Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Hannes Saffrich, Michael Uhl
Wintersemester 2022

Universität Freiburg
Institut für Informatik

Übungsblatt 11

Abgabe: Montag, 16.01.2022, 9:00 Uhr morgens

Hinweis

Es gelten die selben Regeln wie bisher. Diese können in Blatt 10 eingesehen werden.

Hinweis

In diesem Übungsblatt müssen Sie für Aufgaben 11.1 und 11.2 **keine** Typannotationen schreiben. Für Aufgaben 11.3 und 11.4 sind hingegen wieder Typannotationen erforderlich.

Hinweis

In den meisten Aufgaben müssen Sie Funktionen definieren, deren Rumpf aus einem einzigen Ausdruck besteht - wie üblich in der Funktionalen Programmierung. Beispiel:

```
def inc(x: int) -> int:
    return x + 1
```

Diese Funktionen können Sie auch als Variable mit Funktionswert definieren:

```
from typing import Callable

inc: Callable[[int], int] = lambda x: x + 1
```

Diese Schreibweise ist insbesondere bei bestimmten verschachtelten Funktionen angenehmer zu lesen:

```
def mul_1(x: int) -> Callable[[int], int]:
    def mul_with_x(y):
        return x * y
    return mul_with_x

mul_2: Callable[[int], Callable[[int], int]] = lambda x: lambda y: x * y

assert mul_1(2)(3) == 6
assert mul_2(2)(3) == 6
```

Der Linter beschwert sich bei dieser Schreibweise mit

> "Do not assign a lambda expression, use a def."

Diese Meldung dürfen Sie ignorieren.

Aufgabe 11.1 (Funktionskomposition; Datei: `compose.py`; 2 Punkte)

In dieser Aufgabe sollen Ihre Funktionsdefinitionen außer einer `return`-Anweisung keine weiteren Zeilen enthalten, oder wie im Hinweis als Variable mit Funktionswert definiert werden.

Schreiben Sie eine Funktion `compose`, die eine einstellige Funktion `f` und eine zweistellige[1] Funktion `g` als Argument nimmt und die Funktionskomposition $f \circ g$ zurückgibt.

Beispiel:

```
inc = lambda x: x + 1
mul = lambda x, y: x * y
assert compose(inc, mul)(4, 2) == 9
```

Aufgabe 11.2 (Filter; Datei: `filter.py`; 2 + 2 Punkte)

In dieser Aufgabe sollen Ihre Funktionsdefinitionen außer einer `return`-Anweisung keine weiteren Zeilen enthalten, oder wie im Hinweis als Variable mit Funktionswert definiert werden.

(a) Schreiben Sie eine Funktion `my_filter`, welche zwei `set`s `xs` und `ys` entgegennimmt und ein neues `set` mit Elementen zurückgibt, welche nur in `xs`, aber nicht in `ys` vorkommen.

Beispiel:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
assert my_filter(set1, set2) == {1, 2}
```

(b) Schreiben Sie nun eine Funktion `my_diff`, die wieder zwei `set`s entgegennimmt und mithilfe von `my_filter` genau die Elemente zurückgibt, welche exakt einmal in beiden `set`s vorkommen.

Beispiel:

```
set1 = {1, 2, 3}
set2 = {2, 3, 4}
assert my_diff(set1, set2) == {1, 4}
```

[1] Eine zweistellige Funktion ist eine Funktion, die zwei Argumente entgegen nimmt, wie z.B. Addition.

Aufgabe 11.3 (Reducing Octals; Datei: `octs_to_int.py`; Punkte: 3)

In dieser Aufgabe sollen Ihre Funktionsdefinitionen, außer einer `return`-Anweisung, keine weiteren Anweisungen enthalten, oder wie im Hinweis als Variable mit Funktionswert definiert werden.

In unixoiden Systemen wie Linux, macOS, FreeBSD, usw. gibt es für Dateien verschiedene Rechte, die sogenannten Unix-Dateirechte[2]. Diese Rechte werden durch Oktalzahlen dargestellt, die im Gegensatz zum Dezimalsystem nicht 10, sondern 8 zur Basis haben. Um Verwechslungen zu vermeiden schreibt man Oktalzahlen mit führender `0`.

Soll nun der Dateieigentümer Lese- und Schreibzugriff, alle anderen aber nur Lesezugriff haben, so ist das entsprechende Recht mit `0644` kodiert. Diese Oktalzahlen wollen wir nun in Dezimalzahlen umwandeln.

Schreiben Sie eine Funktion `octs_to_int`, die eine Liste von Oktalziffern als Argument nimmt und die zugehörige postive ganze Zahl zurückgibt.

Wie auch im Dezimalsystem werden die Oktalzahlen so interpretiert, dass die erste Ziffer wie gewohnt den größten Exponenten besitzt.

Beispiel:

```
assert octs_to_int([6, 4, 4]) == 420
```

Verwenden Sie hierfür die `reduce`-Funktion aus dem Modul `functools`.

Aufgabe 11.4 (Differentiation und Integration; Punkte: 2+5+2; Datei: `integral.py`)

Die Ableitung einer Funktion $f : R \to R$ an einer Stelle x_0 ist bekanntermaßen der Grenzwert des Differenzenquotienten $\lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h}$. Durch diese Festlegung erhalten wir eine Funktion $D(f)$, die $x_0 \in R$ jeweils auf diesen Grenzwert abbildet, falls er existiert. Die Funktion D ist offensichtlich eine Funktion höherer Ordnung, weil sowohl ihr Argument f als auch ihr Ergebnis eine Funktion ist!

Genauso verhält es sich mit der Integration, nur dass hierbei weitere Argumente ins Spiel kommen.

(a) Definieren Sie eine Funktion, die aus der Funktion f und der Schrittweite h eine Funktion berechnet, die bei Anwendung auf x_0 den zentralen Differenzenquotienten von f bezüglich x_0 und h berechnet:

$$D(f,h)(x_0) = \frac{f(x_0+h) - f(x_0-h)}{2h}$$

Beispiel:

`differentiate(lambda x: 1 / 2 * x ** 2, 1e-2)(0)` entspricht `(lambda x: x)(0)`

Verwenden Sie folgende Signatur für Ihre Implementierung:
`differentiate(f: Callable[[float], float], h: float) -> Callable[[float], float].`

[2] https://de.wikipedia.org/wiki/Unix-Dateirechte

(b) Definieren Sie nun eine Funktion, die zu einer gegebenen Funktion f und Schrittanzahl $n > 0$ eine Funktion berechnet, die aus ihren beiden Parametern a und b, mit $a < b$, eine Approximation des bestimmten Integrals $\int_a^b f(x)dx$ berechnet.

Verwenden Sie zur Approximation des Integrals die Simpsonregel. Setzen Sie dazu

$$
\begin{aligned}
h &= (b-a)/n \\
x_i &= a + ih && 0 \leq i \leq n \\
s_i &= \frac{h}{6} \cdot (f(x_i) + 4 \cdot f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1})) && 0 \leq i < n
\end{aligned}
$$

Die gesuchte Approximation ist die Summe der s_i: $I(f,n)(a,b) = \sum_{i=0}^{n-1} s_i$.

Beispiel:

`integrate(lambda x: math.exp(x), 5)(0, 1)` entspricht `(lambda x: math.exp(x) - 1)(1)`

Verwenden Sie die folgende Signatur für Ihre Implementierung:
`integrate(f: Callable[[float], float], n: int) -> Callable[[float, float], float].`

(c) Schreiben Sie für beide Aufgabenteile jeweils einen aussagekräftigen Test. Da Sie hierbei Werte vom Typ `float` vergleichen (siehe Floating-Point-Arithmetik[3]), verwenden Sie beispielsweise die Funktion `approx` aus dem Paket `pytest`.

Aufgabe 11.5 (Erfahrungen; 2 Punkte; Datei: `NOTES.md`)

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

Editieren Sie hierzu die Datei `NOTES.md` im Abgabeordner dieses Übungsblattes auf unserer Webplatform. Halten Sie sich an das dort vorgegebene Format, da wir den Zeitbedarf mit einem Python-Skript automatisch statistisch auswerten. Die Zeitangabe `7.5 h` steht dabei für 7 Stunden 30 Minuten.

[3] https://de.wikipedia.org/wiki/Gleitkommazahl#Pr%C3%BCfung_auf_Gleichheit