Python, eine moderne Programmiersprache

Inhaltsverzeichnis

1 Was ist Python?

Python ist eine moderne Programmiersprache, die sich bei vielen Entwicklerinnen und Entwicklern großer Beliebtheit erfreut. Sie wurde Anfang der 90er Jahre von Guido van Rossum, einem niederländischen Programmierer, entworfen und wird seitdem von einem großen Team Freiwilliger als freies Open-Source-Projekt gepflegt.

Python ist eine so genannte höhere Programmiersprache, die die Zeit der Programmiererin oder des Programmierers über die Zeit des Computers stellt. Gelegentlich ist Python-Code also etwas langsamer als zum Beispiel mühsam handoptimierter C-Code – dafür lässt es sich in Python viel schneller und angenehmer entwickeln.

Zu den Anwendungsbereichen von Python gehören unter anderen die Web-, System- und Spiele-Entwicklung. Außerdem wird Python für wissenschaftliche Zwecke eingesetzt – das ist der Aspekt, den wir beleuchten werden.

2 Installation von Python

Damit ein Computer Python-Programme ausführen kann, muss man zunächst einen *Python-Interpreter* installieren. Das ist die erste Aufgabe an dich!

Unter Ubuntu Linux und anderen Debian-Derivaten. Öffne eine Konsole und setz den Befehl `sudo apt-get install python-numpy python-scipy python-matplotlib` ab. Das war's schon.

Unter Mac OS X und Windows. Lade auf `http://continuum.io/downloads` das Komplettpaket Anaconda herunter und klicke dich durch die Installation. Wähle die Python-Version 2.7 und unter Windows die 32-Bit-Version (auch, wenn dein Computer ein 64-Bit-Prozessor haben sollte).

Übers Internet. Wenn du Schwierigkeiten bei der Installation hast, schreib uns bitte an. In der Zwischenzeit kannst du aber auch übers Internet unseren Python-Server verwenden; dann musst du auf deinem eigenen Computer nichts installieren. Gehe dazu in einem Browser deiner Wahl auf `http://speicherleck.de:8888/`, klicke auf „New“ und dann auf „Python 2“. Das Passwort haben wir dir per Mail mitgeteilt. Bitte beachte: Das ist unser privater Server, und wir haben ihn nicht besonders gesichert. Du könntest also prinzipiell in unseren Dateien schnüffeln oder sie löschen. Bitte mach das nicht! `:-)`

3 Erste Schritte

Mit diesen Notizen möchten wir dir die Grundlagen der Python-Programmierung beibringen. Wenn du bisher noch keine Programmiersprache beherrschst, wirst du feststellen, dass das Erlernen gar nicht so einfach ist und etwas Zeit benötigt (bitte schreibe uns bei allen Fragen oder Problemen an!). Wir versprechen aber, dass es sich lohnt – nicht nur für unseren Kurs auf der Schülerakademie, sondern auch für das weitere Leben.[1]

Im Folgenden werden wir dir viele Code-Beispiele zeigen. Um Python zu erlernen, genügt es nicht, sich diese anzuschauen. Stattdessen musst du sie ausführen und mit ihnen *experimentieren*: Was passiert, wenn ich folgende zwei Zeilen vertausche? Was passiert, wenn ich die Einrückung entferne? Kann ich die Idee nicht auch mit anderem Code ausdrücken?

Wir stehen dir dabei gerne mit Rat und Tat zur Seite. Melde dich, wenn du nicht weiterkommst.

[1] Eine Freundin von uns war eine Zeit lang in einem Internet-Forum unterwegs, das jede Stunde, in der man aktiv auf dem Forum war, mit Punkten belohnte; diese Punkte konnte man dann in süße digitale Monster umtauschen. Offensichtlich hatte diese Freundin aber nicht Lust, Tag und Nacht auf dem Forum zu verbringen. Fünf Zeilen Python-Code später war das Problem gelöst: Sie schrieb ein Programm, das sich automatisiert jede Stunde einloggte, auf ein paar zufällige Diskussionsfäden klickte und sich dann wieder ausloggte.

3.1 Hallo, Welt!

Das erste Programm, dass man schreibt, wenn man eine neue Programmiersprache lernt, ist *Hallo Welt*: Ein Programm, dass eine kurze Meldung ausgibt und sich dann beendet. In Python sieht das so aus:

