

Skriptum zur VO USW-Computational Basics

von Georg Jäger und Manfred Füllsack



Dieses Werk ist lizenziert unter einer
Creative Commons Namensnennung
4.0 International Lizenz.

Inhalt

Kapitel 0 – Einleitung – Warum Computation?	1
Kapitel 1 – Einführung in Python – Der Froschteich.....	9
Kapitel 2 – Populationsentwicklungen	14
Kapitel 3 – Wachstum mit Feedback – Die If-Bedingung	19
Kapitel 4 – Zufallszahlen – Ein Energiemix	29
Kapitel 5 – Vektoren und Matrizen – Die Bewirtschaftung eines Waldes	37
Kapitel 6 - Rekursionen und Zellulare Automaten	43
Kapitel 7 – Differenzieren und Integrieren – Ein Solarauto	49
Kapitel 8 – Diskret und kontinuierlich - Die Zinsrechnung	58
Kapitel 9 – Gekoppelte Differentialgleichungen – Räuber-Beute-Systeme	64
Kapitel 10 – Analytisch Differenzieren und Integrieren mit dem Paket Sympy	69
Kapitel 11 – Numerische Integrationsmethoden	73
Kapitel 12 – Visualisierungen	76
Kapitel 13 – Netzwerke	83
Kapitel 14 – Die Stabilität von Systemen.....	93
Kapitel 15 – Spaß mit Attraktoren.....	100
Glossar.....	104

Kapitel 0 – Einleitung – Warum Computation?

Die Vorlesung „USW-*Computational Basics*“ führt, zusammen mit der Vorlesung „*Systemwissenschaften-1*“, in das Fach Systemwissenschaften ein. Im vorliegenden Skriptum liegt das besondere Augenmerk auf der zentralen Methodik der Systemwissenschaften, der Verwendung von Computern zur Systemmodellierung und Systemanalyse .

Das Skriptum ist die Textfassung einer Zusammenstellung einer speziell geschriebenen Software-Sammlung, die, als Kapitel mit erklärendem Text gestaltet, in Form von so genannten Jupyter-Notebooks (- was das ist, wird weiter unten beschrieben -) auf <https://moodle.uni-graz.at> für USW-Studierende bereitgestellt wird. Die Notebooks sind in der Programmiersprache *Python* geschrieben, deren Anwendung dieses Skriptum vermitteln will. Das Skriptum ist also eine „Einführung in das

wissenschaftliche Programmieren mit Python“ mit speziellem Schwerpunkt auf Themengebieten, die in den Systemwissenschaften relevant sind.

Zur Anwendung von Python und zur Verwendung der Notebooks ist ein spezielles, frei-verfügbares (d.h. kostenloses) Software-Paket notwendig, dessen Download und Installation ebenfalls weiter unten beschrieben wird.

Für die zeitgenössische Systemwissenschaft sind Grundkenntnisse in wissenschaftlichem Programmieren unerlässlich, da es diese Disziplin vielfach mit **komplexen, dynamischen Systemen** zu tun hat, für deren Analyse rein mathematische Methoden nicht ausreichen. In vielen Fällen führt kein Weg zum Verständnis dieser Systeme an der Simulation am Computer vorbei. Und oftmals unterstützt die Computersimulation auch das Verständnis mathematischer Zusammenhänge.

Was genau sind komplexe, dynamische Systeme?

Im Folgenden wollen wir diese Begriffe kurz in umgekehrter Reihenfolge besprechen.

a. Systeme

Der Begriff System kommt aus dem Griechischen und meint etwas „Zusammengesetztes“ oder „Zusammengestelltes“, also etwas, das aus mehreren Komponenten besteht. Implizit wird im modernen Systemverständnis dabei mitgemeint, dass dieses Zusammengesetzte „irgendwie mehr“ ist als sich aus der Betrachtung seiner Komponenten zu ergeben scheint. *Das Ganze scheint mehr als die Summe seiner Teile zu sein*, lautet diesbezüglich ein bekannter, Aristoteles zugeschriebener Spruch. Das System scheint Eigenschaften zu haben, die die Komponenten, aus denen es zusammengesetzt ist, nicht haben. Man spricht diesbezüglich auch davon, dass das System ein **Eigenverhalten** oder auch eine **Eigenform** hat.

b. Dynamik

Systeme können statisch sein und auch dabei ein Eigenverhalten zeigen. Das zu Eis erstarre Wasser behält eben zwischen circa minus 270 und 0 °C seine spezifische Eigenform. Wie bei diesem Beispiel geht den meisten statischen Systemzuständen aber eine dynamische Phase voraus, in der das System seine spezifische Systemeigenschaft, sein Eigenverhalten, erst ausbildet, bzw. findet. Das Wasser gefriert erst durch Abkühlung zu Eis. Für die Analyse des Eigenverhaltens, d.h. der spezifischen Systemeigenschaft, ist deshalb die dynamische Phase des Systems, oder allgemeiner: das dynamische Systemverhalten entscheidend.

