Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Hannes Saffrich, Marc Fuchs
Wintersemester 2020

Universität Freiburg
Institut für Informatik

Übungsblatt 11

Abgabe: Montag, 01.02.2021, 9:00 Uhr morgens, über git[1]

Hinweis

Aufgabenteile werden mit **0 Punkten** bewertet wenn:

- Dateien und Definitionen nicht so benannt sind, wie im Aufgabentext gefordert;
- Dateien falsche Formate haben, z.B. PDF statt plaintext; oder
- Pythonskripte wegen eines Syntaxfehlers nicht ausführbar sind.

Es gibt **Punktabzug** wenn:

- Funktionen keine oder falsche Typannotationen aufweisen. Ausnahme: Bei Funktionen, die stets `None` zurückgeben, kann der Rückgabetyp weggelassen werden.

Gruppenaufgaben müssen von allen Mitgliedern abgegeben werden und in der ersten Zeile müssen die Mitglieder in einem Kommentar vermerkt werden, z.B:

```
# Gruppe: xy123, yz56, zx934
```

Aufgabe 11.1 (Testing: `testing.py`, Punkte: 3+1)

Laden Sie sich die Datei `testing.py` von der Vorlesungsseite herunter.

Die Datei enthält eine Funktion `contains`, die überprüfen soll, ob ein Element in einer Liste enthalten ist. Die Funktion enthält aber einen Fehler und funktioniert nicht für alle Argumente wie erwünscht.

(a) Schreiben Sie 3 Unittests, die volle Coverage erreichen (jedes Statement wird getestet) und den Randfall abdecken. Zwei Ihrer Testfälle sollten nun fehlschlagen. Definieren Sie die Tests als Funktionen, deren Namen mit `test_` beginnen und verwenden Sie `pytest` zum Ausführen der Tests, wie in der Vorlesung beschrieben.

(b) Finden und reparieren Sie den Fehler, sodass die Testfälle erfolgreich durchlaufen. Erklären Sie in einem Kommentar, wo der Fehler lag (ein Satz reicht).

[1] `https://inpro.informatik.uni-freiburg.de/`

Hinweis

Eine Generator-Funktion, die einen Generator von ganzen Zahlen als Argument nimmt und Strings generiert, hat folgende Typsignatur:

```
from typing import Iterator
def ints_to_strs(int_gen: Iterator[int]) -> Iterator[str]:
    for i in int_gen:
        yield str(i)
```

Aufgabe 11.2 (Generatoren; Datei: `generators.py`; Punkte: 2+2+2+2)

In dieser Aufgabe sollen Sie bestimmte Generator-Funktionen implementieren.

Verwenden Sie dabei lediglich die folgenden Generator-Funktionen aus der Standard-Bibliothek und implementieren Sie die restliche Funktionalität selbst: `range`, `enumerate` und `filter`.

Die Generator-Funktionen sollen dabei das folgende Kriterium erfüllen: um das nächste Element zu generieren, dürfen nur die Berechnungen durchgeführt werden, die dafür auch strikt notwendig sind. Insbesondere, nicht die Berechnungen, die die darauffolgenden Elemente erzeugen.[2]

Schreiben Sie zu jeder Generator-Funktion f mindestens einen Unittest `test_`f und stellen Sie mit `pytest` sicher, dass dieser erfolgreich ist.

Schreiben Sie eine Generator-Funktion

(a) `chain`, die zwei Generatoren `xs` und `ys` als Argumente nimmt, und diese verkettet, d.h. erst die Elemente von `xs` generiert und dann die von `ys`.

(b) `take`, die eine ganze Zahl `n` und einen Generator `xs` als Argumente nimmt, und der Reihe nach die ersten `n` Elemente von `xs` generiert.

(c) `chunks`, die eine ganze Zahl `n` und einen Generator `xs` als Argument nimmt und Listen der nächsten `n` Elemente von `xs` generiert. Sind am Ende weniger als `n` Elemente übrig, so soll einmalig eine kürzere Liste generiert werden.

(d) `cycle`, die einen (endlichen) Generator `xs` als Argument nimmt, die Elemente von `xs` generiert, und nach dem letzten Element wieder mit dem ersten Element weitermacht. Sie können hierzu den Generator `xs` einmalig in einer Liste aufsammeln. Verwenden Sie `take`, um diesen Generator zu testen.

[2] Der Sinn eines Generators besteht ja gerade darin die Elemente erst bei Bedarf zu generieren ("lazy evaluation"). Dadurch kann in bestimmten Fällen Speicherbedarf und Ausführungszeit gespart werden. Ansonsten könnten wir auch gleich eine normale Funktion schreiben, die eine Liste aller zu generierenden Elemente zurückgibt (zumindest wenn der Generator endlich ist).