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 print("Hallo Welt!")
```

Tippe das Programm ab, speichere es unter einem Namen wie `hallo-welt.py` und führe es aus. Wenn du Python noch nicht auf deinem Rechner installiert hast und unseren Server verwenden möchtest, findest du unter `http://speicherleck.de/iblech/stuff/hallo-welt.webm` ein kurzes Video, das die nötigen Schritte illustriert. Wenn es nicht klappt, dann melde dich bei uns oder im Forum.

Die Farben dienen nur der Übersichtlichkeit; es ist üblich, verschiedene Arten von Codepassagen in jeweils anderen Farben zu setzen. Die Färbung muss nicht und kann nicht abgetippt werden; stattdessen wird jeder Quelltext-Editor selbstständig den Code einfärben.

Die ersten beiden Zeilen finden sich in jedem Python-Programm. Die erste ist vor allem unter Linux und OS X wichtig; sie ist der Indikator für das Betriebssystem, dass es sich um ein Python-Programm handelt. Es ist guter Stil, sie auch unter Windows beizubehalten. Zeile 2 hat technische Gründe.[2]

Zeile 3 ist eine Leerzeile. Leerzeilen haben für Python selbst keine Bedeutung, können also nach Belieben hinzugefügt oder entfernt werden. Es ist aber guter Stil, einzelne Sinneinheiten durch Leerzeilen zu trennen, um den Code für den Menschen übersichtlicher zu gestalten. Es gibt ja auch einen Grund, wieso es im Deutschen und vielen anderen natürlichen Sprachen Absätze gibt.

Die eigentliche Arbeit wird durch Zeile 4 angestoßen. Dort wird die in Python vordefinierte Prozedur `print` mit dem *Argument* `"Hallo, Welt!"` aufgerufen. Die Schreibweise soll an die in der Mathematik übliche Notation für Funktionen erinnern – dort schreibt man zum Beispiel „$\sin(5)$“. Im Programmierumfeld meint *print* nur *ausgeben*, nicht *drucken*. Der Begriff stammt aus einer Zeit, als es Bildschirme noch nicht gab und Computerausgaben tatsächlich ausgedruckt werden mussten.

3.2 Die interaktive Python-Shell

Python-Programme speichert man, wie im vorherigen Abschnitt beschrieben, in Dateien. Zum schnellen Experimentieren eignet sich aber auch die *interaktive Python-Shell*. In ihr kann man einzelne Python-Kommandos absetzen und erhält sofort Rückmeldung.

Auf diese Weise kann man Python zum Beispiel als Taschenrechner verwenden:

[2] Zeile 2 setzt fest, dass der Programmcode in der Zeichenkodierung UTF-8 (und nicht etwa in dem älteren Standard ISO-8859-1) interpretiert werden soll. Eine Zeichenkodierung gibt an, wie Umlaute und andere Zeichen, die über den Sprachschatz des amerikanischen ASCII-Standards aus den 60er Jahren hinausgehen, als Bytes gespeichert werden sollen.

```
$ python
Python 2.7.3 (default, Dec 18 2014, 19:03:52)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
55
>>>
```

3.3 Programmierfehler und wie man mit ihnen umgeht

Syntaxfehler Beim Programmieren macht man Fehler. Von denen gibt es vor allem zwei Arten: *Syntaxfehler* und *inhaltliche Fehler*. Ein Syntaxfehler tritt auf, wenn man sich nicht an die Schreibregeln von Python hält. Zum Beispiel wird der Code

```
1 print("Hallo, Welt!"
```

nicht funktionieren, da die schließende Klammer fehlt. Syntaxfehler werden vom Python-Interpreter direkt nach dem Start, noch bevor er mit der Ausführung des Codes beginnt, erkannt und gemeldet:

```
  File "test.py", line 2

                      ^
SyntaxError: invalid syntax
```

Achtung: Gelegentlich befindet sich ein Syntaxfehler an einer früheren Stelle als der Interpreter angibt – hier etwa befindet sich der Fehler in Zeile 1, Python berichtet jedoch einen Fehler in der (eigentlich gar nicht vorhandenen) Zeile 2. Das liegt daran, dass die Auswirkungen eines Fehlers manchmal erst später eine nicht auflösbare Inkonsistenz verursachen.

Programmiersprachen wie Python sind in ihrer Syntax sehr streng. Schon scheinbare Kleinigkeiten wie Klammerfehler oder Vertauschung ähnlich aussehender Sonderzeichen (zum Beispiel " und ') führen dazu, dass der Interpreter den Code nicht mehr versteht. Anfangs macht man viele solche Syntaxfehler, was frustrierend sein kann. Das wird aber schnell besser!

Eine Anmerkung zu Umlauten: Wenn der Interpreter sich über diese beschwert, liegt das meistens an technischen Kodierungsproblemen. Am einfachsten ist es, in solchen Fällen dem Problem aus dem Weg zu gehen und Umlaute zu umschreiben.

Inhaltliche Fehler Neben Syntaxfehlern kann man inhaltliche Fehler begehen. Diese gehören leider zur schlimmeren Sorte, da der Interpreter sich über diese nicht beschwert – die Schwierigkeit liegt bei diesen Fehlern darin, dass der Code zwar genau das tut, was er sagt; dass das aber nicht das ist, was man meinte. Ein einfaches Beispiel könnte folgender Code sein, der die Inhalte der Variablen `a` und `b` vertauschen soll (mehr zu Variablen im nächsten Abschnitt):

