Mathematik für Informatiker - Übungsblatt 0

Python, Jupyter Notebooks und Grundlagen der Programmierung

Die Vorlesung *Mathematik für Informatiker* folgt in weiten Teilen dem Buch *Konkrete Mathematik (nicht nur) für Informatiker* von Edmund Weitz [1]. Dieses Buch verfolgt den Ansatz, dass Informatiker*innen in ihrem späteren Berufsleben mehr angewandte (konkrete) Mathematik benötigen wie „reine“ Mathematik, welche Mathematik der Mathematik wegen betreibt. Anwendungswerkzeug der Wahl: Programmieren! Sie lernen in meiner Vorlesung die mathematischen Begriffe, Methoden und vor allem deren Anwendungen, indem sie diese in Computerprogramme „übersetzen“. Wir werden dafür (wie auch Herr Weitz) Python und Jupyter Notebooks verwenden.

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. [2]

The **Jupyter Notebook** is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. [3]

Die Installation von Python und Jupyter ist recht einfach - vorausgesetzt sie verwenden dafür wie von Herr Weitz vorgeschlagen Docker:

http://weitz.de/install/

Falls sie lieber Python direkt auf ihrem Betriebssystem laufen lassen möchten ist dies natürlich auch möglich... am einfachsten unter Linux ;-)... ist ihr Grundbetriebssystem Windows (so wie bei mir), können sie entweder mit einer virtuellen Maschine (z.B. VMWare) oder mit dem Windows Subsystem for Linux (WSL) arbeiten. Öffnen sie dann auf Linux ein Terminal und installieren sie Miniconda wir hier beschrieben:

https://docs.conda.io/projects/conda/en/latest/user-guide/install/linux.html

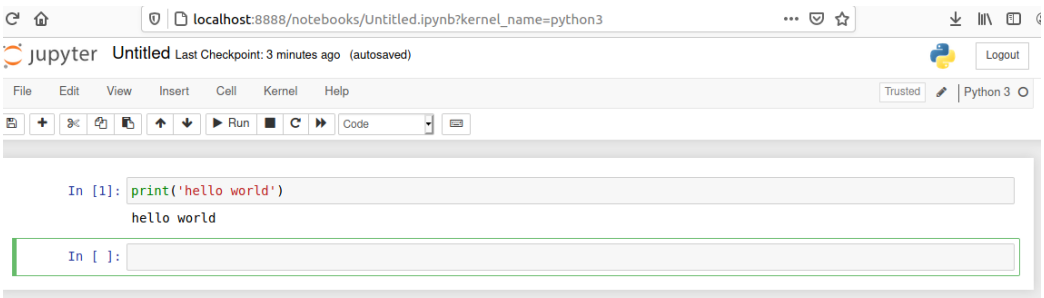
Nun sollten sie (nach Neustart des Terminals) mit

```
conda install jupyterlab
```

jupyterlab installieren und danach mit

```
jupyter notebook
```

ihr erstes Jupyter-Notebook erstellen (new-Button) und ihr erstes Python-Programm (natürlich die Ausgabe von Hello World!) damit schreiben können:



Wenn sie was ähnliches wie im obigen Bild auf ihrem Bildschirm sehen, haben sie den ersten Schritt zum Erlernen der linearen Algebra und der Grundlegenden der Analysis mittels Programmieren geschafft → Herzlichen Glückwunsch!

Vielleicht fragen sie sich, warum ich hier nicht nur einen Weg für die Installation von Jupyter angegeben habe, der dann für alle funktioniert. Nun, das liegt zum Einen daran, dass ich ihr „Setup“ (Rechner, Betriebssystem, ...) nicht kenne, zum Anderen darin, dass sie sich zu Informatiker*innen ausbilden lassen und sich möglichst früh daran gewöhnen sollten, dass es meist nicht „den einen richtigen Weg“ zur Lösung einer Aufgabenstellung gibt, sondern mehrere Wege, welche alle zu einem akzeptablen Ergebnis führen. Dies gilt nicht nur für die Auswahl von Werkzeugen (so wie hier), sondern insbesondere auch bei der Architektur von Software bzw. deren Implementierung.

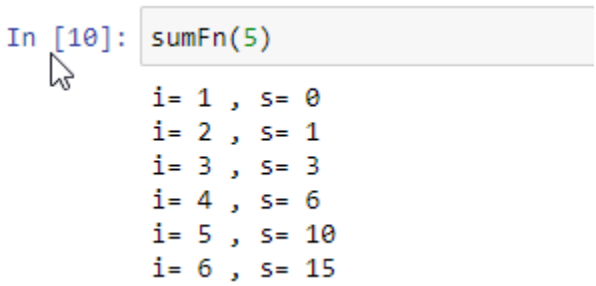
Nun benötigen wir noch einige programatische Grundlagen in Python die sie schon aus anderen Vorlesungen kennen sollten. Falls nicht, auch nicht schlimm! Dann haben sie die Chance sie hier zu lernen.

