D-MATH
Dr. Vasile Gradinaru
Dr. Adrian Montgomery Ruf

Serie 1

Best before: Di. 25.2 / Mi. 26.2, in den Übungsgruppen
Koordinatoren: Adrian Montgomery Ruf, HG G 54.1, `adrian.ruf@sam.math.ethz.ch`
Webpage: `http://metaphor.ethz.ch/x/2020/fs/401-1662-10L/#exercises`

1. *Erste Schritte in* `Python`

Diese Serie ist eine Einführung ins wissenschaftliche Rechnen in Python. Wir werden lernen effizient, lesbaren Python code zu schreiben.

Es gibt auch andere Tutorials. Welches ihr durcharbeitet ist nicht wichtig. Eigentliche Aufgaben gibt es in Aufgabe 4, 5, 7. Wenn ihr diese lösen könnt, dann reichen eure Python-Kentnisse fürs Erste.

- Das Tutorial der letzten Jahre `http://www.sam.math.ethz.ch/~aldabrow/teaching/2016NM/PythonTutorial/index.html`
- Scipy lecture notes: `http://www.scipy-lectures.org`

Die Syntax für eine Funktion ist:

```
def␣function_name(argument):
␣␣␣␣"""Dies␣ist␣eine␣'docstring'.

␣␣␣␣Die␣docstring␣beschreibt␣was␣die␣Funktion␣macht.
␣␣␣␣"""

␣␣␣␣print(argument)
␣␣␣␣return␣42.0

def␣another_function():
␣␣␣␣"""Lest␣den␣Abschnitt␣ueber␣gute␣Kommentare."""

␣␣␣␣#␣Gibt␣-42.0␣zurueck.␣␣<--␣Uhm,␣ja.␣Offensichtlich,
␣␣␣␣#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣aber␣wiso␣-42.0??
␣␣␣␣return␣-42.0
```

In Python beginnt ein neuer Codeblock immer mit einem `:`. Der darauf folgende Block *muss* eingerückt werden. Der Standard sagt man soll mit 4 Leerschlägen einzurücken. Der Block endet mit dem Indentlevel.

Funktionen können und sollten mit einer docstring kommentiert werden. Die erste Zeile der docstring ist ein kurzer präziser Satz. Falls mehr gesagt werden muss, folgt eine leere Zeile und danach beliebig viel Text.

Bitte wenden!

Kommentare sind äusserst wichtig aber nicht alle Kommentare sind nützlich. Man muss trainieren gute Kommentare zu schreiben. Nicht immer ist ein Kommentar die beste Lösung um den Code leserlicher zu machen. Manchmal sollte man den Code umschreiben, damit er leichter zu verstehen ist. Eventuell braucht es eine neue Funktion, oder einen besseren Funktionsnamen. Dies wird deutlich beschrieben in Robert C. Martin, *Clean Code.*

Wir wollen den Code ausprobieren. Dazu speichern wir den Code in einer Datei `functions.py` und testen den code in `ipython`.

```
$ ipython3 functions.py
```

Nichts geschieht.

```
$ ipython3 -i functions.py
```

Diesmal passiert fast nichts, aber jetzt seid ihr in IPython drin. Ihr könnt direkt Code auprobieren

```
In [1]: function_name("this is a string, i.e. text")
this is a string, i.e. text
Out[1]: 42.0
```

Modifiziert `function_name`, z.B. ersetzt `42.0` mit `43.0`.

```
In [2]: function_name(3)
3
Out[2]: 42.0
```

Seltsam, die Funktion gibt immer noch `42.0` zurück. Man kann die Funktionen neuladen indem man das script nochmals ausführt:

```
In [3]: run functions.py
```

Beendet IPython mit `Ctrl+d` oder `exit`. Überigens kann ein laufendes Script mit `Ctrl+c` abgebrochen werden.

2. *Arrays erstellen in* `Python`

Die wichtigste Datenstruktur für numerisches Programmieren ist das mehrdimensionale Array. Das Paket dafür heisst `numpy`.