```
1 a = b
2 b = a
```

Aber was macht dieser Ausschnitt wirklich? Zu Beginn haben die Variablen `a` und `b` irgendwelche Werte, zum Beispiel 3 und 7. Nach Ausführung der ersten Zeile haben beide Variablen denselben Wert, nämlich 7. Das ändert sich auch nach Ausführung der letzten Zeile nicht mehr.

Eine neue Idee muss her! Der alte Wert der Variablen `a` muss in eine Hilfsvariable gesichert werden, bevor `a` mit dem Inhalt von `b` überschrieben wird:

```
x = a
a = b
b = x
```

Dieser Code funktioniert.[3]

Grundtechniken im Debugging Unter *Debugging* versteht man den Prozess, Syntaxfehler und inhaltliche Fehler im Programmcode zu beheben. Syntaxfehler werden vom Interpreter gemeldet; man behebt sie, indem man sich die betreffende Zeile genau anschaut und die Unstimmigkeit sucht. Manchmal möchte man einen Fehler partout nicht erkennen, dann hilft es, eine Pause zu machen oder den Code jemand anderem zu zeigen.

Eine grundlegende Strategie, um inhaltliche Fehler zu beheben, besteht darin, den Inhalt von Variablen durch **`print`**-Anweisungen zu verfolgen, zum Beispiel so:

```
print("VORHER", a, b)
a = b
print("DANACH", a, b)
b = a
print("AM ENDE", a, b)
```

So kann man Widersprüche zwischen dem erwarteten und tatsächlichen Verhalten aufdecken.

4 Variablen und Kontrollstrukturen

Programme werden erst dann interessant, wenn sie durch veränderliche Variablen und *Kontrollstrukturen* nicht einfach linear von oben nach unten ablaufen. Variablen sind Platzhalter für Daten, die sich während des Programmablaufs mehrmals ändern können. Zum Beispiel wird das Programm

```
foo = 5
print(foo)
foo = 100
print(foo)
foo = foo + 3
print(foo)
```

die Zahlen `5`, `100` und `103` ausgeben. Dabei ist „`foo`“ anders als **`print`** kein von Python vordefinierter Begriff; wir hätten die Variable auch anders nennen können.

[3] Wenn man sich in Python besser auskennt, weiß man, dass es sogar noch eine idiomatischere Lösung gibt: Zur Vertauschung kann man einfach die Anweisung `a, b = b, a` verwenden.

4.1 For-Schleifen

Was ist die Summe der ersten 10 positiven natürlichen Zahlen? Ein Python-Programm, das die Antwort auf diese Frage berechnet, könnte wie folgt aussehen:

```
summe = 0
summe = summe + 1
summe = summe + 2
summe = summe + 3
summe = summe + 4
summe = summe + 5
summe = summe + 6
summe = summe + 7
summe = summe + 8
summe = summe + 9
summe = summe + 10
```

Es ist offensichtlich, dass es keinen Spaß macht, diese Art Code zu schreiben! Eine einfache Idee, immer die nächste Zahl auf den aktuellen Zwischenstand zu addieren, wird hier in zehn Zeilen ausgebreitet. **Echte ProgrammierInnen sind nicht bereitet, derart repetitiven Code zu schreiben!** Und du solltest es auch nicht sein. (Was wäre, wenn die Aufgabe gefordert hätte, die ersten 1000 Zahlen zu summieren?) Viel besser geht es mit einer *For-Schleife*:

```
summe = 0
for i in range(11):
    summe = summe + i
```

Was passiert hier? `range(11)` ist die Liste der Zahlen von `0` einschließlich bis `11` ausschließlich. Das kann man in einer interaktiven Python-Shell auch überprüfen:

```
>>> range(11)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Durch das Schlüsselwort `for` passiert nun folgendes: Der eingerückte Block wird mehrmals durchlaufen. Dabei hat die neue Variable `i` im ersten Durchgang den Wert `0`, im zweiten den Wert `1`, und so weiter, bis schließlich im letzten Durchlauf `i` den Wert `10` hat.

Aufgabe 1. *Wo steckt der Fehler?*

Erkläre, wieso folgender Code – der doch ganz ähnlich aussieht – nicht funktioniert.

```
summe = 0
for i in range(11):
summe = summe + i
```

Aufgabe 2. *Summe der Quadratzahlen*

Schreibe ein Python-Programm, das die Summe der ersten 1000 Quadratzahlen bestimmt.

Aufgabe 3. *Geschachtelte Schleifen*

Was macht folgendes Programm? Füge gegebenenfalls `print`-Anweisungen ein, um die Veränderungen der Variablen `i` und `j` nachzuverfolgen! Kannst du das Programm vereinfachen?

```
summe = 0
for i in range(11):
    for j in range(i):
        summe = summe + 1
```

Aufgabe 4. *Die harmonische Reihe*

Schreibe ein Python-Programm, dass folgende unendliche Summe berechnet:

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots .$$

Gibt es ein Problem?

