!!!
```

Nach dieser Manipulation rufen wir noch einmal doctest. testfile auf:

```
>>> import doctest
>>> doctest.testfile("sqrt.py")
**********************************************************************
File "./sqrt.py", line 16, in sqrt.py
Failed example:
    sqrt.sqrt(3343*3343)
Expected:
    3343
Got:
    3342
**********************************************************************
1 items had failures:
   1 of   5 in sqrt.py
***Test Failed*** 1 failures.
TestResults(failed=1, attempted=5)
```

Hurra! Der Test findet den Fehler! Bevor wir weitermachen, berichtigen wir den Fehler. Dann rufen wir wieder doctest. testfile , auf, um uns davon zu überzeugen, dass wir den Fehler wieder losgeworden sind und keinen neuen Fehler eingeführt haben (ganz wichtig!).

Module in Python:

- Unterprogramme, die wir wiederverwenden wollen, können in *Modulen* gesammelt werden – typischerweise in einer Datei "⟨Modulname⟩.py".
- Um die *Unterprogramme in einem Modul* zu nutzen, muss man das Modul zuerst mit **import** ⟨Modulname⟩ importieren, dann kann ein Unterprogramm unter dem Namen ⟨Modulname⟩.⟨Name⟩ aufrufen.
- Nicht nur Python, sondern auch Menschen müssen Programme/Module lesen und verstehen können. Sparsam eingesetzte *Kommentare* sind dabei wichtig. In Python beginnen Kommentare mit eine Raute ("#") und enden am Zeilenende.
- Die Funktionsweise von Modulen und Unterprogrammen sollte man durch *Docstrings* dokumentieren. Diese sollten insbesondere beschreiben

  - was ein Unterprogramm anbietet (wozu man es aufrufen soll) und
  - was es verlangt (welche Vorbedingungen erfüllt sein sollen).

- Gut gepflegte Docstrings sorgen unter anderem dafür, dass man beim Aufruf der *Hilfefunktion* help die Information bekommt die man benötigt um ein Modul bzw. Unterprogramm zu nutzen.

- Für das automatische Testen eines Moduls kann man die Testfälle in die Datei "⟨Modulname⟩.py", in einer Syntax, die einer interaktiven Python-Sitzung enstspricht, schreiben.

- Für den eigentlichen Test ruft man doctest. testfile auf.

Aufgaben:

1. Schreiben Sie ein Modul cqrt mit Ihrem Unterprogramm cqrt.cqrt zur Berechnung von Kubikwurzeln. Testen Sie Ihr Modul mit doctest.

2. Bauen Sie einen Fehler in das Modul ein.

3. Überpüfen Sie, ob doctest den Fehler findet.

4. Berichtigen Sie den Fehler, und vergewissern Sie sich dann, dass doctest keinen Fehler mehr findet.

4. Zufallsexperimente mit Python

Wenn man mit einem fairen Würfel 60-mal würfelt, wie oft hat man dann eine 6 gewürfelt? Wer spontan "10-mal" antwortet, hat zwar *im Durchschnitt* recht – aber wenn man 9-mal oder 11-mal eine Sechs würfelt, sollte man auch nicht überrascht sein.

Angenommen, wir würfeln n-mal. Statistisch erwarten wir $\mu = n/6$ 6-en. Sei $x \in \{0, \ldots, n\}$ die Anzahl an 6-en, die wir bei einem Experiment tatsächlich erwürfeln. Wir interessieren uns für die folgenden Fragen:

1. Wahrscheinlicheit des Erwartungswertes: Wie wahrscheinlich ist $x = \mu$?

2. Absolute Abweichung: Wie wahrscheinlich ist $\mu - \delta \leq x \leq \mu + \delta$ für $\delta \geq 1$?

3. Relative Abweichung: Wie wahrscheinlich ist $\mu - c\mu \leq x \leq \mu + c\mu$ für $c > 0$?

Natürlich kann man diese Fragen mathematisch beantworten. Aber wir werden die Antwort einfach "auswürfeln". Besser noch: Wir lassen Python die Arbeit tun! Um Zufallsereignisse zu simulieren, brauchen wir einen Zufallsgenerator. Dazu nutzen wir die Funktion randrange aus dem Paket random. Ganzzahlige Zufallswerte zwischen a und b (einschließlich) erhalten wir durch den Aufruf von random.randrange(a, b+1)[12]:

Listing 4: Zeilen 1–8 aus dice.py

```
1 import random
2
3 def throw():
4     """
5     Simulates a die throw.
6     Returns a random value in {1,2,3,4,5,6}.
7     """
8     return random.randrange(1,7) # uniformly from {1,2,3,4,5,6}
```

```
>>> import dice
>>> dice.throw()
4
>>> dice.throw()
1
>>> dice.throw()
4
>>> dice.throw()
6
>>> dice.throw()
5
>>> dice.throw()
4
```

Wir haben 6-mal gewürfelt und – wie statistisch zu erwarten – genau eine 6 gewürfelt. Nun lassen wir unseren Python öfter würfeln. Python übernimmt dabei gleich das Zählen wie oft eine 6 gewürfelt wurde:

Listing 5: Zeilen 10–16 aus dice.py

```
10 def count_six(r):
11     """Throws a die r times; counts how often a six is thrown."""
12     result = 0
13     for throws in range(r):
```

[12] Analog zu range ist die Zufallszahl größer oder gleich a, aber echt kleiner als $b + 1$.

```
14          if throw() == 6:
15              result = result + 1
16      return result
```

Man beachte die **if**-Kontrollstuktur: Nur wenn das Ergebnis von throw() gleich 6 ist, wird der Wert der Variablen result um 1 erhöht, sonst wird er nicht verändert.

Was für Ergebnisse liefert uns Python nun?

```
>>> dice.count_six(6)
0
>>> dice.count_six(600)
101
>>> dice.count_six(60000)
9951
>>> dice.count_six(6000000)
999806
```

Wie man sieht, ist die Anzahl an 6-en nahe an dem statistisch zu erwartenden Ergebnis, aber bei keinem unserer Experimente hat sie exakt diesen Wert.

Nun wiederholen wir das Experiment 1000-mal. Das Ergebnis ist eine *Liste*:

Listing 6: Zeilen 18–29 aus dice.py

```
18  def experiment(n):
19      """
20      Counts how often count_six(n) gives a specific value;
21      this is repeated 1000 times;
22      experiment(n) returns a list 'results' of length n+1
23      'results[x]' is the number of times count_six(n) returned x.
24      """
25      results = [0]*(n+1)
26      for repetitions in range(1000):
27          x = count_six(n)
28          results[x] = results[x] + 1
29      return results
```

Etwas vereinfacht leistet das Unterprogramm experiment das Folgende:

- Die Zuweisung results = [0] * (n+1) in Zeile 25 legt eine Liste aus $n + 1$ Werten an, auf die man wie auf Variablen unter den Namen results[0], ..., results[n] zugreifen kann. Diese Werte sind alle auf Null gesetzt.

- Zeile 26–28: Insgesamt wird count_six 1000-mal aufgerufen. Für jedes der 1000 Ergebnisse x wird der Wert results[x] um eins vergrößert.

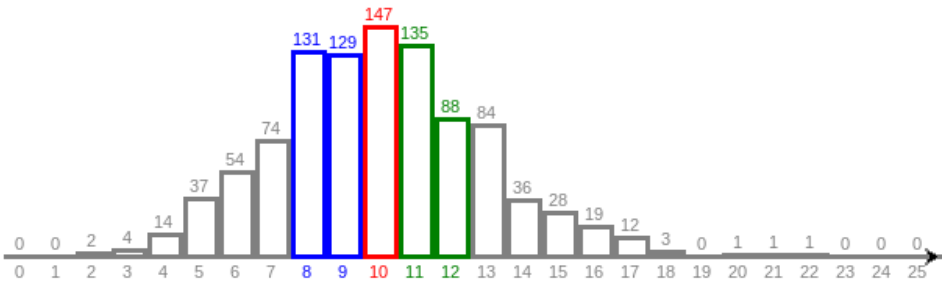
- In Zeile 29 wird diese Liste results Ergebnis zurückgegeben.

Probieren wir aus, was passiert:

```
>>> dice.experiment(60)
[0, 0, 2, 4, 14, 37, 54, 74, 131, 129, 147, 135, 88, 84, 36, 28,
19, 12, 3, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0]
```

Also galt am Ende des Experimentes results[0]==0 results[1]==0 results[2]==2, ..., results[10]==147, ... Das bedeutet, bei keinem der 1000 Versuche haben wir nur eine oder keine 6 gewürfelt, bei immerin zwei von 1000 Versuchen hatten wir genau zwei 6-en, ..., in 14.7 % der Versuche hatten wir genau den Erwartungswert von 10 6-en, ...[13] Insgesamt die Darstellung des Ergebnisses etwas unübersichtlich. Besser kann man sie als Histogramm darstellen, siehe Abbildung 1.

Abbildung 1: Die Ergebnisse des Zufallsexperimentes nach 1000 Wiederholungen: Wie oft wirft man bei 60 Würfen mit einem fairen Würfel eine 6?



Im nächsten Kapitel werden wir sehen, wie man mit Python sehr einfach ein Paket entwickeln kann um derartige Histogramme zu erzeugen.

Zunächst wollen wir aber noch die zu Beginn dieses Kapitels gestellten Fragen beantworten. Dazu definieren wir eine spezielle Ausgabeprozedur eval_list_of_results im Modul dice. Anders als die Unterprogramme, die wir bisher definiert haben, gibt es keinen Befehl **return** ⟨Ergebnis⟩. Vielmehr ruft eval_list_of_results die interne Funktion output auf, die ihrerseits mit print Ergebnisse ausgibt:

[13] Sie werden bei diesem Experiment so gut wie sicher etwas andere Ergebnisse bekommen als die hier dargestellten – ebenso wie ich selbst bei einer Wiederholung des Experimentes.

Listing 7: Zeilen 31–56 aus dice.py

```
31 def eval_list_of_results(hist_list, exp):
32     """
33     Given the output hist_list from experiment(n)
34     and the expected value exp,
35     this procedure prints how many outcomes from count_six are
36       1. equal to exp,
37       2. in range exp plusminus 1, 10,
38       3. in range exp plusminus 10%, 1%, 0.1%.
39     """
40
41     def output(message, hist_list, low, high):
42         sum = 0
43         for i in range(low, high+1):
44             if i >= 0 and i < len(hist_list):
45                 sum = sum + hist_list[i]
46         print(message, ":", sum/10, "%")
47
48     output("1. == exp", hist_list, exp, exp)
49     output("2. exp plus-minus  1  ", hist_list, exp-1, exp+1)
50     output("   exp plus-minus 10  ", hist_list, exp-10, exp+10)
51     output("3. exp plus-minus 10% ", hist_list,
52                  round(exp - exp*0.1), round(exp + exp*0.1))
53     output("   exp plus-minus  1% ", hist_list,
54                  round(exp - exp*0.01), round(exp + exp*0.01))
55     output("   exp plus-minus  0.1%", hist_list,
56                  round(exp - exp*0.001), round(exp + exp*0.001))
```

Nun wollen wir sehen, was usere “Versuchsanordnung” so für Ergebnisse bringt:

```
>>> h2 = dice.experiment(600)
>>> h4 = dice.experiment(60000)
>>> dice.eval_list_of_results(h2, 100)
1. == exp : 4.6 %
2. exp plus-minus  1   : 12.9 %
   exp plus-minus 10   : 72.9 %
3. exp plus-minus 10%  : 72.9 %
   exp plus-minus  1%  : 12.9 %
   exp plus-minus  0.1% : 4.6 %
>>> dice.eval_list_of_results(h4, 10000)
1. == exp : 0.4 %
2. exp plus-minus  1   : 1.5 %
   exp plus-minus 10   : 8.8 %
```

```
3. exp plus-minus 10%   : 100.0 %
   exp plus-minus  1%   : 71.9 %
   exp plus-minus  0.1% : 8.8 %
```

Anhand dieser (und weiterer) Experimente stellen wir die folgenden Hypothesen auf: Je häufiger wir würfeln,

1. um so *kleiner* ist die Wahrscheinlichkeit, *genau* das erwartete Ergebnis zu erzielen,

2. um so *kleiner* ist auch die Wahrscheinlichkeit, nur um einen *absoluten Wert* vom erwarteten Ergebnis abzuweichen, und

3. um so *größer* ist die Wahrscheinlichkeit, nur um einen *relativen Wert* vom erwarteten Ergebnis abzuweichen.

Die dritte Hypothese ist tatsächlich eine Konsequenz aus dem "Gesetz der großen Zahlen". Die ersten beiden Hypothesen widersprechen einer naiven Auslegung dieses Gesetzes.

Elemente von Python:

- Das *Paket* random stellt verschiedene Zufallsgeneratoren zur Verfügung.

  Mit random.randrange(a, $b+1$) zieht man ganzzahlige Zufallswerte aus $\{a, \ldots, b\}$.

- Eine Liste some_list enthält $n+1$ Elemente, die mit some_list[0], some_list[1], ..., some_list[n] referenziert werden können. Diese Elemente verhalten sich analog zu Variablen: results [x] = results [x] + 1.

  Um eine Liste der Länge $n+1$ anzulegen, kann man alle $n+1$ Elemente der Liste auf einen konstanten Wert setzen, wie in results = [0] * (n + 1).

- Die syntaktisch mit **if** dargestellte *bedingte Anweisung* ist eine wichtige Kontrollstruktur. Sie

  - beginnt immer mit "**if** ⟨Bedingung⟩:" und einem Anweisungsblock,

  - es können beliebig viele "**elif** ⟨Bedingung⟩:" mit Anweisungsblöcken folgen ("elif" steht für "else if")

  - und maximal ein "**else**:" mit Anweisungsblock:

```
>>> def three_and_five(x):
```

```
        if x % 15 == 0:
            print("dividable by 3 and by 5")
        elif x % 5 == 0:
            print("dividable by 5")
        elif x % 3 == 0:
            print("dividable by 3")
        else:
            print("relative prime to 15")
```

Diese Kontrollstuktur führt den Anweisungsblock aus, der auf die erste erfüllte Bedingung oder auf **else** folgt, falls keine Bedingung erfüllt ist:

```
>>> three_and_five(3343)
relative prime to 15
>>> three_and_five(3345)
dividable by 3 and by 5
>>> three_and_five(3354)
dividable by 3
>>> three_and_five(3335)
dividable by 5
```

Aufgaben:

1. Wenn Sie mit einer fairen Münze 100-mal werfen, erwarten Sie 50-mal Null ("Kopf") und 50-mal Eins ("Zahl"). Implementieren Sie dieses Zufallsexperiment in Python, analog zu dice.count_six.

2. Wiederholen sie das Zufallsexperiment 1000-mal und geben Sie das Ergebnis als Liste aus, analog zu dice.experiment.

5. Zeichnen mit Python

Es gibt verschiedene wissenschaftliche Pakete für Python, mit denen man Histogramme wie Abb. 1 erzeugen kann. Es ist aber lehrreicher und macht mehr Spaß, selber ein Paket zu entwickeln um Histogramme zu erzeugen.

Um mit Python einfach und verständlich zu "zeichnen", verwenden wir das Paket turtle. Eine "Schildkröte" hat einen Stift, den sie hoch und herunternehmen kann. Die Schildkröte nimmt einfache Befehle entgegen ("gehe 100 Einheiten vorwärts, drehe dich um 90*1.5

Grad nach links, ...”). Ist der Stift unten, zeichnet die Schildkröte den Weg, den sie zurücklegt. Ist der Stift oben, ist der Weg unsichtbar.

Zur Demonstration zeichnen wir das “Haus vom Nikolaus”, siehe Abbildung 2.

Abbildung 2: Das Haus vom Nikolaus mit Turtle-Graphik gezeichnet. Bild 1 ist nach dem Zeichnen von drei Linien entstanden, Bild 2 nach sechs Linien, von denen die letzte leider zu lang ist, Bild 3 nachdem die falsche sechste Linie wieder entfernt wurde, und Bild 4 zeigt das fertige Haus (acht Linien). Die Pfeilspitze bezeichnet Position und Richtung der Turtle.