Funktionen und Schleifen

Lassen sie uns zuerst eine Funktion schreiben, welche die Summe der ersten n natürlichen Zahlen berechnet. Passen sie dazu folgenden Python-Code so an (ersetzen sie `...` durch den richtigen Befehl), dass der Aufruf `sumFn(5)` das richtige Ergebnis ($1+2+3+4+5=15$) liefert:

```
def sumFn(n):
    i = 0
    s = 0
    while i <= n:
        ...
        i = i +1
    return s
sumFn(5)
```

Erweitern sie die Funktion nun so, dass sie alle Zwischenresultate ausgibt:



Sie sollte nun wissen,

- wie sie in Python Funktionen definieren und diese aufrufen können,
- wie die while-Schleife funktioniert und
- wie sie in Python Text und Variableninhalte gemeinsam ausgeben können.

Erweitern sie die Funktion `sumFn(n)` nun so, dass sie auch für negative Werte etwas „sinnvolles“ zurückgibt (also z.B. für `-4` den Wert `-10`). Hinweis: sie können Funktionen auch aus Funkionen aufrufen!

Anonyme(Lambda) Funktionen

Bisher haben wir unseren Funkionen immer Namen gegeben. In Python kann man aber auch ”Namenlose” (== Anonyme) Funktionen mit Hilfe des reservierten Worts `lambda` definieren. Dies ist besonders dann hilfreich, wenn man einer Funktion eine anderen Funktion als Parameter übergeben möchte, z.B. um diese gleich für mehrere Werte auswerten zu lassen.

So können wir zum Beispiel eine Funktion schreiben, die eine anonyme Funktion f, die ihr als Parameter übergeben wird, für alle ganzzahligen Werte zwischen a und b (die wir ebenfalls als Parameter übergeben) auswertet:

```
def evaluate (f, a, b):
  return [f(x) for x in range(a, b)]
```

Rufen wir nun die Funktion `evaluate` mit den Parameter `lambda x: x * x`, 2 und 5 auf. Sie sollten als Ausgabe `[4, 9, 16]` bekommen.

Spielen sie ein wenig mit der Funktion `evaluate` herum indem sie unterschiedliche Funktionen (z.B. $\frac{1}{x}$ oder auch e^x) und Wertebereiche übergeben.
Hinweis: Implementieren sie die Exponential-Funktion (noch :-)) nicht selbst sondern verwenden sie die Mathematik-Bibliothek (`import math`) von Python.

Schreiben sie die Funktion derart um, dass auch der Parameter b mit in die Auswertung aufgenommen wird.

Sie können in Python auch Funktion als Paremeter übergeben. So können wir obiges Ergebnis auch mit

```
def sqrt(x):
    return x * x
evaluate (sqrt, 2, 5)
```

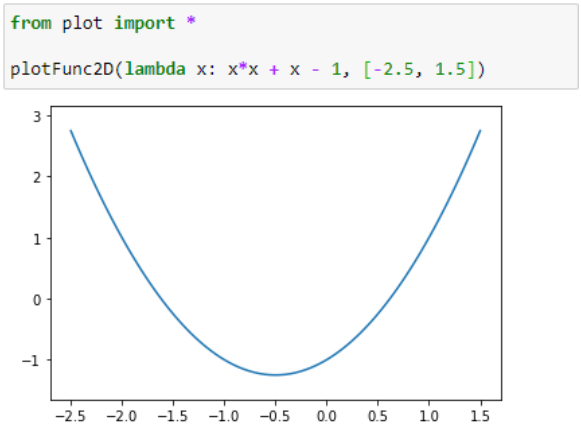
berechnen.