4.2 If-Abfragen

Was ist die Summe der ersten 100 geraden Zahlen? Eine Möglichkeit, die Antwort auf diese Frage zu finden, besteht darin, zunächst *alle* ersten 100 Zahlen durchzugehen (mit `for` und `range(101)`), dann aber nur im Fall einer geraden Zahl den Zähler `summe` zu erhöhen. Das geht so:

```
summe = 0
for i in range(101):
    if i % 2 == 0:
        summe = summe + i
```

Oder etwas ausführlicher:

```
summe = 0
for i in range(101):
    if i % 2 == 0:
        print("Die Zahl", i, "ist gerade! Addiere sie.")
        summe = summe + i
    else:
        print("Die Zahl", i, "ist ungerade! Überspringe sie.")
```

Das Prozentzeichen führt eine *Modulo-Rechnung* durch; das Ergebnis von `a % b` ist der Rest, der bei der Division von `a` durch `b` übrig bleibt. In der Python-Shell kann man das anhand einiger Beispiele nachvollziehen:

```
>>> 10 % 4
2
>>> 11 % 4
3
>>> 12 % 4
0
>>> 13 % 4
1
```

Mit dem `==`-Operator wird ein Vergleich durchgeführt: Stimmen die Zahlen links und rechts des Operators überein?

```
>>> 17 == 0
False
>>> 0 == 0
True
>>> 12 % 4 == 0
True
>>> 13 % 4 == 0
False
```

Achtung: Verwechsle nicht `a == b` mit `a = b`. Der erste Operator vergleicht die Werte `a` und `b` miteinander (und gibt `True` oder `False` zurück, je nachdem, ob sie gleich oder ungleich sind). Der zweite Operator setzt die Variable `a` auf den Wert von `b`.[4]

Mit dem **`if`**-Schlüsselwort kann man erreichen, dass je nachdem, ob der Fall `True` oder der Fall `False` vorliegt, anderer Code ausgeführt wird. Im Beispielcode von Beginn dieses Abschnitts erhöhen wir die Variable `summe` also nur dann um den Wert von `i`, wenn `i` bei Division durch 2 keinen Rest lässt, wenn also `i` gerade ist.

Aufgabe 5. *Perfekte Zahlen*

Eine *perfekte Zahl* ist eine positive Zahl mit der besonderen Eigenschaft, dass die Summe ihrer echten positiven Teiler sie selbst ist. Zum Beispiel ist die Zahl 6 eine perfekte Zahl, denn $6 = 1 + 2 + 3$. Schreibe ein Python-Programm, das von einer gegebenen Zahl überprüft, ob sie perfekt ist!

[4] Wenn du die beiden Konstrukte doch verwechselst, bist du in guter Gesellschaft. Es gab schon manche gravierende Sicherheitslücke in Computersystemen, welche durch eine solche Unachtsamkeit verursacht wurde.

4.3 While-Schleifen

For-Schleifen sind super, wenn man von vornherein weiß, wie viele Schleifendurchläufe man sich wünscht. Für den Fall, dass man das nicht weiß, gibt es *While-Schleifen*. Diese werden so lange durchlaufen, wie eine beliebige Bedingung, die man selbst angeben kann, erfüllt ist.

Wie viele Zahlen muss man addieren, damit die Summe die Zahl 1000 übersteigt? Folgendes Programm berechnet die Antwort:

```
1 summe = 0
2 i     = 0
3
4 while summe <= 1000:
5     i     = i + 1
6     summe = summe + i
7
8 print("Die Summe der ersten", i, "Zahlen ist größer als 1000.")
```

Manche Leute finden es schön, im Fall von mehreren Zuweisungen die Gleichheitszeichen untereinander auszurichten. Anderen ist das egal; Python selbst interessiert sich jedenfalls nicht dafür.

Es gibt noch eine zweite Art und Weise, diesen Code zu formulieren:

```
1 summe = 0
2 i     = 0
3
4 while True:
5     if summe > 1000:
6         break
7     i     = i + 1
8     summe = summe + i
9
10 print("Die Summe der ersten", i, "Zahlen ist größer als 1000.")
```

Als While-Bedingung steht hier `True`; a priori wird der Schleifenrumpf (die Zeilen 5–8) also endlos ausgeführt. Falls jedoch der Wert der Variable `summe` die Grenze `1000` übersteigt, erreicht der Programmfluss Zeile 6. Das Schlüsselwort **`break`** führt dann dazu, dass in diesem Fall die Schleife vorzeitig verlassen wird; die Code-Ausführung wird also auf Zeile 9 fortgesetzt.

Diese Art der Formulierung ist manchmal praktikabler – zum Beispiel, wenn die Abbruchbedingung selbst das Resultat längerer, sich über mehrere Zeilen erstreckender Rechnungen ist.

Aufgabe 6. *Die Collatz-Vermutung*

Denke dir eine positive natürliche Zahl. Ist sie gerade, so halbiere sie. Ist sie ungerade, so verdreifache sie und addiere anschließend Eins. Fahre in beiden Fällen auf dieselbe Art und

Weise mit der entstehenden Zahl fort. Beginnt man beispielsweise mit der Zahl 17, so erhält man die Folge

$$17,\quad 52,\quad 26,\quad 13,\quad 40,\quad 20,\quad 10,\quad 5,\quad 16,\quad 8,\quad 4,\quad 2,\quad 1,\quad 4,\quad 2,\quad 1,\quad \ldots$$

Die *Collatz-Vermutung* besagt, dass man bei *jeder* Ausgangszahl schlussendlich die Zahl 1 (und somit den Zykel 4–2–1) erreicht. Sie ist noch völlig offen; der berühmte Mathematiker Terence Tao ist der Meinung, dass Mathematik noch nicht reif für Probleme dieser Art sei.