Für diese Aufgabe werden wir Python interaktiv benutzen. Startet `ipython3`.

Kleine strukturlose Arrays erstellt man so:

```
In [1]: import numpy as np
In [2]: x = np.array([19.0, 12.0, -3.0])
In [3]: A = np.array([[1.0, 0.0], [1.0, 4.0]])
```

Arrays mit lauter Nullen

Siehe nächstes Blatt!

```
In [4]: A = np.zeros((3, 4))
In [5]: A
In [6]: A.shape
```

In der ersten Aufgabe haben wir Docstrings gesehen. Wir können den Docstring einer beliebigen Funktion aufrufen:

```
In [7]: ?np.empty
```

Ganz praktisch, wenn man keinen Internetzugang hat.

Dies erstellt ein eindimensionales Array aufeinander folgenden Zahlen:

```
In [8]: ?np.arange
In [9]: x = np.arange(4)
In [10]: x = np.arange(3, 10)
```

Beachtet, dass die letzte Zahl nicht mit dabei ist, die erste aber schon. Es sind also 4 Elemente in `np.arange(4)`.

```
In [11]: np.arange(4).size
```

Ein uniformes Gitter in einer Dimension erstellt man am einfachsten mit:

```
In [12]: ?np.linspace
In [13]: np.linspace(0.0, 1.0, 5)
```

Die Funktion `np.meshgrid` eignet sich für rechteckige Gitter in zwei Dimensionen. Ein 3×5 Gitter auf $[1, 2] \times [3, 4]$ erstellt man so:

```
In [14]: x = np.linspace(1, 2, 3)
In [15]: y = np.linspace(3, 4, 5)
In [16]: X, Y = np.meshgrid(x, y)
```

3. *Punktweise Operationen auf Arrays*

Kann man den Sinus eines Arrays berechnen? Eigentlich nicht, denn was soll das sein? Aber,

```
In [1]: x = np.random.random((2, 3))
In [2]: sinx = np.sin(2.0*np.pi*x)
```

funktioniert. Numpy interpretiert "Sinus eines Arrays" als elementweise den Sinus berechnen. Die Denkweise in Numpy ist, dass jede Funktion die für Skalare definiert ist elementweise auf Arrays angewandt wird.

```
In [3]: for i in range(x.shape[0]):
   ...:     for j in range(x.shape[1]):
   ...:         print(np.sin(2.0*np.pi*x[i,j]) == sinx[i,j])
```

Bitte wenden!

Solche explizite Schlaufen über alle Elemente in `x` sind ein Problem. Wer an die Informatikvorlesung denkt, erinnert sich, dass dort fast sofort Datentypen, `int, double,` etc., behandelt wurden. Bisher kamen keine Datentypen vor. Python ist dynamically typed, es erkennt selber was der richtige Datentyp sein sollte. Dies kostet Laufzeit. Zuviel Laufzeit, wenn das Array grösser ist als ein paar 100 oder 1000 Elemente. Numpy interpretiert `np.sin(x)` elementweise und implementiert die Doppelschlaufe in `C`. Deswegen ist `np.sin(x)` ähnlich schnell wie eine Implementation in `C`. Wenn man eine Operation *vektorisiert*, meint man, dass die explizite Doppelschlaufe vermieden und stattdessen (optimierten) `C` code aufrufen wird.

Numpy implementiert alle wichtigen Funktionen, `np.sin`, `cos`, `exp`, `+`, `-`, etc. Besondere Beachtung verdienen Multiplication und Potenz, beide sind elementweise. Die Operation `*` bedeutet immer elementweise Multiplikation, nie ein Matrix-vector Produkt. In Python gibt es ein Symbol für potenzieren `**`.

```
In [4]: 2**np.arange(4, 10)
In [5]: np.arange(4, 7)**np.arange(3)
```

4. *Sequenzen*

Schreiben Sie das Programm `prog2.py` welches $k = 2i + 1$ für $i = 0, 1, \ldots, 10$ untereinander im Terminal ausgibt.

