Institut für Numerische und Angewandte Mathematik
FB Mathematik und Informatik der Universität Münster
Prof. Dr. Mario Ohlberger
M.Sc.-Math. Julia Brunken

23. September 2016

Übung zum Kompaktkurs

Einführung in die Programmierung zur Numerik mit Python

Sommersemesterferien 2016 — Blatt 5

Aufgabe 1 (Lambda-Ausdrücke)

Schreiben Sie einen interaktiven Funktionsplotter. Das Programm soll von der Kommandozeile Benutzereingaben in Form von Python-Ausdrücken einlesen und diese mittels Lambda-Ausdrücken in Funktionen umsetzen. Lesen Sie weiter ein Intervall `[a,b]` und eine Stützstellenzahl `N` ein und visualisieren Sie die eingegebene Funktion mittels `matplotlib`. Schreiben Sie dazu eine Klasse `Funktion` mit folgenden Eigenschaften:

- Objekte der Klasse sollen durch `funktion = Funktion(ausdruck, glob)` angelegt werden können, wobei `ausdruck` ein String ist und `glob` (das vom Typ `dict` ist) im Ausdruck verwendete globale Variablen angibt, siehe Hinweis.
- Objekte der Klasse sollen durch `funktion(x)` an einer Stelle $\texttt{x} \in \mathbb{R}$ ausgewertet werden können.
- Stellen Sie eine Methode `plotten(a,b,N)` bereit, welche die repräsentierte Funktion über das Intervall `[a,b]` plottet und dabei `N` Stützpunkte verwendet.

Hinweise: *Einen Ausdruck in Form eines Strings können Sie mit der Funktion* `eval` *auswerten lassen. Wollen Sie in einem Ausdruck Funktionen aus einem Modul wie zum Beispiel* `numpy` *verwenden, so können Sie* `eval` *dieses Modul als globale Variable übergeben. Schauen Sie sich dazu die Hilfe von* `eval` *an!*

Aufgabe 2 (Generatoren)

(a) Schreiben Sie einen Generator `permutationen`, der zu einer gegebenen Menge von Elementen nach und nach sämtliche Permutationen dieser Elemente ausgibt. Beachten Sie dazu, dass man Generatoren rekursiv verwenden kann, d.h. die Generatorfunktion `permutationen` kann aus sich selbst heraus mit anderen Werten aufgerufen werden. Verwenden Sie als Argument der Funktion den Datentyp `set`.

(b) Betrachten Sie nun das folgende Mini-Sudoku: Es gilt, ein 4×4 Felder großes Gitter so mit den Ziffern von 1 bis 4 zu füllen, dass in jeder Zeile, jeder Spalte und jedem der vier

Unterquadrate der Größe 2×2 alle vier Ziffern je genau einmal vorkommen. Schreiben Sie – unter Verwendung des Generators aus Teil (a) – ein Programm, das nach und nach sämtliche gültigen 4×4-Sudokus erzeugt. Geben Sie die Anzahl der gefundenen Sodukus aus. *(Kontrollergebnis: Es sind genau 288)*

Aufgabe 3 (Dekoratoren)

Wir haben gesehen, dass man Funktionen mit sogenannten *Dekoratoren* ausstatten kann. Neben den bereits am Dienstag erwähnten Dekoratoren `@staticmethod` und `@classmethod` kann man auch eigene Dekoratoren etwa in Form spezieller Klassen erstellen. Die Funktionsweise solcher Dekoratoren ist es, dass sie mit einer gegebenen Funktion als Argument aufgerufen werden und eine andere, sozusagen *dekorierte* Funktion zurückgeben. Neben dem in der Vorlesung gegebenen Beispiel könnte man auch einen Dekorator implementieren der dafür sorgt, dass vor jedem Funktionsaufruf das übergebene Argument ausgegeben wird. Das geht so:

```
class AusgabeDekorator:
  def __init__(self):
    self.funktion = None

  def dekorierte_funktion(self, argument):
    print('argument = {}'.format(argument))
    return self.funktion(argument)

  def __call__(self, funktion):
    self.funktion = funktion
    return self.dekorierte_funktion

@AusgabeDekorator()
def quadrieren(a):
  return a**2

quadrieren(1) # Ausgabe: argument = 1
```

(a) Schreiben Sie in analoger Weise einen `CacheDekorator` der bei jedem Funktionsaufruf überprüft, ob die Funktion bereits einmal mit dem selben Argument aufgerufen wurde und wenn ja diesen Wert – ohne Neuberechnung! – liefert. Überlegen Sie dazu zunächst, welche Datenstruktur für diesen Cache geeignet ist. Geben Sie auch eine Meldung aus die anzeigt, ob der Wert berechnet oder aus dem Cache geholt wurde.

(b) Testen Sie die Implementierung an einer Funktion `fibonacci(n)`, welche die n-te Fibonacci-Zahl zurückgibt und verwenden Sie den `CacheDekorator`.