Schreibe ein Python-Programm, das die Vermutung zumindest für die ersten 10.000 Zahlen überprüft.

5 Visualisierung von Daten

5.1 Plots von Punkten

Im vorherigen Abschnitt hast du die grundlegenden Möglichkeiten der Programmflussgestaltung kennengelernt. Jetzt soll es darum gehen, wie man Daten – zum Beispiel Punkte in der Ebene – grafisch darstellen kann. Folgendes Programm etwa erzeugt Abbildung 1.[5]

```
1 import numpy as np
2 import matplotlib.pyplot as pl
3
4 xs = [-3, 5, 2, 1]
5 ys = [ 9, 1, 0, 4]
6
7 pl.plot(xs, ys)
8 pl.show()
```

Was passiert hier? In den ersten beiden Zeilen werden die fürs Plotten benötigten Pakete geladen. Die Zeilen 4 und 5 definieren gemeinsam die darzustellenden Punkte; die Variable `xs` enthält die x-Werte, die Variable `ys` enthält die y-Werte der Punkte. Der Plot beginnt also mit dem Punkt $(-3|9)$. Beide Variablen hätte man auch anders nennen können. Schließlich weisen wir in Zeile 7 die Grafikbibliothek an, einen Plot der Punkte vorzubereiten, welcher durch den Befehl in Zeile 8 angezeigt wird.

Auf diese Weise kann man also *Punkte* plotten – irgendwelche x-Werte gegen irgendwelche y-Werte.

5.2 Plots von Funktionen I

Wenn man *Funktionsgraphen* erstellen möchte, muss man selbst Hand anlegen:

[5] Wenn du das Python von unserem Server verwendest, musst du als erste Zeile im Programm `%matplotlib inline` angeben, so wie im Beispiel unter `http://speicherleck.de:8888/notebooks/einfacher-plot.ipynb`.

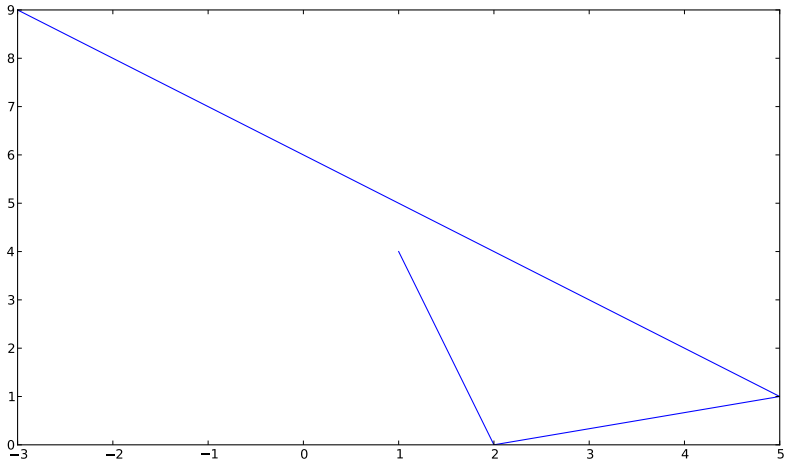

Abbildung 1: Unser erster Plot mit Python.

```
1 import numpy as np
2 import matplotlib.pyplot as pl
3
4 xs = np.linspace(0, 7, 300)
5 ys = np.sin(xs)
6
7 pl.plot(xs, ys)
8 pl.show()
```

Dieser Code erzeugt Abbildung 2. Während wir im vorherigen Beispiel manuell eine Liste von x-Koordinaten eingaben, übernimmt das jetzt die Methode `np.linspace`. Sie erzeugt gleichmäßig verteilte Stellen zwischen zwei Werten. Das kann man gut in der interaktiven Python-Shell erkennen:

```
>>> np.linspace(3, 7, 10)
array([ 3.        ,  3.44444444,  3.88888889,  4.33333333,  4.77777778,
        5.22222222,  5.66666667,  6.11111111,  6.55555556,  7.        ])
```

In Zeile 5 rufen wir die Sinusfunktion auf – und zwar für jeden x-Wert in der Variablen `xs` einmal. Die NumPy-Bibliothek, die wir verwenden, nimmt uns dabei das Schreiben der eigentlich dafür benötigten For-Schleife ab und erlaubt uns die bequeme Notation.

```
>>> np.sin(np.linspace(3,7,10))
array([ 0.14112001, -0.29824342, -0.67965796, -0.9290145 , -0.99786291,
       -0.8728259 , -0.57819824, -0.17122628,  0.26901506,  0.6569866 ])
```