5. *Fakultät*

Schreiben Sie das Programm `prog3.py` welches für ein vorgegebenes $n \in \mathbb{N}$ die Fakultät $n!$ unter Verwendung einer for-Schleife berechnet und im Terminal ausgibt. Tipp: `input(), int()`.

6. *Wurzel einer positiven Zahl*

Das Newton-Verfahren zur Berechnung der Wurzel einer positiven Zahl a ist gegeben durch

$$x_k = \frac{1}{2}\left(x_{k-1} + \frac{a}{x_{k-1}}\right) \qquad x_0 = 2a$$

Schreiben Sie das Programm `prog4.py`, welche $\sqrt{a}$ bis auf eine gewisse Genauigkeit $|x_k - x_{k-1}| \leq \tau$ berechnet, und anschliessend die Approximation an $\sqrt{a}$ und die dazu benötige Anzahl an Iterationen k im Terminal ausgibt. Probieren Sie Ihr Programm mit $\tau = 10^{-5}$ und $a = 7$. Tipp: Es kann nützlich sein, eine *while*-Schleife zu verwenden:

```
while Bedingung :
Anweisungsblock
```

und die Funktion `abs`, die unter `numpy` verfügbar ist.

7. *Fibonacci Zahlen*

Schreiben Sie das Programm `prog5.py` welches die ersten n Fibonacci Zahlen

$$F_k = \begin{cases} 0 & k = 1 \\ 1 & k = 2 \\ F_{k-1} + F_{k-2} & k \geq 3 \end{cases}, \quad k \in \mathbb{N}$$

Siehe nächstes Blatt!

im Terminal ausgibt. Dabei soll n im Terminal eingegeben werden. Falls für die Eingabe $n \leq 0$ gilt, soll eine Fehlermeldung ausgegeben und das Programm beendet werden.

8. *Approximation durch Serien*

Schreiben Sie das Programm `prog6.py`, welche das kleineste n bestimmt, für welches

$$|2 - \sum_{i=0}^{n} 2^{-i}| \leq 0.01$$

gilt.

9. *Volumen und Oberfläche einer Kugel*

Schreiben Sie das Programm `prog8.py`, welches folgendes macht:

- Im Terminal wird der Benutzer gebeten einen Radius einzugeben (dieser wird entgegen genommen),
- Unter Verwendung des eingegeben Radius wird das Volumen und die Oberfläche der zugegörigen Kugel berechnet und anschließend im Terminal ausgegeben.

Schreiben Sie dazu die Funktionen `Volumen` und `Oberflaeche` und definieren Sie $\pi \approx 3.1415926$ als Konstante.

10. *Die Julia Menge.*

Sei $f(z) = z^2 + 0.279$ und $D = [-1, 1]^2 \subset \mathbb{C}$. Für 1000×1000 Punkte $z \in D$ berechne

$$g(z) = \max\{n \in \mathbb{N} : |f^n(z)| < 100\}. \tag{1}$$

Plotte $\log g$ mittels

```
import numpy as np
import matplotlib.pyplot as plt

...

plt.contourf(np.real(z), np.imag(z), np.log(n_times_finite), 100)
```

Das Problem ist so gewählt, dass eine einfache, elegante Lösung existiert. Falls irgendein Schritt kompliziert oder mühsam wird sollte man in die Referenzlösung schauen.

Schrittweise Anleitung:

(a) Implementiere f.

(b) Berechne ein 1D Gitter auf $[0, 1]$.

(c) Berechne ein 2D Gitter auf $[0, 1]^2 \subset \mathbb{R}^2$.

Bitte wenden!

(d) Berechne ein Gitter für $D \subset \mathbb{C}$. Verwende die Abbildung $z = x + \mathbf{i}y$. Die imaginäre Einheit ist in Python `1.0j`.

(e) Für $k < n$, berechne $f(f^{k-1}(z))$ und überprüfe ob der Wert endlich, d.h im Betrag kleiner als 100, ist. Numpy macht dies leicht: `y[np.abs(x) < 100] += 1.0`. Wähle $n = 60$ oder so.

