Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Dr. Daniel Büscher, Hannes Saffrich
Wintersemester 2019

Universität Freiburg
Institut für Informatik

Übungsblatt 14 – Lösungen

Abgabe: Montag, 03.02.2020, 9:00 Uhr morgens

Aufgabe 14.1 (Faltung; Datei: `reduce.py`; Punkte: 2+1+1+1+1+2)

In der Vorlesung wurde die Faltungsfunktion `functools.reduce` besprochen. In dieser Aufgabe sollen Sie `functools.reduce` selbst implementieren und anschließend verwenden um einige Listenoperation zu definieren.

(a) Implementieren Sie die Funktion `my_reduce(f, xs, x0)`, wie in der Vorlesung beschrieben, d.h. sodass

- `my_reduce(f, [], x0)` sich verhält wie `x0`
- `my_reduce(f, [x1], x0)` sich verhält wie `f(x0, x1)`
- `my_reduce(f, [x1, x2], x0)` sich verhält wie `f(f(x0, x1), x2)`
- `my_reduce(f, [x1, x2, x3], x0)` sich verhält wie `f(f(f(x0, x1), x2), x3)`
- usw.

(b) Implementieren Sie die folgenden Funktionen durch jeweils einen einzelnen Aufruf von `my_reduce`, d.h. nach folgendem Muster:

```
def function(args):
    return my_reduce((lambda acc, x: ...), ..., ...)
```

Sollten Sie Aufgabenteil (a) nicht gelöst haben, können Sie die `reduce`-Funktion aus dem Modul `functools` importieren und verwenden.

- Die Funktion `my_or` verknüpft eine Liste von boolschen Werten mit `or`. Für die leere Liste soll `my_or` dabei `False` zurückgeben, sodass `my_or(xs1 + xs2)` stets das Gleiche liefert wie `my_or(xs1) or my_or(xs2)`. Ohne `my_reduce` kann diese Funktion wie folgt definiert werden:

```
def my_or(xs):
    acc = False
    for x in xs:
        acc = acc or x
    return acc
```

- Die Funktion `my_and` verknüpft eine Liste von boolschen Werten mit `and`. Für die leere Liste soll `my_and` dabei `True` zurückgeben, sodass `my_and(xs1 + xs2)` stets das Gleiche liefert wie `my_and(xs1) and my_and(xs2)`. Ohne `my_reduce` kann diese Funktion wie folgt definiert werden:

```
def my_and(xs):
    acc = True
    for x in xs:
        acc = acc and x
    return acc
```

– Auch die `map`-Funktion kann durch `my_reduce` implementiert werden. Ohne `my_reduce` kann diese Funktion wie folgt definiert werden:

```
def my_map(f, xs):
    acc = []
    for x in xs:
        acc = acc + [f(x)]
    return acc
```

– Die `my_unique`-Funktion entfernt Duplikate aus einer bereits sortierten Liste. Ohne `my_reduce` kann diese Funktion wie folgt definiert werden:

```
def my_unique(xs):
    acc = []
    for x in xs:
        if not acc or acc[-1] != x:
            acc = acc + [x]
    return acc
```

(c) Schreiben Sie für `my_and`, `my_or`, `my_map` und `my_unique` jeweils zwei Unittests.

Lösung:

```
def my_reduce(f, xs, x0):
    acc = x0
    for x in xs:
        acc = f(acc, x)
    return acc


def my_or(xs):
    return my_reduce(lambda acc, x: acc or x, xs, False)

def my_and(xs):
    return my_reduce(lambda acc, x: acc and x, xs, True)

def my_map(f, xs):
    return my_reduce(lambda acc, x: acc + [f(x)], xs, [])

def my_unique(xs):
    return my_reduce(lambda acc, x: acc + [x] if not acc or acc[-1] != x else acc, xs, [
```

```
def test_or_1():
    assert my_or([]) == False

def test_or_2():
    assert my_or([False, True, False]) == True

def test_and_1():
    assert my_and([]) == True

def test_and_2():
    assert my_and([False, True, False]) == False

def test_map_1():
    assert list(my_map(lambda x: x+1, [])) == []

def test_map_2():
    assert list(my_map(lambda x: x+1, range(0,5))) == list(range(1,6))

def test_unique_1():
    xs = list(range(20))
    assert my_unique(xs) == xs

def test_unique_2():
    assert my_unique([1,1,2,4,4,4,5]) == [1,2,4,5]
```