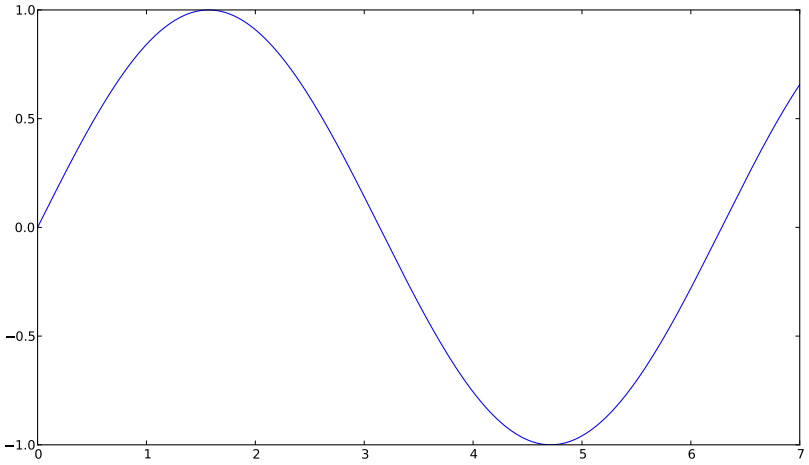

Abbildung 2: Die Sinus-Funktion.

5.3 Plots von Funktionen II

Im vorherigen Beispiel ging es um die Sinus-Funktion, für die NumPy schon Unterstützung mitbringt. Aber was, wenn man eine Funktion plotten möchte, die NumPy nicht kennt?

```
1 import numpy as np
2 import matplotlib.pyplot as pl
3
4 xs = np.linspace(0, 7, 300)
5 ys = [ x*x-4*x+3 for x in xs ]
6
7 pl.plot(xs, ys)
8 pl.show()
```

Der Ausdruck in eckigen Klammern auf Zeile 5 ist eine so genannte *list comprehension*. Man liest ihn wie folgt: Die Variable `ys` ist die Liste all derjenigen Ausdrücke `x^2-4*x+3`, wobei `x` über alle Werte der Liste `xs` läuft. Anders formuliert: Python geht nach und nach alle Werte der Liste `xs` durch, also alle x-Koordinaten. Der jeweils aktuelle Wert landet dabei in der Hilfsvariable `x`. Zur neu entstehenden Liste `ys` wird dann der Wert `x*x-4*x+3`, in mathematischer Notation also $x^2 - 4x + 3$, hinzugefügt.

Das Ergebnis zeigt Abbildung 3.

Aufgabe 7. *Plots von Ableitungen*

Wir möchten die Ableitung der Funktion $f : \mathbb{R} \to \mathbb{R}$ mit $f(x) = x^2$ untersuchen. Im nächsten Mathe-Blatt lernst du, wie man die Ableitung per Hand und exakt bestimmen kann; für jetzt soll

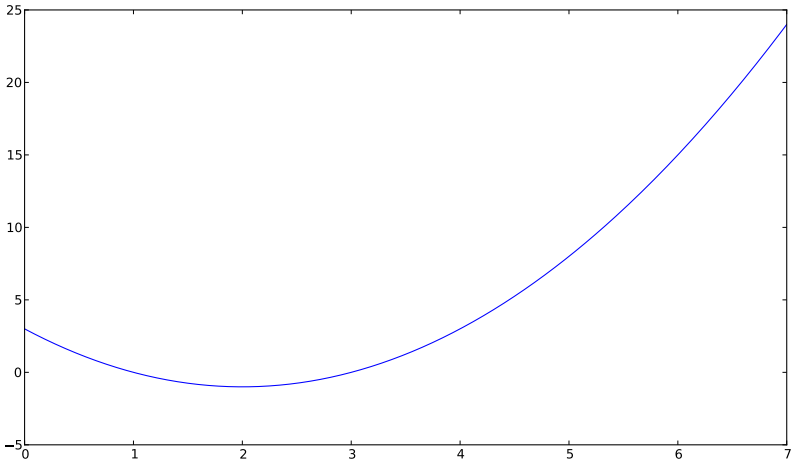

Abbildung 3: Ein quadratisches Polynom.

uns jedoch eine *numerische Approximation* genügen. Die Formel für den Differenzenquotient lautet

$$\frac{f(x+h) - f(x)}{h}.$$

Dabei ist h frei wählbar; je kleiner h ist, desto näher liegt dieser Quotient an der tatsächlichen Ableitung $f'(x)$ von f an der Stelle x. (Der Idealfall wäre $h = 0$, aber Null können wir in die Formel nicht einsetzen!)

Schreibe ein Programm, das speziell für $h = 0.1$ den Verlauf des Differenzenquotienten plottet! Orientiere dich dabei an dem Beispiel aus diesem Abschnitt.

Aufgabe 8. *Numerische Schwierigkeiten*

Diese Aufgabe setzt die vorherige fort. Welches seltsame Verhalten zeigt sich, wenn du $h = 10^{-14}$ verwendest? In Python kannst du für diese Konstante einfach `1e-14` schreiben.