(f) Plotte $\log g$.

11. *Stencil Operationen in Numpy*

In vielen Algorithmen findet man

$$y_i = f(x_{i-1}, x_i, x_{i+1}), \qquad i = 1, \cdots, n. \tag{2}$$

Hier sind uns die Spezialfälle $i = 1$ und $i = n$ egal.

Damit wir eine solche Operation implementieren können brauchen wir "slices".

Öffnet wieder ein interaktives IPython.

```
In [1]: import numpy as np
In [2]: A = np.arange(3*4).reshape((3, 4))
In [3]: A
In [4]: A[:,0]
In [5]: A[1,:]
In [6]: A[1:,:]
In [7]: A[1:-1,:]
```

Das reicht um einen 3-Punkte Stencil zu implementieren.

```
# -*- coding: utf-8 -*-

import numpy as np

def stencil(y, dx):
    """Zeigt wie man einen simplen Stencil implementiert.

       Berechnet die approximative zweite Ableitung der gegebenen
       Punktwerte.

          y : array_like
              Dies sind die Punktwerte der Funktion von der die
              zweite Ableitung berechnet wird.

         dx : double
              Gitterkonstante, delta x = x[1] - x[0].
    """

    # Sehr kurze, logisch äusserst eng verknüpfte Zuweisungen
    # darf man auf eine Zeile schreiben.
    left, middle, right = y[:-2], y[1:-1], y[2:]
    return ((right - middle)/dx - (middle - left)/dx)/dx
```

Siehe nächstes Blatt!

```
x = np.linspace(0, 1, 1000)
y = np.sin(2.0*np.pi*x)
stencil(y, x[1] - x[0])
```

Im oberen Beispiel sind `left, middle, right` keine Kopien von `y` sondern eine view. Das heisst, `left` sieht den gleichen Speicher wie `y`, jedoch ist das Array kürzer. `middle` zeigt auch wieder auf den gleichen Speicher, aber die Indizes sind verschoben.

(a) Im Beispiel wird die Operation `(b - a)/dx` drei mal verwendet. Eigentlich könnte man das mit weniger Wiederholungen schreiben. Implementiert `first_derivative(y, dx)`. Schreibt eine neue Funktion `second_derivative`, welche `first_derivative` zweimal aufruft. Testet `second_derivative` in dem ihr die Werte mit `stencil` vergleicht. Tipp: `np.abs, np.all`.

(b) Zählt die Anzahl FLoating-point OPerations (FLOP) in `stencil` und `second_derivative` und überlegt ob man dies weiter optimieren könnte.

(c) Messt die Laufzeit der drei Implementationen. Tipp: `timeit`.

(d) Explizite Schlaufen seien langsam in Python. Stimmt das überhaupt? Falls ja, wie viel langsamer?

12. *Operation entlang bestimmter Achsen*

Die Dimension eines Arrays ist die Anzahl Achsen die es besitzt. Ein paar Beispiele:

```
In [1]: x = np.array(34)  # Eine Achse, eindimensional.
In [2]: x = np.array((34,3))  # Zwei Achsen, zwei-dimensional.
In [3]: x[:,0]   # ist ein slice entlang der ersten Achse.
```

Die Funktionen `np.max`, `np.min`, `np.sum` sind Reduktionen. Als Argument nehmen sie ein Array und geben einen Skalar zurück.

Es ist nicht immer gewünscht, dass die Reduktion über das ganze Array geschieht, manmal will man die Reduktion nur entlang einer bestimmen Achse ausführen. Die meisten Reduktionen in Numpy akzeptieren ein Argument `axis`. Wir schauen uns ein Beispiel an:

```
In [4]: x = np.random.random((2, 3, 1000))
In [5]: np.max(x, axis=2)
In [6]: np.min(x, axis=-1)
```

13. *Broadcasting*

Das letzte Thema welches besprochen werden muss ist *broadcasting*.