```
>>> import turtle
>>> turtle.pensize(3)
>>> import math
>>> w2 = math.sqrt(2)
>>> turtle.forward(100)
>>> turtle.left(1.5*90)
>>> turtle.forward(100*w2)
>>> turtle.right(1.5*90)
>>> turtle.forward(100)          ## Bild 1
>>> turtle.right(90*1.5)
>>> turtle.forward(100*w2)
>>> turtle.right(90*1.5)
>>> turtle.forward(100)
>>> turtle.right(90*0.5)
>>> turtle.forward(100*w2)       ## Bild 2
>>> turtle.undo()                ## Bild 3
>>> turtle.forward(50*w2)
>>> turtle.right(90)
>>> turtle.forward(50*w2)
>>> turtle.right(90*0.5)
>>> turtle.forward(100)          ## Bild 4
```

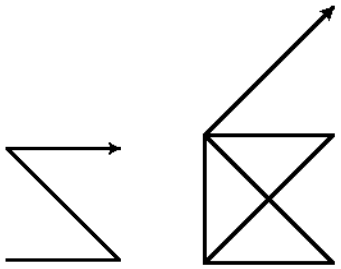


Bild 1 Bild 2

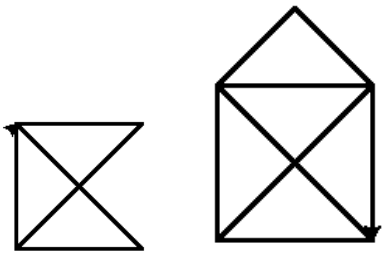


Bild 3 Bild 4

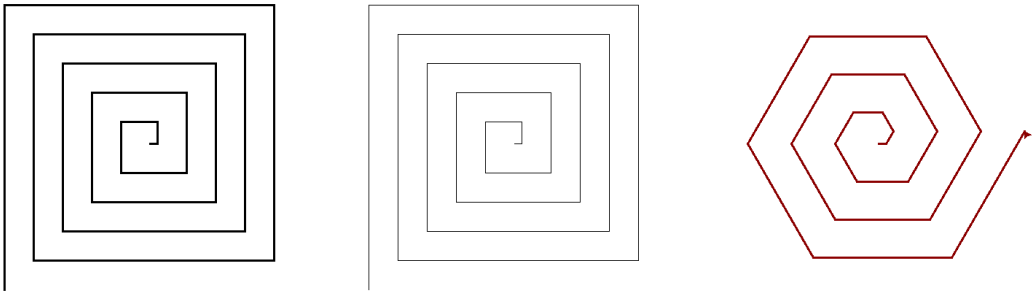
Noch ein zweites Beispiel: Wir definieren ein Unterprogram spiral, um eine Spirale zu zeichnen. Die Spirale besteht aus steps geraden Strichen, der erste ist start_len lang, jeder weitere ist um delta_len länger, und wird um den Winkel delta_angle nach links gedreht:

```
>>> def spiral(start_len, delta_len, delta_angle, steps):
        l = start_len
        for i in range(steps):
            turtle.forward(l)
            l = l + delta_len
```

```
            turtle.left(delta_angle)
```

Nun fangen wir an, Spiralen zu zeichnen. Die Ergebnisse sind in Abbildung 3 zu sehen.

Abbildung 3: Die drei Spiralen. In der Mitte die erste (Stichstärke 1, Farbe Schwarz), links die zweite (Stichstärke 3, Farbe Schwarz) und rechts die dritte (Strichstärke 3, Farbe rot, Winkel 60 Grad).



```
>>> spiral(10, 20, 90, 20)
```

Die Striche sind dünn. Also zeichnen wir links davon mit dickeren Strichen:

```
>>> turtle.penup()
>>> turtle.goto(-500, 0)
>>> turtle.pendown()
>>> turtle.pensize(3)
>>> spiral(10, 20, 90, 20)
```

Das Ergebnis ganz links ist die gleiche Spirale, nur eben mit der dreifachen Stichstärke. Rechtwinklig und Schwarz ist etwas langweilig – wie wäre es, den Winkel zu verändern, und die Farbe? Das Ergebnis kommt ganz nach rechts.

```
>>> turtle.penup()
>>> turtle.goto(500, 0)
>>> turtle.color("dark red")
>>> turtle.pendown()
>>> spiral(10, 10, 60, 20)
```

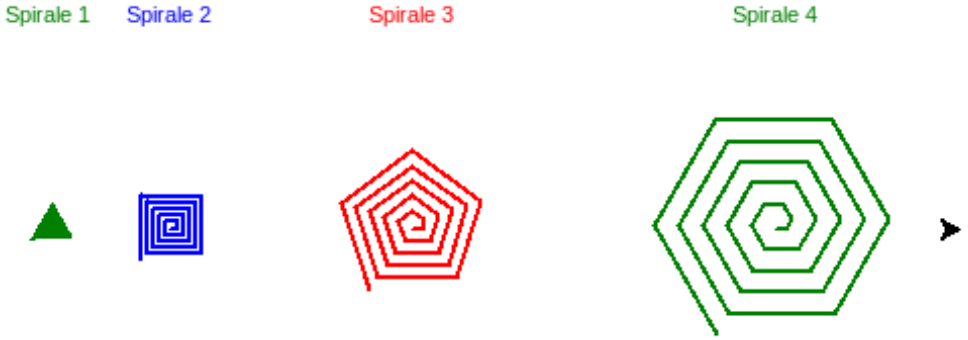
Die Möglichkeiten, mit turtle zu spielen, sind fast unbeschränkt. Abbildung 4 zeigt das Ergebnis eines weiteren Experimentes mit spiral. Wir nutzen auch ein selbstdefiniertes Unterprogramm goto, das die lästigen und sich wiederholenden Sequenz

turtle.penup()−turtle.goto(...)−turtle.pendown() ersetzt.

Um unsere Spiralen zu beschriften, nutzen wir das Unterprogramm turtle.write:

```
>>> def goto(x, y):
        turtle.penup()
        turtle.goto(x, y)
        turtle.pendown()

>>> turtle.pensize(2)
>>> for i in range(1,5):
        if i % 3 == 0:
                turtle.color("red")
        elif i % 3 == 1:
                turtle.color("green")
        else:
                turtle.color("blue")
        goto(30*i*(i-1),0)
        spiral(2+i/2, 1+i/4, 360/(i+2), 10+5*i)
        goto(30*i*(i-1), 100)
        turtle.write("Spirale " + str(i),
                     align="center")


>>> goto(450, 0)
>>> turtle.color("black")
```

Abbildung 4: Vier weitere Spiralen – diesmal mit Beschriftung. Man beachte das schwarze Dreieck ganz rechts, das die aktuelle Position und Richtung anzeigt.

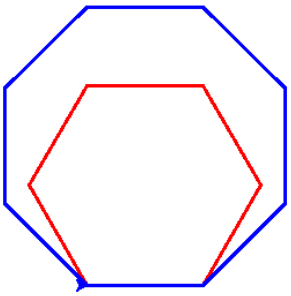


Lauf, Schildkröte lauf – einige Unterprogramme aus `turtle`:

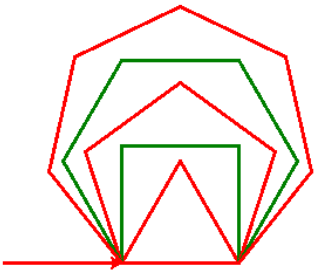
- Stift und Zeichenverhalten: pensize, penup, pendown, color

- Richtung: left , right
- Turtle bewegen: forward, goto
- Befehl rückgängig machen: undo
- Text ausgeben: write

Abbildung 5: Bilder, die mit der Turtle-Grafik erzeugt wurden.



Regelmäßiges Sechs- und Achteck, siehe Aufgabe 1.



Das Ergebnis von acht(!) Aufrufen des Polygon-Unterprogramms, siehe Aufgabe 2.

Aufgaben:

1. Schreiben Sie ein Unterprogramm polygon(n, ℓ, f), mit dem Sie für regelmäßige n-Ecke mit der Seitenlänge ℓ und in der Farbe f erzeugen können. Mit den folgenden Aufrufen können Sie z. B. das linke Bild von Abbildung 5 erzeugen:

```
>>> turtle.pensize(3)
>>> Polygon(6, 100, "red")
>>> Polygon(8, 100, "blue")
```

2. Der folgende Code ruft das Polygon-Unterprogramm achtmal auf. Wir würden eigentlich vier grüne und vier rote Polygone erwarten – tatsächlich sehen wir insgesamt fünf Polygone und einen geraden "Strich". Erssetzen sie die beiden mit ??? markierten Stellen durch geeignete ganzzahlige Ausdrücke um das rechte Bild aus Abbilduung 5 zu erzeugen.

```
>>> turtle.reset()
>>> turtle.pensize(3)
>>> for i in range(4):
        Polygon(???, 100, "green")
```

```
        Polygon(???, 100, "red")
```

6. Die Schildkröte zeichnet Histogramme

Nun haben wir alles was wir benötigen um Histogramme zu zeichnen. Histogramme bestehen im Wesentlichen aus einer Folge von Rechtecken. Ein $x * y$-Rechteck kann man zeichnen, indem man x Einheiten geht, dann links abbiegt, y Einheiten, links, x Einheiten, links, y Einheiten. Das Rechteck ist fertig und wir stehen wieder am Ausgangspunkt. Damit die Turtle die gleiche Richtung wie zu Beginn hat, drehen wir uns noch einmal nach links.

Wenn wir mehrere Rechtecke nebeneinander zeichnen wollen, gehen wir die erste Strecke noch einmal (das schadet nichts) und ein kleines bisschen weiter, nämlich x Einheiten *plus die Strichstärke*. Sonst würde die linke Seite des nächsten Rechtecks die rechte Seite unseres gerade gezeichneten Rechtecks übermalen – was zumindest dann, wenn die Rechtecke unterschiedliche Farben haben, ärgerlich wäre.

```
>>> import turtle
>>> def rectangle(len_x, len_y, pensize, color):
    turtle.pensize(pensize)
    turtle.pencolor(color)
    turtle.forward(len_x)
    turtle.left(90)
    turtle.forward(len_y)
    turtle.left(90)
    turtle.forward(len_x)
    turtle.left(90)
    turtle.forward(len_y)
    turtle.left(90)
    turtle.forward(len_x+pensize)
```

Abbildung 6 zeigt, wie man das Unterprogramm nutzt.

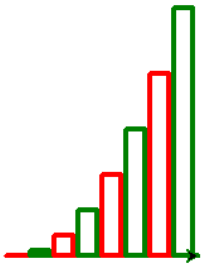
Allerdings fehlt noch die Beschriftung. Ein Rechteck mit einer Beschriftung unten und oben bezeichnen wir als “pillar” (“Säule”). Wir schreiben das Modul histogram mit einem Unterprogramm histogram.pillar. Geeignete Positionen, für die Texte below und above haben wir durch Versuch und Irrtum gefunden. Das Ergebnis findet sich in Abbildung 7.

Listing 8: Zeilen 1–40 aus histogram.py

```
1 """module histogram: Subprograms for histogram drawing."""
```

Abbildung 6: Einsatz von rectangle, um 0^2, 1^2, ... 7^2 als Histogramm darzustellen.

```
>>> for i in range(8):
        if i % 2 == 0:
                color = "red"
        else:
                color = "green"
        rectangle(15, i*i*4, 4, color)
```

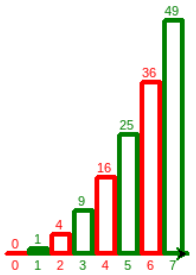


```
2
3  import turtle
4
5  def pillar(below, above, x, y, pensize, color):
6      """
7      Draws rectangle with given pensize and color; above and
8      below are texts positioned accordingly. Current position is
9      SW of rectangle, current + (x, y) is NE, final position is
10     current + (x + pensize, 0), final direction is east.
11
12     Requires initial direction to be east.
13     """
14
15     def write(message, distance):
16         turtle.right(90)
17         turtle.penup()
18         turtle.forward(distance)
19         turtle.write(message, align="center")
20         turtle.right(180)
21         turtle.forward(distance)
22         turtle.right(90)
23         turtle.pendown()
24
25     turtle.pensize(pensize)
26     turtle.pencolor(color)
27     turtle.forward(x / 2)
28     write(below, 16) # 16 chosen after some experiments
29     turtle.forward(x / 2)
30     turtle.left(90)
31     turtle.forward(y)
32     turtle.left(90)
33     turtle.forward(x / 2)
```

```
34      write(above, 2) # 2 chosen after some experiments
35      turtle.forward(x / 2)
36      turtle.left(90)
37      turtle.forward(y)
38      turtle.left(90)
39      turtle.forward(x + pensize) # move to the position
40                                  # for the next pillar
```

Abbildung 7: Einsatz von histogram.pillar.

```
>>> import histogram, turtle
>>> for i in range(8):
        if i % 2 == 0:
            c = "red"
        else:
            c = "green"
        histogram.pillar(i, i*i, 15, i*i*4, 4, c)
```



Zuguterletzt, hier ist die interaktive Sitzung, mit dem ich Abbildung 1 erzeugt habe:

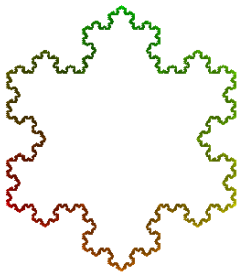
```
>>> import histogram
>>> Li = [0, 0, 2, 4, 14, 37, 54, 74, 131, 129, 147, 135, 88,
84, 36, 28, 19, 12, 3, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0]
>>> for i in range(0, 8):
        histogram.pillar(i, Li[i], 20, Li[i], 3, "grey")

>>> for i in range(8, 10):
        histogram.pillar(i, Li[i], 20, Li[i], 3, "blue")

>>> for i in range(10, 11):
        histogram.pillar(i, Li[i], 20, Li[i], 3, "red")

>>> for i in range(11, 13):
        histogram.pillar(i, Li[i], 20, Li[i], 3, "green")

>>> for i in range(13, 26):
        histogram.pillar(i, Li[i], 20, Li[i], 3, "grey")
```

Aufgabe: Stellen Sie das Ergebnis Ihres Zufallsexperiments aus Aufgabe 2 aus Abschnitt 4 als buntes Histogramm dar!

7. Schneeflocken in Regenbogenfarben

Mit der turtle-Grafik kann man nicht nur nützliche Diagramme sondern auch schöne bunte Bilder, wie die Schneeflocke in Abbildung 8, zeichnen.

Abbildung 8: Eine bunte Koch-Schneeflocke.



Um von A nach B zu gehen, kann man *entweder* den direkten Weg oder einen Umweg nehmen. Zum Beispiel sucht man einem Punkt C, und geht zuerst von A nach C und dann von C nach B. Von A nach C kann man wiederum entweder den direkten Weg gehen, oder erneut einen Umweg. Ebenso für C nach B. D. h., das Programm "von hier nach dort" ruft sich rekursiv selbst auf. Um irgendwann tatsächlich am Punkt B anzukommen, muss die Rekursion irgendwann abbrechen.

Diese Idee setzen wir wie folgt um: Wenn die Blickrichtung von A zu B geht, und die Entfernung zwischen A und B ℓ Schritte beträgt, definieren wir den Punkt C durch die folgende Anweisung:

- drehe Dich 45 Grad nach rechts
- gehe $\ell\sqrt{2}$ Schritte vorwärts

Mit der folgenden Anweisung kommt man dann von C nach B

- drehe Dich um 90 Grad nach links
- gehe $\ell\sqrt{2}$ Schritte vorwärts

Die ursprüngliche Blickrichtung erhält man, indem man zum Schluss die folgende Anweisung befolgt:

- drehe Dich 45 Grad nach rechts

Das folgende Programm aus dem Paket fractal setzt diese Anweisung in Python um. A

ist der Standort der Turtle, B ist der Punkt in Turtle-Richtung, dessen Entfernung length beträgt. Der Parameter step definiert die Rekursionstiefe. Null bedeutet einen Weg ohne Umwege, step> 0 bedeutet, zwei Wege mit step−1 von A nach C und C nach B.

Listing 9: Zeilen 40–53 aus fractal.py