Aufgabe 11.3 (CSV Parsing; Datei: `csv.py`; Punkte: 2+2+2+0)

Das CSV-Format (Comma Separated Values) erlaubt es Tabellen in Textdateien darzustellen, sodass diese einfach maschinell verarbeitet werden können.

Jede Zeile einer `.csv`-Datei entspricht dabei der Zeile einer Tabelle. Innerhalb einer Zeile werden die Tabellenzellen durch Kommas getrennt.[3]

CSV findet zum Beispiel beim Online-Banking Anwendung. Die meisten Anbieter erlauben es z.B. die Umsatzübersicht als `umsatz.csv` mit folgendem Inhalt zu exportieren:

```
Datum,Verwendungszweck,Betrag
30.12.2020,Bafoeg-Foerdergeld,+514.00
01.01.2021,Miete,-400.00
03.01.2021,Bestellung im Amazonas,-43.20
```

Wie in der vorherigen Aufgabe sollen auch in dieser Aufgabe Ihre Generator-Funktionen nur die Berechnungen durchführen, die für das jeweils nächste Element benötigt werden.

Das Beispiel `umsatz.csv` finden Sie auch auf der Vorlesungsseite.

(a) Schreiben Sie eine Generator-Funktion `lines`, die den Dateipfad einer Textdatei als Argument nimmt und deren Zeilen generiert. Beispiel:

```
>>> for line in lines("umsatz.csv")
>>>     print(line)
Datum,Verwendungszweck,Betrag
30.12.2020,Bafoeg-Foerdergeld,+514.00
01.01.2021,Miete,-400.00
03.01.2021,Bestellung im Amazonas,-43.20
```

(b) Schreiben Sie eine Generator-Funktion `parse_csv`, die einen Generator von Strings als Argument nimmt, diese als Zeilen einer CSV-Datei auffasst und die zugehörigen Listen von Zellen generiert. Beispiel:

```
>>> list(parse_csv(["Datum,Verwendungszweck,Betrag",
                    "30.12.2020,Bafoeg-Foerdergeld,+514.00"]))
[ ["Datum", "Verwendungszweck", "Betrag"],
  ["30.12.2020", "Bafoeg-Foerdergeld", "+514.00"] ]
```

Verwenden Sie hierbei *nicht* das `csv`-Modul aus der Standard-Bibliothek.

(c) Schreiben Sie eine Funktion `update_balance`, die als Argument einen Startkontostand `balance` und einen Dateipfad `csv_path` nimmt, und den Kontostand nach dem Ausführen der Transaktionen zurückgibt. Verwenden Sie hierzu die

[3] Das `CSV`-Format erlaubt es eigentlich auch Strings zu verwenden, sodass Kommas innerhalb einer Tabellenzelle verwendet werden können. Zum Beispiel beschreibt dann `"[23,4]",25` zwei Spalten statt drei. Sie können dieses Feature in dieser Aufgabe ignorieren.

Funktionen `lines` und `parse_csv` aus den vorherigen Aufgabenteilen. Sie können dabei annehmen, dass `csv_path` eine gültige `.csv`-Datei beschreibt, die eine Tabelle mit dem gleichen Format wie im Beispiel enthält. Beispiel:

```
>>> update_balance(100.00, "umsatz.csv")
170.8   # == 100.0 + 514.0 - 400.0 - 43.2
```

(d) Atmen Sie tief durch und freuen Sie sich, dass ihr Code auch in der Lage wäre den Umsatz von Amazon zu verarbeiten. Hier könnte die `umsatz.csv` mehrere Terrabyte groß sein, aber Ihr Arbeitsspeicher ist im Regelfall nur ein paar Gigabyte groß. Würde Ihr Programm zunächst versuchen die gesamte Datei/Tabelle in den Speicher zu laden und dann erst den neuen Kontostand berechnen, dann könnten Sie diese Dateigrößen nicht verarbeiten, da Ihnen der Arbeitsspeicher ausgehen würde. Die Generatoren erlauben es dabei die einzelnen Verarbeitungsschritte wie gewohnt in verschiedene Funktionen zu unterteilen - ein weiterer Baustein um selbst komplexe Programme noch einigermaßen verständlich und wiederverwendbar aufzuschreiben.

Aufgabe 11.4 (Erfahrungen; Datei: `erfahrungen.txt`; 2 Punkte)

Notieren Sie hier Ihre Erfahrungen mit diesem Übungsblatt (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Schreiben Sie den groben Zeitaufwand, den Sie für das *gesamte* Blatt benötigt haben, bitte wie folgt in die erste Zeile:

```
Zeitaufwand: 3.5h

[... Freitext, wie bisher ...]
```

Wenn sich genug Leute daran halten, dann können wir ein Pythonskript schreiben, welches automatisiert die durchschnittliche Bearbeitungszeit berechnet ;-)