Aufgabe 14.2 (Exceptions; `suppress.py`; Punkte: 4)

Schreiben Sie eine Funktion `suppress(f, ignore: tuple)`, welche eine parameterlose Funktion `f` und ein Tuple an Exceptions `ignore` als Argumente erhält und eine neue Funktion `g` zurückgibt. `g` soll sich dabei identisch zu `f` verhalten, solange beim Aufruf keine Exception aus `ignore` auftritt. Tritt während des Aufrufs `f()` eine solche Exception auf, so soll diese beim Aufruf `g()` ignoriert und `None` zurückgeben werden. Beispiel:

```
>>> from functools import partial
>>> def foo(n: int) -> int:
...     return 35 // n
...
>>> assert suppress(partial(foo, 1), ())() == 35 == foo(1)
>>> suppress(partial(foo, 0), (ZeroDivisionError))()
>>> suppress(partial(foo, 0), ())()
Traceback (most recent call last):
  ...
  File "suppress.py", line 26, in <lambda>
    suppress(lambda: 3 / 0, ())()
```

```
ZeroDivisionError: division by zero
```

Hinweis: In Python können Funktionen auch verschachtelt, also innerhalb von anderen Funktionen, definiert werden. Beispiel:

```
>>> def foo(s):
...     def bar():
...         print("hi", s)
...     return bar
...
>>> b = foo("dude")
>>> b()
hi dude
```

Im Beispiel wird die Variable `s` Teil der Closure von `bar`. Das bedeutet wenn `bar` von `foo` zurückgegeben wird, dann bleibt `s` innerhalb von `bar` weiter gültig. Bei jedem Aufruf von `foo` wird sozusagen eine neue `bar`-Funktion erstellt, die sich das jeweilige `s` merkt.

Das Gleiche funktioniert auch mit Lambdas:

```
>>> def foo(s):
...     return lambda: print("hi", s)
...
>>> b = foo("dude")
>>> b()
hi dude
```

Bei verschachtelten Funktionen können jedoch, wie bei normalen Funktionen auch, mehrere Anweisungen hintereinander geschrieben werden, was bei Lambdas nicht möglich ist, da Lambdas aus einem einzelnen Ausdruck bestehen. In dieser Aufgabe empfiehlt es sich also verschachtelte Funktionen zu verwenden.

Lösung:

```
def suppress(f, ignore: tuple):
    def wrapped_f():
        try:
            return f()
        except ignore:
            return None
    return wrapped_f
```

Aufgabe 14.3 (Comprehensions; Datei: `comprehensions.py`; Punkte: 3+3)

Implementieren Sie die folgenden Funktionen mit List-Comprehensions und schreiben Sie jeweils einen Unittest:

(a) Ein pythagoreisches Tripel (x, y, z) besteht aus drei natürlichen Zahlen x, y und z, so dass $x^2+y^2 = z^2$. Schreiben Sie eine Funktion `pythagorean_triples(n: int) -> list`,

welche alle pythagoreischen Tripel mit Hilfe von List-Comprehensions berechnet und als Liste zurückgibt, so dass $x \leq n$, $y \leq n$ und $z \leq n$.