Hinweis: Wenn du kein seltsames Verhalten feststellst, experimentiere mit anderen Werten von h – noch kleineren oder größeren. Es kann sein, dass bei deiner Python-Installation das Problem bei einer anderen Grenze auftritt.

6 Das Newton-Verfahren

Sei $f : \mathbb{R} \to \mathbb{R}$ eine Funktion, zum Beispiel $f(x) = x^2 - 7x + 5$. Ein wichtige Frage mit vielen Anwendungsfällen lautet: *Wie können wir Nullstellen von f finden?* In manchen Situationen ist das leicht. Etwa ist im Beispiel f eine quadratische Funktion, daher kann man mit der

Lösungsformel für quadratische Gleichungen („Mitternachtsformel“) die Nullstellen sofort hinschreiben:

$$x_{1,2} = \frac{7 \pm \sqrt{(-7)^2 - 4 \cdot 1 \cdot 5}}{2 \cdot 1}.$$

Aber was, wenn f kubisch oder quartisch ist? Was, wenn f einen noch höheren Grad hat? Was, wenn f gar kein Polynom ist, sondern zum Beispiel $f(x) = \sin(x^2 - e^x) + \tan(\sqrt{x^2+1})$ als Funktionsterm hat? Dann gibt es keine allgemeine Formel, mit der man die Nullstellen über eine einfache Rechnung bestimmen könnte.[6] Tatsächlich gibt es in vielen Fällen sogar *überhaupt keine Formel* für die Nullstellen – das kann man sogar beweisen.

Wir müssen uns daher im Allgemeinen mit dem nächstbesten begnügen: numerische Approximationen für die Nullstellen. Das Newton-Verfahren ist eines der wichtigsten Verfahren zur numerischen Nullstellenberechnung.

Aufgabe 9. *Erste Schritte mit dem Newton-Verfahren*

Hol dir einen Taschenrechner; ein ganz einfacher, nicht programmierbarer, genügt. Er muss nur eine Taste Ans haben, die das zuletzt berechnete Ergebnis einfügt. Nimm deine Lieblingsfunktion $f : \mathbb{R} \to \mathbb{R}$. Gib deine Lieblingszahl in den Taschenrechner ein. Tippe dann die Formel

$$\texttt{Ans} - f(\texttt{Ans})/f'(\texttt{Ans})$$

ein und betätige wiederholt die „Jetzt ausrechnen“-Taste, wobei du den Term für die Ableitung vorher per Hand ausrechnen musst (siehe Tabelle 1). Wenn du zum Beispiel $f(x) = x^2 - 7x + 5$ gewählt hast, musst du die Formel

$$\texttt{Ans} - (\texttt{Ans}^2 - 7 \cdot \texttt{Ans} + 5)/(2 \cdot \texttt{Ans} - 7)$$

verwenden (Klammern nicht vergessen). Probiere es aus! Mit ein bisschen Glück werden sich – unabhängig von der zu Beginn gewählten Lieblingszahl – die Zwischenergebnisse nach und nach immer weniger ändern und immer bessere Näherungen für eine Nullstelle von f liefern.

Aufgabe 10. *Das Newton-Verfahren in Python*

Schreibe ein Programm, dass das, was du in der vorherigen Aufgabe mit dem Taschenrechner gemacht hast, automatisiert. Zu deiner Lieblingsfunktion und deiner Lieblingszahl soll das Programm also die Formel $\texttt{Ans} - f(\texttt{Ans})/f'(\texttt{Ans})$ des Newton-Verfahrens immer wieder, sagen wir 30 Mal, verwenden und dabei alle Zwischenschritte ausgeben.

Hier ein grober Plan, um das zu erreichen:

[6] Für kubische und quartische Gleichungen gibt es noch Lösungsformeln wie die Mitternachtsformel. Man hat aber beweisen können, dass es für Polynomgleichungen höheren Grades *keine Lösungsformeln geben kann*. Es ist also nicht nur so, dass wir Menschen zum jetzigen Zeitpunkt keine solche Formel kennen; vielmehr kann es rein prinzipiell eine solche Formel nicht geben. Dieses verblüffende Resultat lernt man im Mathe-Studium in jeder Vorlesung über *Galoistheorie*.

Funktion $f(x)$	Ableitung $f'(x)$
12	0
x	1
x^2	$2x$
x^3	$3x^2$
x^4	$4x^3$
$\sin(x)$	$\cos(x)$
$\cos(x)$	$-\sin(x)$
$\ln(x)$	$\frac{1}{x}$
e^x	e^x
$\sqrt{x}$	$\frac{1}{2\sqrt{x}}$
$\sin(x) + x^3$	$\cos(x) + 3x^2$
$\cos(x) + \ln(x)$	$-\sin(x) + \frac{1}{x}$
$17 \cdot \sin(x)$	$17 \cdot \cos(x)$
$x^3 - 5x^2 + 2x - 8$	$3x^2 - 10x + 2$
$g(x) + h(x)$	$g'(x) + h'(x)$
$g(x) \cdot h(x)$	$g'(x) \cdot h(x) + g(x) \cdot h'(x)$

Tafel 1: Praktische Regeln zum Ableiten.