```
40 # Levy-C-curve
41
42 sqrt_2 = math.sqrt(2) # we'll need this constant
43
44 def levy(step, length):
45     """Draws a Levy-C-curve."""
46     if step > 0:
47         turtle.right(45)
48         levy(step-1, length / sqrt_2)
49         turtle.left(90)
50         levy(step-1, length / sqrt_2)
51         turtle.right(45)
52     else:
53         turtle.forward(length)
```

Nun können wir unsere Wege gehen – jeweils 50 Einheiten nach rechts, auf immer längeren Umwegen, wie in Abbildung 9 dargestellt.

```
>>> import turtle, fractal
>>> turtle.pensize(2)
>>> turtle.color("black")
>>> fractal.levy(0, 50)
>>> fractal.turtle_invisible_move(50)
>>> turtle.color("red")
>>> turtle.color("blue")
>>> fractal.levy(1, 50)
>>> turtle.color("green")
>>> fractal.turtle_invisible_move(50)
>>> fractal.levy(2, 50)
>>> turtle.color("red")
>>> fractal.turtle_invisible_move(50)
>>> fractal.levy(3, 50)
>>> fractal.turtle_invisible_move(50)
>>> turtle.color("black")
>>> turtle.pensize(1)
>>> fractal.levy(4, 50)
>>> fractal.turtle_invisible_move(50)
```

Abbildung 9: Lévy-C-Kurven vom Grad 0 (schwarz), 1 (blau), 2 (grün), 3 (rot) und 4 (schwarz, dünne Pinselstärke).

Das Ergebnis ist die sogenannte C-Kurve von Lévy. Für hinreichend große Werte von step erhält man eine Figur, die an ein verschnörkeltes C erinnert, siehe Abbildung 10.

Dabei wird unsere Turtle-Graphik allerdings etwas langsam. Es gibt Turtle-Befehle, um die Animation der Turtle zu beschleunigen. Für das Paket fractal definieren wir das Unterprogramm turtle_fast und einige weitere Befehle. Unter anderem das Unterprogramm fractal.turtle_invisible_move, das wir beim Erstellen von Abbildung 9 bereits benutzt haben, um die Turtle unsichtbar vorwärts zu bewegen.

Listing 10: Zeilen 1–29 aus fractal.py

```
1  import turtle, math
2
3  # convenience functions (don't need to call turtle.xxx directly)
4
5  def turtle_fast():
6      """Sets turtle movement to high speed."""
7      turtle.speed(0)
8
9  def turtle_normal_speed():
10     """Sets turtle movement to normal speed."""
11     turtle.speed(3)
12
13 def turtle_pensize(size):
14     """Sets pensize (default 1) to a new value."""
15     turtle.pensize(size)
16
17 def turtle_hide():
18     """Makes the turtle invisible."""
19     turtle.hideturtle()
20
21 def turtle_show():
22     """Makes the turtle visible."""
23     turtle.showturtle()
24
25 def turtle_invisible_move(length):
26     """Moves turtle into given direction, without drawing."""
27     turtle.penup()
```

```
28      turtle.forward(length)
29      turtle.pendown()
```

Bleibt noch die Frage, wie wir unsere Kurven bunt gestalten, mit fließenden Farbübergängen wie in den Abbildungen 8 und 10. Dafür definieren wir zunächst einige Farben als rot-grün-blau-Tripel:

Listing 11: Zeilen 32–37 aus fractal.py

```
32 # color-triples to be used for colored curves
33
34 red =  (0.999, 0.001, 0.001)
35 blue = (0.001, 0.001, 0.999)
36 green = (0.001, 0.999, 0.001)
37 yellow = (0.999, 0.999, 0.001)
```

Damit haben wir die Zutaten um eine Lévy'sche C-Kurve auch bunt zu zeichnen.

Listing 12: Zeilen 55–68 aus fractal.py

```
55 def levy_colored(step, length, cols_0, cols_1):
56     """Draws a Levy-C-curve, colors changing cols_0 -> cols_1."""
57     (r1,  y1,  b1)  = cols_0
58     (r2,  y2,  b2)  = cols_1
59     cols_50 = ((r1+r2)/2,   (y1+y2)/2,   (b1+b2)/2)
60     if step > 0:
61         turtle.right(45)
62         levy_colored(step-1, length / sqrt_2, cols_0, cols_50)
63         turtle.left(90)
64         levy_colored(step-1, length / sqrt_2, cols_50, cols_1)
65         turtle.right(45)
66     else:
67         turtle.color(cols_50)
68         turtle.forward(length)
```

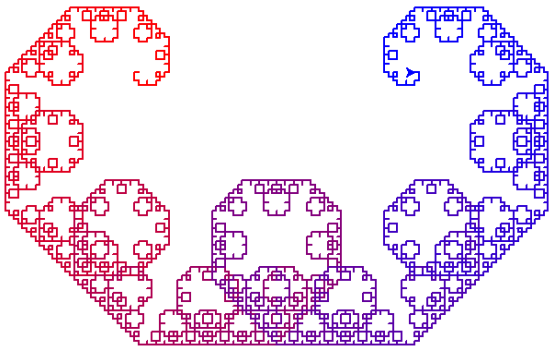
Der folgende Aufruf liefert einen Farbverlauf von Rot nach Blau, siehe Abbildung 10:

```
>>> fractal.turtle_fast()
>>> fractal.turtle_pensize(2)
>>> fractal.levy_colored(12, 300, fractal.red, fractal.blue)
```

Was hat das Ganze mit Schneeflocken zu tun? Die C-Kurve ist ein Fraktal[14]. Für die

[14]Siehe `https://de.wikipedia.org/wiki/Fraktal`.

Abbildung 10: Ergebnis von fractal.levy_colored(12, 300, fractal.red, fractal.blue)



Schneeflocken brauchen wir ein anderes Fraktal, eine “Koch-Kurve”. Die Anweisungen, mit denen wir eine Koch-Kurve erzeugen, sind denen der C-Kurve sehr ähnlich:

- (Punkt A) gehe $\ell/3$ Schritte vorwärts
- drehe Dich 60 Grad nach rechts
- gehe $\ell/3$ Schritte vorwärts (Punkt C)
- drehe Dich 120 Grad nach links
- gehe $\ell/3$ Schritte vorwärts
- drehe Dich 60 Grad nach rechts
- gehe $\ell/3$ Schritte vorwärts

Es ist nicht schwierig, das entsprechende Unterprogramm zu anzugeben. Das Ergebnis ist allerdings noch keine “Schneeflocke”, sondern nur eine verschnörkelte Kurve, die zwei Punkte miteinander verbindet. Siehe Abbildung 11.

Listing 13: Zeilen 71–86 aus fractal.py

```
71 # Koch-curve and snowflake
72 def koch(step, length):
73     """
74     Draws a Koch-curve of given 'length'.
75     Recursion depth: 'step'.
76     """
77     if step < 1:
78         turtle.forward(length)
```