Wer `?np.ndarray.shape` und `?np.ndarray.reshape` noch nicht durchgelesen und ausprobiert hat, sollte dies nun tun.

```
In [1]: x = np.arange(4*5).reshape((4,-1))
In [2]: y = np.arange(4).reshape((4, 1))
In [3]: x + y
```

Bitte wenden!

Dies ist erstaunlich, weil `A` und `x` nicht die gleiche Form (shape) haben.

Falls zwei Arrays `x` und `y` die gleiche Dimension haben und die gleiche Anzahl Elemente entlang jeder Achse mit mehr als einem Element haben, dann wird broadcasting angewandt. Dies bedeutet, dass entlang Achsen mit einem Element das Array mit Kopien von sich selbst aufgefüllt wird, bis die Form zum anderen Array passt.

```
[[1, 2, 3]] + [[1, 2, 3],  -->  [[1, 2, 3], + [[1, 2, 3],
               [2, 3, 4]]        [1, 2, 3]]    [2, 3, 4]]

[[1, 2, 3]] + [[1],  -->  [[1, 2, 3], + [[1, 1, 1],
               [2]]       [1, 2, 3]]    [2, 2, 2]]
```

Falls die zwei Arrays eine underschiedliche Dimension haben, dann wird zuerst von links mit Achsen mit einem Element aufgefüllt und dann versucht broadcasting anzuwenden. Auch dazu ein Beispiel:

```
[ 1, 2, 3] + [[1],  -->  [[1, 2, 3]], + [[1],
              [2]]                       [2]]

                    -->  [[1, 2, 3], + [[1, 1, 1],
                          [1, 2, 3]]    [2, 2, 2]]
```

Aufgepasst, die leeren Achsen werden links hinzugefügt:

```
  [ 1, 2, 3] + [[1, 2]]  -->  [[1, 2, 3]] + [[1, 2]] --> Exception
Shape: (3,) + (1, 2)     -->       (1, 3) + (1, 2)
```

Oft wird `reshape` benutzt um leere Achsen einzufügen, damit broadcasting klappt. In diesem Fall kann `np.newaxis` nützlich sein.

```
In [4]: A = np.arange(2*3*4).reshape(2, 3, 4)
In [5]: B = np.arange(2*4).reshape(2, 4)
In [6]: A + B
In [7]: A + B.reshape((A.shape[0], 1, A.shape[2]))
In [8]: A + B[:,np.newaxis,:]
```

(a) In Aufgabe 4, ersetzt `meshgrid` durch geeignetes broadscasting im Schritt (d).

14. `PDB` *der Python Debugger* [Optional]

Den Debugger muss man nicht gleich am ersten Tag kennen lernen. Wer schon genug gelernt hat für einen Tag kann dieses Kapitel ein andermal durcharbeiten.

PDB ist ein Debugger. Ein Debugger hilft Code zu debuggen. Er ermöglicht es Zeile für Zeile zu verfolgen, wie der Code ausgeführt wird und welche Werte in beliebigen Variabeln gespeichert sind.

Lasst euch nicht vom archaischen Interface abschrecken. PDB ist besser als `printf` debugging.

Ein cheat-sheet ist empfehlenswert.

Siehe nächstes Blatt!

15. *Fortgeschrittenes IPython* [Optional]

IPython verfügt über interessante profiling tools. Dieser post erklärt recht gut:
http://pynash.org/2013/03/06/timing-and-profiling/.

Jupyter ermöglicht ein Mathematica ähnliches Interface. Bedenkt, dass ihr an der Prüfung Pythonskripte schreiben müsst, ihr werdet Jupyter nicht benutzen können. Aber ihr lernt nicht nur für die Prüfung.

YT ist eine Plotting- und Datenanalyse-Bibliothek für Physiker. Ein wesentlicher Unterschied zu matplotlib ist, dass es Einheiten kennt. Ausserdem kann es den Output einiger komplizierter 3D-Simulationen lesen. Das kann für eine Bachelorarbeit durchaus nützlich sein.