- Du brauchst eine Variable `x`. Zu Beginn weist du dieser Variablen deine Lieblingszahl als Wert zu. Im Lauf des Programms ändert sie sich dann immer wieder, gemäß der Formel.
- Du brauchst eine For-Schleife, um die 30 Wiederholungen umsetzen zu können.
- Du musst die Formel in Python-Code übertragen.
- Du musst die Zwischenergebnisse mit **`print`** ausgeben.
- Du brauchst nicht: While, If und Plots.

Das Newton-Verfahren funktioniert nicht immer, aber sehr oft. Außerdem liefert es, wenn es überhaupt funktioniert, meistens schon nach wenigen Schritten gute Näherungen für eine Nullstelle. Prinzipbedingt hat das Newton-Verfahren Schwierigkeiten bei Funktionen, die an manchen Stellen nicht differenzierbar sind. Schwierig sind außerdem Funktionen, die an manchen Bereichen sehr flach verlaufen: Denn auf solchen Bereichen ist die Ableitung fast Null, das Dividieren durch die Ableitung also problematisch.

Zur Lösung der Programmieraufgabe musst du das nicht wissen, aber vielleichst bist du trotzdem neugierig, was die Idee hinter dem Newton-Verfahren ist und woher die Formel $\texttt{Ans} - f(\texttt{Ans})/f'(\texttt{Ans})$ kommt. Das erklärt der Wikipedia-Artikel zum Newton-Verfahren recht gut. Wenn du gerade keine Lust auf Lesen hast, kannst du aber auch einfach ein YouTube-Video dazu anschauen: `https://www.youtube.com/watch?v=xGemDmrCqEk`

7 Das Gradientenabstiegsverfahren zur Minimumssuche

Sei $f : \mathbb{R} \to \mathbb{R}$ eine Funktion. Wie können wir Minimal- und Maximalstellen von f finden? Wenn man f zeichnen kann, und wenn der Graph überschaubar ist, ist das natürlich ganz leicht. Auch ist es leicht, wenn man die Ableitung berechnen kann und wenn diese nicht zu kompliziert ist.

Aber wie geht es allgemein? Und – noch viel wichtiger – wie geht man bei Funktionen vor, die nicht von einer Variable abhängen („$f(x)$“), sondern von mehreren („$f(x, y, z, w)$“ oder „$f(x_1, x_2, \ldots, x_{3400})$“)? In diesem Fall ist ein Plot völlig unmöglich.

Dann kann man das *Gradientenabstiegsverfahren* verwenden. Das gibt es in einer Variante für die Minimums- und einer für die Maximumssuche. Für die Minimumssuche läuft es so:

Man startet mit irgendeiner Stelle x. Dann berechnet man die Ableitung $f'(x)$ an dieser Stelle. Wenn diese positiv ist, so steigt die Funktion in einer Umgebung von x und man sollte das Minimum weiter links suchen. Man verringert x also ein bisschen, und zwar mit der Formel $\tilde{x} := x - \eta f'(x)$, und macht dann mit $\tilde{x}$ als neue Stelle weiter. Wenn $f'(x)$ dagegen negativ ist, so fällt die Funktion und man sollte das Minimum weiter rechts suchen. Das macht witzigerweise dieselbe Formel $\tilde{x} := x - \eta f'(x)$.

Dabei ist η ein Parameter, mit dem man steuern kann, wie groß die Schritte sein sollen. Es gibt kein Patentrezept, wie man η wählen sollte; es gibt aber zumindest Heuristiken, mit denen man die Wahl von η während des Verfahrens dynamisch anpassen kann. So tief möchten wir an dieser Stelle aber nicht einsteigen.

Aufgabe 11. *Das eindimensionale Gradientenabstiegsverfahren in Python*

Setze das Gradientenabstiegsverfahren in Python um. Schreib also Code, der mit dem Gradientenabstiegsverfahren versucht, ein Minimum deiner Lieblingsfunktion zu finden. Probiere verschiedene Werte von η aus, zum Beispiel $\eta = 1$ und $\eta = 0.001$. Funktioniert das Verfahren gut? Läufst du in Probleme?

Der grobe Plan dazu ist:

- Du brauchst eine Variable `x`. Zu Beginn weist du dieser Variablen deine Lieblingszahl als Wert zu. Im Lauf des Programms ändert sie sich dann immer wieder, gemäß der Formel $\tilde{x} = x - \eta f'(x)$.
- Du brauchst eine For- oder While-Schleife, um viele Wiederholungen umsetzen zu können.
- Du musst die Formel in Python-Code übertragen.
- Du musst die Zwischenergebnisse mit `print` ausgeben.
- Du brauchst nicht: If und Plots.

Wieso heißt das Verfahren eigentlich *Gradientenabstiegsverfahren*? Der Name erklärt sich am besten in mehreren Dimensionen, in denen das Verfahren auch funktioniert. Dann verwendet man statt $f'(x)$ den Gradienten $\nabla f(x)$.