```
79      else:
80          koch(step-1, length / 3)
81          turtle.right(60)
82          koch(step-1, length / 3)
83          turtle.left(120)
84          koch(step-1, length / 3)
85          turtle.right(60)
86          koch(step-1, length / 3)
```

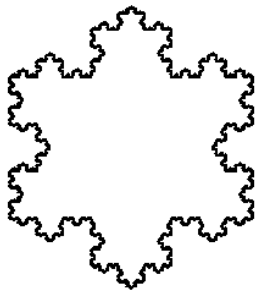
Abbildung 11: Ergebnis von fractal.koch(5, 200).

Eine Schneeflocke erhält man indem man mehrere Koch-Kurven, wie im folgenden Unterprogramm, zusammenfügt. Das Ergebnis (mit 3 Koch-Kurven) ist endlich unsere Schneeflocke, siehe Abbildung 12.

Listing 14: Zeilen 88–96 aus fractal.py

```
88  def koch_snowflake(step, length, lines):
89      """
90      Calls koch 'lines' times, turns to the left by
91      360/'lines' degrees after each call.
92      """
93      angle = 360/lines
94      for i in range(lines):
95          koch(step, length)
96          turtle.left(angle)
```

Abbildung 12: Ergebnis von fractal.koch_snowflake(5, 200, 3).



Noch ist die Schneeflocke nicht bunt. Die Unterprogramme, mit denen Abbildung 8 tatsächlich erzeugt wurde, sind die folgenden:

Listing 15: Zeilen 101–137 aus fractal.py

```
101 def koch_color(step, length, cols_0, cols_1):
102     """
103     Draws Koch-curve of given 'length' with recursion 'step'
104     slowly changes color from cols_1 to cols_1
105     cols_i shall hold a triple of values between 0 and 1.
106     """
107
108     (r1, y1, b1) = cols_0
109     (r2, y2, b2) = cols_1
110     cols_50 = ((r1+r2)/2, (y1+y2)/2, (b1+b2)/2)
111     cols_25 = ((2*r1+r2)/4, (2*y1+y2)/4, (2*b1+b2)/4)
112     cols_75 = ((r1+2*r2)/4, (y1+2*y2)/4, (b1+2*b2)/4)
113     if step <= 1:
114         turtle.color(cols_50[0], cols_50[1], cols_50[2])
115         turtle.forward(length)
116     else:
117         koch_color(step-1, length / 3, cols_0, cols_25)
118         turtle.right(60)
119         koch_color(step-1, length / 3, cols_25, cols_50)
120         turtle.left(120)
121         koch_color(step-1, length / 3, cols_50, cols_75)
122         turtle.right(60)
123         koch_color(step-1, length / 3, cols_75, cols_1)
124
125 def koch_snowflake_color(step, length, lines, cols):
126     """
127     Calls koch_color lines times, turns to the left by
128     360/lines degrees after each call.
129     cols: list of color-triples
130     colors change from cols[0] to ... to cols[lines-1]
131     and back to cols[0]
132     Requries length(cols) >= lines.
133     """
134     angle = 360/lines
135     for i in range(lines):
136         koch_color(step, length, cols[i], cols [(i+1) % lines])
137         turtle.left(angle)
```

Aufgaben:

1. Experimentieren Sie mit der C-Kurve. Welche Figuren erhält man, wenn man die

C-Kurve, analog zur Koch'schen Schneeflocke, zu einem geschlossenen Linienzug zusammensetzt?

2. Experimentieren Sie mit der Koch-Kurve. Was erhält man wenn man nicht drei, sondern vier, fünf, ..., Koch Kurven zu einer größeren "Schneeflocke" zusammensetzt?

3. Implementieren sie die berühmte "Drachenkurve"[15] in Python. Experimentieren Sie damit.

8. Warum nicht immer Python?

Wenn Sie bis zu dieser Stelle gelesen und die Aufgaben bearbeitet haben, können Sie leidlich mit Python programmieren. Vielleicht halten Sie Python für eine so tolle Programmiersprache, dass Sie alle Programmierprobleme damit lösen wollen.[16] Das ist keine gute Idee – erst recht nicht für die wissenschaftlich ausgebildete Informatikerin oder den wissenschaftlich ausgebildeten Informatiker, die oder der Sie einmal sein werden.

Python hat die wesentlichen Eigenschaften einer *Skriptsprache*. D.h., es ist ideal für die Entwicklung

- überschaubar kurzer Programmen ("Skripte"),
- die nicht über lange Zeit von verschiedenen Leuten gewartet werden müssen und
- an die keine hohe Ansprüche an Sicherheit oder Fehlertoleranz gestellt werden.

Zum Beispiel können Sie Variablen einfach benutzen.

In kompilierten (Nicht-Skript-)Sprachen wie Ada, C, C++, Java, Pascal, ... müssen Sie typischerweise Variablen *deklarieren*, bevor Sie diese benutzen können. Sie geben den *Typ der Variablen* en. Dieser kann sich während der Lebenszeit der Variablen nicht ändern. In Python hat jedes Objekt einen Typ (z.B. String, ganze Zahl, Liste, ...), aber einer Variable, die ein Objekt von einem Typ enthält, kann jederzeit ein Objekt von einem anderen Typ zugewiesen werden. Entsprechend müssen in Nicht-Skript-Sprachen auch Funktionen deklariert werden – eine Funktion, die laut Deklaration einen String als Ergebnis zurückgeben soll, kann keine Zahl zurückgeben, und umgekehrt. Verletzt ein Programm diese Vorgabe, liefert der Compiler eine Fehlermeldung statt eines ausführbaren Programms.

[15] Siehe `https://de.wikipedia.org/wiki/Fraktal`.

[16] Es gibt aus gutem Grund ziemlich viele Python-Enthusiasten. Aber die meisten verstehen genug von Software-Engineering, um Python nicht überall einsetzen zu wollen.

Bei kurzen Programmen, und wenn Sie keine großen Ansprüche an die Sicherheit und Zuverlässigkeit Ihrer Programme stellen (kurz, wenn Sie ein “Skript” schreiben), sind Deklarationszwang und Typenbindung lästig. Wenn aber

- Ihre Programme länger und unübersichtlicher sind,
- über einen längeren Zeitraum genutzt, gepflegt bzw. weiterentwickelt werden sollen, ggf. von wechselnden Personen,
- oder der Schaden bei einem Fehlverhalten Ihres Programms beträchtlich wäre,

sollten Sie *keine Skriptsprache* wie Python verwenden. Der Compiler ist, dank Deklarationszwang und Typenbindung (und weiterer Dinge, die wir hier nicht diskutieren) ein extrem nützliches Werkzeug um Fehler zu entdecken und zu beheben, ohne dass die Fehler jemals Bestandteil eines ausführbaren Programmes gewesen sind.

Unten sehen Sie “dummy.py”. Die Funktion dummy.text_to_num soll Buchstabenfolgen ähnlich einer Handy-Tastatur in Ziffernfolgen übersetzen.

Listing 16: Zeilen 1–28 aus dummy.py

```
1  """this is a dummy-module with a broken text_to_num function"""
2
3  def text_to_num(text):
4      def char_to_num(c):
5          if c >= "a" and c <= "c":
6              return "2"
7          elif c <= "f":
8              return "3"
9          elif c <= "i":
10             return "4"
11         elif c <= "l":
12             return "5"
13         elif c <= "o":
14             return "6"
15         elif c <= "s":
16             return "7"
17         elif c <= "v":
18             return 8
19         elif c <= "z":
20             return "9"
21         elif c == " ":
22             return "0"
23         else:
```

```
24                 return "?"
25         num = char_to_num(text[0])
26         for i in range(1, len(text)):
27             num = num + char_to_num(text[i])
28         return num
```

Zeile 18 ist fehlerhaft. Deshalb liefert char_to_num manchmal eine Zahl statt, wie beabsichtigt, einen String zurück. Die Funktion text_to_num liefert ihrerseits

- oft das richtige Ergebnis, ("ada" und "programmieren"),
- manchmal ein falsches Ergebnis, ("tut")
- und manchmal eine Fehlermeldung zurück ("gut").[17]