Funktionen Zeichnen

Oft hilft es enorm eine mathematische Funktion zu „Zeichnen“ um sie besser zu „verstehen“. Herr Weitz hat zum Plotten von mathematischen Funktionen eine eigene Biblithek (`plot`) geschrieben, welche einen *wrapper* für die etwas schwieriger anzuwendende Bibliothek `mathplotlib` bildet. Wenn sie `Jupyter` im Docker von Herr Weitz laufen lassen, ist diese Bibliothek bereits „installiert“. Falls nicht, finden sie seine Bibliothek auf seiner Homepage. Ist alles richtig installiert, sollten sie mit

```
from plot import *
plotFunc2D(lambda x: x*x + x - 1, [-2.5, 1.5])
```

einen Plot der Funktion $f(x) = x^2 + x - 1$ in karthesischen Koordinaten erhalten:



Solche Plots werden Funktionsgraphen oder nur Graphen genannt. Versuchen sie mit den Parametern `grid=True` und `scaled=True` den Plot zu beeinflussen. Plotten sie auch andere ihnen bekannte Funktionen um sich an den Umgang mit `plotFunc2D` zu gewöhnen.

Nun gibt es aber Kurven in der Ebene, die sich *nicht* als Graph einer Funktionen darstellen lassen, da sie einem Punkt der x-Achse merhrere Punkte in der x-y-Ebene zuweisen. Dies gilt z.B. für den Einheitskreis $x^2 + y^2 = 1$. Falls sie versuchen diese Funktion umzuformen und mit

`plotFunc2D(lambda x: math.sqrt(x*x - 1), [-2.5, 1.5])`) zu plotten, können sie zwar einiges über die Internas der Plot-Bibliothek erfahren, werden aber keinen Graph des Einheitskreises erhalten. Hier schafft die Parameterdarstellung Abhilfe, in der man die Punkte des Einheitskreises als Vektor abhängig von einem Parameter t (dem Winkel) darstellt

$$\vec{r}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} \cos(t) \\ \sin(t) \end{pmatrix}.$$

Machen sie sich keine Sorgen wenn sie das (noch) nicht ganz verstehen! Wir werden die Parameterdarstellung noch ausführlich behandeln. In dieser Darstellung können wir den Einheitskreis nun unter Verwendung der Funktion `plotCurve2D` Plotten, indem wir für jeden Parameter t zwei Werte (einen für $x(t)$ und einen für $y(t)$) angeben:

```
plotCurve2D(lambda t: (math.cos(t), math.sin(t)), [0, 2*math.pi])
```

Versuchen sie nun, einen Kreis mit Mittelpunkt $(1, 1)$ und Radius π zu zeichnen.

Die Bibliothek `plot` kann noch viel mehr, unter anderem 3D-Plots, Höhenlinien oder auch Konturdiagramme zeichnen. So wir diese Plots brauchen werden wir sie einführen. Wenn sie in ihrem Setup eine einfache mathematische Funktoin plotten können, reicht das zum jetzigen Zeitpunkt vollkommen.

Listen und Generatoren

In Python kann man Listen sehr einfach erzeugen. So weist die Eingabe

```
a = [1,2,3], b = ['a', 'b', 'c']
```

der Variable `a` eine Liste mit den drei Integer-Werten `1`, `2` und `3` zu, die Variable `b` bekommt eine Liste mit den drei Buchstaben `a`, `b` und `c` zugewiesen. Über Listen kann man nun *iterieren* (== die Elemente der Liste einzeln durchgehen):

```
for i in a:
  print(i)
1
2
3
```

oder auch unterschiedliche Operatoren anwenden:

```
a.append(5), a.revert(), reversed(a), a.insert(3, 4), count(a), ...
```

Spielen sie ein wenig mit den Operatoren (auch mit den hier nicht angeführten, aber auf der Python-Seite [2] angegebenen) herum und machen sie sich den Umgang mit Listen zu eigen. Wir werden sie in der linearen Algebra intensiv für die Darstellung von Vektoren und Matrizen verwenden.

Mit dem Schlüsselwort `in` können sie prüfen, ob ein Element in der Liste vorhanden ist. Der Aufruf

```
10 in a, 1 in a
```

sollte `(False, True)` liefern.

Neben der Angabe von einzelnen Elementen können wir Listen auch mit dem Befehl Range erzeugen. Für die Eingabe von

```
a = list(range(5))
b = list(range(2,6))
c = list(range(2,10,2))
```

bekommen wir `[0, 1, 2, 3, 4]`, `[2, 3, 4, 5]` und `[2, 4, 6, 8]`. Bitte studieren sie auch hier die Python-Dokumentation [2] falls sie sich nicht sicher sind was die einzelnen Parameter zu bedeuten haben. Python kann auch über Objekte vom Typ Range iterieren → lassen sie bitte das `list` weg, wenn sie über einen mit Range erzeugen Bereich iterieren möchten (sie werden am Ende dieses

Abschnitts verstehen warum dies besser ist).