(b) Schreiben Sie eine Funktion `cookable(xs: list) -> dict`, welche eine Liste `xs` an Zutaten (jede Zutat ist ein String) als Argument erhält und alle Rezepte aus einem Dictionary `recipes`, welche mit den gegebenen Zutaten kochbar sind, als Dictionary zurückgibt. `recipes` ist wie folgt definiert:

```
recipes = {
    "Sushi":             ["Fisch", "Reis", "Nori"],
    "Sashimi":           ["Fisch", "Reis"],
    "Pfannkuchen":       ["Mehl", "Ei", "Milch"],
    "Burger":            ["Brötchen", "Rind"],
    "Burger TS":         ["Brötchen", "Rind", "Tomate", "Salat"],
    "Cheese Burger":     ["Brötchen", "Rind", "Tomate", "Käse"],
    "Gemischter Salat":  ["Salat", "Tomate", "Gurke"]
}
```

Achtung: Ihre Funktionsdefinition soll außer einer `return`-Anweisung keine weiteren Zeilen enthalten. Innerhalb des `return`-Statements dürfen/sollen allerdings ein oder mehrere (List-/Dict-/Generator-)Comprehensions benutzt werden. Beispiele:

```
>>> cookable(["Brötchen", "Tomate", "Gurke", "Salat", "Rind", "Brötchen"])
{'Burger': ['Brötchen', 'Rind'],
 'Burger TS': ['Brötchen', 'Rind', 'Tomate', 'Salat'],
 'Gemischter Salat': ['Salat', 'Tomate', 'Gurke']}
>>> cookable(["Fisch", "Reis", "Tomate"])
{'Sashimi': ['Fisch', 'Reis']}
```

Lösung:

```
def pythagorean_triples(n: int) -> list:
    return [(a, b, c) for a in range(n+1)
                      for b in range(n+1)
                      for c in range(n+1) if a ** 2 + b ** 2 == c ** 2]


def test_pythagorean_triples():
    triples = pythagorean_triples(100)
    for (a, b, c) in triples:
        assert a ** 2 + b ** 2 == c ** 2


recipes = {
    "Sushi":             ["Fisch", "Reis", "Nori"],
    "Sashimi":           ["Fisch", "Reis"],
    "Pfannkuchen":       ["Mehl", "Ei", "Milch"],
```

```
    "Burger":            ["Brötchen", "Rind"],
    "Burger TS":         ["Brötchen", "Rind", "Tomate", "Salat"],
    "Cheese Burger":     ["Brötchen", "Rind", "Tomate", "Käse"],
    "Gemischter Salat":  ["Salat", "Tomate", "Gurke"]
}


# Three variants of `cookable`:

def cookable(xs: list) -> dict:
    return dict((k, v) for k, v in recipes.items() if all([x in xs for x in v]))

def cookable2(xs: list) -> dict:
    return dict((k, v) for k, v in recipes.items() if set(v) <= set(xs))

def cookable3(xs: list) -> dict:
    return {k: v for k, v in recipes.items() if set(v) <= set(xs)}


def test_cookable():
    for c in [cookable, cookable2, cookable3]:
        assert c(["Brötchen", "Tomate", "Gurke", "Salat", "Rind",
            "Brötchen"]) == {
            'Burger': ['Brötchen', 'Rind'],
            'Burger TS': ['Brötchen', 'Rind', 'Tomate', 'Salat'],
            'Gemischter Salat': ['Salat', 'Tomate', 'Gurke']
        }
        assert c(["Fisch", "Reis", "Tomate"]) == {
            'Sashimi': ['Fisch', 'Reis']
        }
```

Aufgabe 14.4 (Erfahrungen; Datei: `erfahrungen.txt`; Punkte: 2)

Legen Sie im Unterverzeichnis `sheet14` eine Textdatei `erfahrungen.txt` an. Notieren Sie in dieser Datei kurz Ihre Erfahrungen beim Bearbeiten der Übungsaufgaben (Probleme, Bezug zur Vorlesung, Interessantes, benötigter Zeitaufwand, etc.).