```
>>> import dummy
>>> dummy.text_to_num("ada")
'232'
>>> dummy.text_to_num("programmieren")
'7764726643736'
>>> dummy.text_to_num("tut")
24
>>> dummy.text_to_num("gut")
Traceback (most recent call last):
  File "<pyshell#150>", line 1, in <module>
    dummy.text_to_num("gut")
  File ".../python_bsp/dummy.py", line 27, in text_to_num
    num = num + char_to_num(text[i])
TypeError: Can't convert 'int' object to str implicitly
```

Bei einer kompilierten Sprache hätte der Compiler den Fehler sofort entdeckt und gemeldet. Das vorliegende Beispiel ist ein Trivialbeispiel – aber stellen Sie sich vor, Sie würden in einem Programm mit tausenden von Zeilen nach einem solchen Fehler suchen.

Ein weniger wichtiger Punkt, den man aber auch nicht unterschätzen sollte, ist, dass kompilierte Programme meistens deutlich schneller laufen als (interpretierte) Skripte.

[17] Die Fehlermeldung liegt daran, dass der Operator "+" *entweder* zwei Strings hintereinander hängen kann (hier beabsichtigt) *oder* zwei Zahlen addieren (falsch, siehe dummy.text_to_num("tut")), *aber nicht*, keinesfalls Zahlen und Strings miteinander verknüpfen kann.

A. Anhang: Quadratwurzeln Reloaded

Das Unterprogramm sqrt.sqrt funktioniert zwar, ist aber etwas langsam, denn die Schleife wird etwa $\sqrt{y}$-mal durchlaufen. Wenn $y \approx 2^n$ eine Zahl der Länge n ist, dann ist $\sqrt{y} \approx 2^{n/2}$. D.h., die Laufzteit von sqrt.sqrt ist *exponentiell* in der Eingabelänge n. Das geht viel effizienter:

1. $\ell = 0 \leq \sqrt{y} < y + 1 = h$.

2. Seien ℓ eine untere Schranke und h eine obere Schranke für die Quadratwurzel von y. Genauer: Seien $\ell \leq \sqrt{y} < h$. Wiederhole:

   a) Berechne $m = (\ell + h)/2$.

   b) Vergleiche $m * m$ mit y.

      i. Wenn $m * m = y$, also $m = \sqrt{y}$: Quadratwurzel gefunden!

      ii. Wenn $m * m \leq y$, also $m \leq \sqrt{y} < h$: Neues Intervall $m \leq \sqrt{y} < h$.

      iii. Wenn $m * m > y$, also $\ell \leq \sqrt{y} < m$: Neues Intervall $\ell \leq \sqrt{y} < m$.

   bis Quadratwurzel gefunden oder $\ell = h - 1$ (Intervall hat Länge 1).

In jedem jedem Berechnungsschritt halbieren wir das Intervall $\{\ell \ldots, h - 1\}$, in dem $\sqrt{y}$ liegt, etwa. Deshalb finden wir das gesuchte Ergebnis in etwa $\log_2(y)$ Schritten. Für $y \approx 2^n$ ist $\log_2(y) \approx n$, d.h. die Laufzeit ist dann nur noch *linear* in der Eingabelänge n, statt exponentiell. Deshalb schreiben wir das Unterprogramm fast_sqrt:

Listing 17: Zeilen 26–42 aus sqrt.py

```
26 def fast_sqrt(y):
27     """Same as sqrt, but much faster."""
28
29     def fsqrt(y, low, high):
30         """result must be in range(low, high+1)"""
31         mid = (low + high) // 2
32         square_of_mid = mid * mid
33         if low +1 == high:
34             return low
35         elif square_of_mid == y:
36             return mid # uses obervation 2.a
37         elif square_of_mid < y:
38             return fsqrt(y, mid, high) # observation 2.b
```

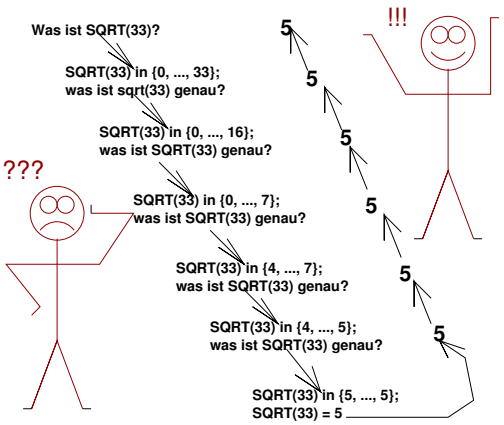
```
39          else:
40              return fsqrt(y, low, mid) # observation 2.c
41
42      return fsqrt(y, 0, y+1) # observation 1
```

Man beachte die verschachtelte Struktur. Die äußere Funktion fast_sqrt basiert auf der ersten Beobachtung, die innere fsqrt nutzt die zweite Beobachtung und ruft sich selber rekursiv auf. Beim Aufruf von fast_sqrt(33) passiert zum Beispiel das folgende:

1. Python ruft fast_sqrt(33, 0, 33) auf.
2. Python setzt mid=17 und berechnet square_of_mid=289. Weil $289 > 33$ ist, ruft Python dann fsqrt(33, 0, 17) auf (Beobachtung 2.c).
3. Python setzt mid=8, berechnet square_of_mid=64 ruft fsqrt(33, 0, 8) auf, weil $64 > 33$ ist (wieder 2.c)
4. Python setzt mid=4 und berechnet square_of_mid=16. Diesmal ist $16 < 33$ (Beobachtung 2.b), deshalb ruft Python fsqrt(33, 4, 8) auf.
5. Python setzt mid=6, berechnet square_of_mid=36 und, weil $36 > 33$ ist, wird nun fsqrt(33, 4, 6) aufgerufen.
6. Python setzt mid=5, berechnet square_of_mid=25 und wendet Beobachtung 2.b an, weil $25 < 33$ ist: fsqrt(33, 5, 6)
7. Es ist $5 + 1$ gleich 6, also liefert der Aufruf von fsqrt(33, 5, 6) den Wert 5.
8. Nun können wir fsqrt(33, 4, 6) berechnen. Es ist 5.
9. Nun können wir fsqrt(33, 4, 8) berechnen: Es ist 5.
10. Nun können wir fsqrt(33, 0, 8) berechnen: Es ist 5.
11. Nun können wir fsqrt(33, 0, 17) berechnen: Es ist 5.
12. Nun können wir fsqrt(33, 0, 34) berechnen: Es ist 5.
13. Nun können wir fast_sqrt(33) berechnen: Es ist 5.

Die Funktion fsqrt hat also ein teilweises Wissen über die gesuchte Lösung (in welchem Intervall liegt sie). Entweder verfeinert sie dieses Wissen (das Intervall wird kleiner) oder sie findet die Lösung direkt. Nach endlich vielen Schritten ist das Intervall immer klein

genug, um die Lösung direkt anzugeben. Die folgende Grafik verdeutlicht den Workflow beim Berechnen der Quadratwurzel von 33:



Wir können (nein, müssen) unser neues Unterprogramm fast_sqrt noch testen:

Listing 18: Zeilen 44–55 aus sqrt.py

```
44 """
45 >>> import sqrt
46 >>> sqrt.fast_sqrt(0)
47 0
48 >>> sqrt.fast_sqrt(3343*3343)
49 3343
50 >>> sqrt.fast_sqrt(3343*3343-1)
51 3342
52 >>> sqrt.fast_sqrt(3343*3343+1)
53 3343
54 >>>
55 """
```

Unsere Tests scheitern – anscheinend ist fast_sqrt korrekt.

A.1. Ist der rekursive Algorithmus wirklich schneller?

Die Funktion fast_sqrt ist offensichtlich komplizierter als sqrt. Gibt den behaupteten Geschwindigkeitsvorteil wirklich? Um die Zeit zu messen, nutzen wir das Python-Standardpacket time:

```
>>> import time, sqrt
>>> def benchmark(func, arg, repetitions):
        start = time.clock()
        for i in range(repetitions):
                func(arg)
        stop = time.clock()
        return stop-start

>>> benchmark(sqrt.sqrt, 3343, 10000)
0.06477399999999989
>>> benchmark(sqrt.fast_sqrt, 3343, 10000)
0.07146400000000064
```

Autsch! Tatsächlich ist fast_sqrt ist *langsamer* als sqrt! Haben wir uns so geirrt? Zum Vergleich berechnen wir die Quadratwurzel einer doppelt so langen Zahl.