Von der Syntax her besonders *elegent* finde ich den Zugriff auf Listenelementen in Python. Für

```
a = list(range(5))
a[1:3], a[3:5], a[-2], a[3:-1], a[0], a[-1]
```

sollten sie `([1, 2], [3, 4], 3, [3], 0, 4)` als Ausgabe bekommen. Auch hier empfielt sich das Studium der Python-Dokumentation [2] falls ihnen z.B. der negative Index nicht ganz geheuer ist.

Lassen sie uns zu guter letzt noch ein weiteres (meiner Meinung nach sehr *schönes*, aber zumindest sehr nützliches) Sprachkonstrukt von Python - **Generatoren** - anschauen. Überlegen sie sich dazu in einem ersten Schritt, was folgende Funktion

```
def first_n(n):
    num, nums = 0, []
    while num < n:
        nums.append(num)
        num += 1
    return nums
```

bei dem Aufruf `sum(first_n(1000000))` mit dem Hauptspeicher ihres Rechners „anstellt“. Wenn sie zu dem Schluss „nichts gutes!“ kommen, liegen sie richtig. Der Aufruf dieser Methode „zwingt“ Python dazu, alle 1000000 Werte im Hauptspeicher abzulegen und als Resultat zurückzuliefern → sehr ineffizient! Da einer der Hauptanwendungen von Listen darin besteht, auf den einzelnen Listenelementen „der Reihe nach“ Operationen auszuführen (== durch die Liste zu iterieren), ist es im Allgemeinen (so auch in unserer speziellen Anwendung, dem Bilden der Summe über alle Werte) nicht notwendig, alle Elemente auf einmal zurückzuliefern. Wir können die Elemente zu dem Zeitpunkt *generieren* zu dem wir sie benötigen → genau dies machen in Python Generatoren für uns:

```
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1
```

Diese Funktion liefert nun keine Liste mehr zurück, sondern aufgrund des `yield`-Ausdrucks einen Generator über den wir iterieren können. Das Resultat von `sum(firstn(1000000))` sollte mit dem obigen Aufruf Übereinstimmen (499999500000), nur die Berechnung ist auf diese Art wesentlich effizienten! Neben dem geringeren Speicherverbrauch führt das Verwenden von `yield` auch zu einem verbesserten Laufzeitverhalten (== wir benötigen weniger Zeit zum Berechnen der Summe). Die benötigte Zeit zum Ausführen von Python-Code kann man sehr einfach mit der Funktion `timeit` ermitteln:

```
import timeit

n = 1000000

starttime = timeit.default_timer()
sum(first_n(n))
print('Execution time of "sum(first_n(n))":', timeit.default_timer() - starttime)
starttime = timeit.default_timer()
sum(firstn(n))
print('Execution time of "sum(firstn(n))":', timeit.default_timer() - starttime)
```

Führen sie diesen Code „ein paar mal“ aus → sie sollten feststellen können, dass die Funktion `firstn(n)` im Mittel nur halb soviel Zeit benötigt wie die Funktion `first_n(n)`.

Wenn ich ihnen nun noch sage, dass der Befehl `Range(5)` keine Liste sondern einen Generator liefert, sollte mein „Versprechen“ von etwas weiter oben eingelöst sein und sie verstehen, warum es besser ist, über `range(5)` zu iterieren wie über `list(range(5))`.

Wenn sie bis hierher folgen konnten und die Beispiele selbständig ausgeführt und auch ein wenig mit Python experimentiert haben, sollten sie in der Lage sein, folgenden Code zu verstehen:

```
def someValues(seq, L = [10 ** k for k in range(10)]):
    for n in L:
        print("{:>15}: {}".format(n, seq(n)))

someValues(lambda n: n / (n + 1))
```

Wir werden diese Funktion später in der Analysis beötigen um auf einfache Art prüfen zu können, wie sich eine mathematische Funktion (im Beispiel $\frac{n}{n+1}$) „verhält“. Zum jetzigen Zeitpunkt ist aber nur wichtig, dass sie die eingeführten Python-Konstrukte gut genug verinnerlicht haben, um die Funktionsweise von `someValues` verstehen zu können. Ist dies der Fall, können sie sich in der Vorlesung und den Seminaren voll auf die Mathematik konzentrieren und dem Erlernen der linearen Algebra sowie der Grundlagen der Analysis unter Zuhilfenahme der Programmierung steht nichts mehr im Wege!

Literatur

[1] E. Weitz, Konkrete Mathematik (nicht nur) für Informatiker. 2021, https://www.springer.com/de/book/9783662626177

[2] Python.org - `https://docs.python.org/3/tutorial/index.html`

[3] Jupyter.org - `https://jupyter.org/`