```
>>> benchmark(sqrt.sqrt, 3343*3343, 10000)
2.1000189999999996
>>> benchmark(sqrt.fast_sqrt, 3343*3343, 10000)
0.113173999999999
```

Die Laufzeit von fast_sqrt verdoppelt sich nicht einmal, (von 0.071 auf 0.113), die von sqrt ver-32-facht sich (von 0.065 auf 2.1). Nun untersuchen wir, untersuchen wir systematich die Laufzeiten von sqrt und fast_sqrt bei den Eingaben 3343, 33430, ...:

```
>>> l_slow = [0] * 10
>>> l_fast = [0] * 10
>>> for i in range(10):
        l_slow[i] = benchmark(sqrt.sqrt, 3343*10**i, 5)
        l_fast[i] = benchmark(sqrt.fast_sqrt, 3343*10**i, 5)
```

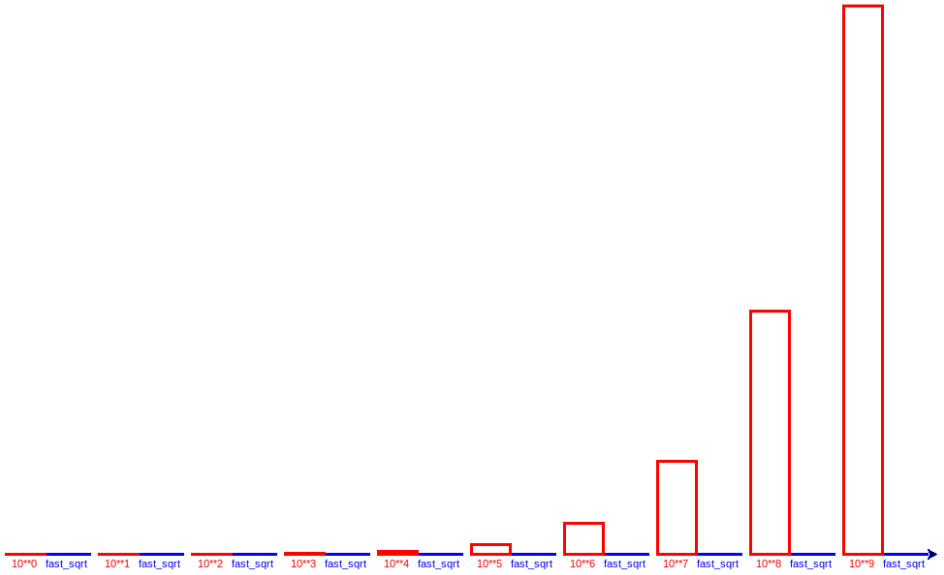
Die Ergebnisse sind als Histogramm in Abbildung 13 dargestellt. Sie sind typisch für den Vergleich von exponentieller Laufzeit (sqrt, rot) mit linearer Laufzeit (fast_sqrt, blau).

A.2. Quadratwurzelberechnung mit der Standardbibliothek

Standardfunktionen wie, z.B., Quadratwurzeln muss man in Python natürlich nicht selbst programmieren – für uns ist das nur ein einführendes Beispiel. In dem Modul `math` gibt es bereits ein Unterprogramm zum Berechnen von Quadratwurzeln:

```
>>> import math
```

Abbildung 13: Vergleich der Laufzeiten von sqrt (rot) und fast_sqrt (blau); Berechnung der zehn Quadratwurzeln von $3342 * 10^i$, $i \in \{0, \ldots, 9\}$.




```
>>> math.sqrt(3343)
57.81868210189506
>>> int(math.sqrt(3343))
57
>>> math.sqrt(529)
23.0
>>> int(math.sqrt(529))
23
```

Wie man sieht, liefert math.sqrt eine Fließkommazahl. Mit `int` kann man abrunden und erhält das (von uns) gewünschte ganzzahlige Ergebnis – und zwar sehr effizient. Für alle unsere Testfälle $x \in \{0, 43343 * 3343, 3343 * 3343 - 1, 3343 * 3343 + 1\}$ liefert int(math.sqrt(x)) das jeweils richtige Ergebnis. Also gibt es wohl keinen Grund, eine selbstdefinierte Quadratwurzelfunktion zu nutzen ... außer, das es sich um ein schönes Beispiel handelt, um die Implemtation einfacher Algorithmen zu lernen ...

Vorsichtshalber vergleichen wir die beiden Quadratwurzelfunktionen. Wir setzen eine Variable big auf irgend eine ziemlich große Zahl und wenden nun einmal sqrt.fast_sqrt und int(math.sqrt) an. Beide liefernt praktisch "sofort" ein Ergebnis:

```
>>> big =123478924789078907893567*2**800 + 234789234891234234890
```

```
>>> y1 = sqrt.fast_sqrt(big)
>>> y2 = int(math.sqrt(big))
>>> y1
907391455023720274374303240180227670105539105033374001067150459360069209670541951928928859915700108076190649211713913730954579561460
>>> y2
907391455023720227208767086999584239519505474910448031366871824343084669326647810955370682633125665735697461875304536530625891401728
>>> y1 - y2
471655361531806434305860336301229259697002786350169845403438941409735581772825744423404931873364093772003286881597 32
```

Halt! Stop! Hilfe! Irgend etwas stimmt nicht, denn die beiden Ergebnisse sind verschieden. Es kann nur ein richtiges Ergebnis $y \in \mathbb{N}$ geben. Das ist definiert durch

$$y * y \leq \text{big} < (y+1) * (y+1).$$

Probieren wir aus, ob y1 falsch ist, oder y2 (oder gar beide falsch sind):

```
>>> big >= y1*y1
True
>>> big < (y1+1)*(y1+1)
True
>>> big >= y2*y2
True
>>> big < (y2+1)*(y2+1)
False
```

Also ist y1 richtig, aber y2 zu klein![18] *Ist die Mathe-Bibliothek von Python kaputt?*

Keine Sorge, grundsätzlich ist die Mathe-Bibliothek von Python in Ordnung. Aber die Funktion math.sqrt liefert eine Fließkommazahl als Ergebnis. Fließkommazahlen haben eine vorgegebene Genauigkeit. Sie repräsentieren deshalb fast immer nur *Näherungen* der tastächlichen Ergebnisse. Das oft, gut genug, aber manchmal auch fatal. Um die Quadratwurzeln großer Zahlen zu finden, brauchen wir Programme, die den Umweg über Näherungen bzw. Fließkommazahlen vermeiden, zum Beispiel sqrt.fast_sqrt.[19]

[18] Zwar hat int(math.sqrt(x)) bei allen Testfällen das richtige Ergebnis geliefert. Die Tests sind also fehlgeschlagen. Doch während erfolgreiche Tests beweisen, dass ein Programm fehlerhaft ist, beweisen erfolglose Tests eben nicht, dass ein Programm korrekt ist.

[19] Theoretisch liefert auch sqrt.sqrt immer das richtige Ergebnis liefern. Praktisch scheitert das an der exponentiellen Laufzeit. Die Berechnung von sqrt.sqrt(big) würde viele *Milliarden Jahre* dauern, während fast_sqrt.sqrt(big) viel weniger als eine Sekunde braucht.