Dynamik meint, dass sich das System und/oder seine Komponenten bewegen oder verändern. Mathematisch lassen sich diese Veränderungen mithilfe von Differenzen- oder Differentialgleichungen darstellen. Für die einfacheren unter diesen Gleichungen kennt die Mathematik analytische Lösungen, das heißt, für einfache dynamische Systeme, die nur aus ein oder zwei sich verändernden Komponenten bestehen, gibt es für Darstellung (Modellierung) und Analyse mathematische Werkzeuge. Leider gilt dies aber nur für einen sehr überschaubaren Bereich von Systemen. Schon wenn in einem System drei statt nur zwei Komponenten dynamisch interagieren – etwa drei Himmelskörper, die gegenseitig ihre Umlaufbahnen mit ihrer Schwerkraft beeinflussen – versagen gängige mathematische Mittel ihren Dienst. Hier beginnt der Bereich der Computersimulation. Da viele der heutzutage analytisch untersuchten Systeme aus 1000en bis

100.000en interagierenden Komponenten bestehen, ist die Analyse entsprechender Systeme unweigerlich auf digitale Werkzeuge, auf den **Computer** also, angewiesen.

Dies gilt umso mehr für so genannte komplexe Systeme.

c. Komplexität

Der Begriff komplex kommt aus dem Lateinischen und steht für so etwas wie „verflochten“ oder „verwoben“. Er meint das Ineinandergreifen und aufeinander Wechselwirken der Komponenten eines Systems, aus dem in der Gesamtwirkung das Eigenverhalten des Systems entsteht. Man spricht bezüglich dieses Entstehens auch von **Emergenz** der spezifischen Systemeigenschaften und meint damit, dass die spezifischen Systemeigenschaften in gewisser Weise *unerwartet* und *plötzlich* auftauchen. Das Ineinandergreifen und Wechselwirken, das in der Regel auch mit *nicht-linearen* Entwicklungen, sprich mit Rückkoppelungswirkungen, verbunden ist, erzeugt damit ein analytisches Problem: trotz genauer Kenntnis der Einzelkomponenten und ihrer Dynamik lässt sich nicht ohne weiteres auf das Systemverhalten, also auf die Konsequenzen des Zusammenwirkens schließen. Das Ganze scheint eben *mehr* als die Summe seiner Teile. Das Ganze (d.h. das System) kann sich *kontraintuitiv* verhalten.

Strenggenommen lässt sich Komplexität auch nicht reduzieren, da dabei die spezifische Systemeigenschaft, also unter Umständen genau das systemische Eigenverhalten, das eigentlich den Grund einer Untersuchung darstellt, verloren geht. Das System wäre dann nicht mehr dasselbe.

Diesbezüglich unterscheidet sich Komplexität auch von Kompliziertheit. Während komplizierte Zusammenhänge zwar mitunter schwer, aber letztendlich doch, im Hinblick auf Ursache-Wirkungsketten (Kausalketten) analysiert und verstanden werden können, gelingt dies im Fall von komplexen Zusammenhängen nicht. Hier ist eben trotz vollständiger Information zum Verhalten der Einzelkomponenten des Systems der Schluss auf den Effekt ihres Zusammenwirkens nicht immer möglich. Einzig die Computersimulation erlaubt hier im Versuch, das System in seinen entscheidenden Komponenten zu modellieren, so etwas wie eine „Filmaufnahme“, die dann durch Vor- und Zurückspulen der entscheidenden Sequenzen, so etwas wie analytisches Detailverständnis erbringen kann. Es wird diesbezüglich davon gesprochen, dass mithilfe der Computersimulation das mikro-kausale Netzwerk der Systembedingungen „durchkrochen“ werden kann¹.

Dies ist im Wesentlichen der Grund dafür, dass die modernen Systemwissenschaften unverzichtbar auf grundlegende Programmierkenntnisse angewiesen sind. Die Vorlesung „USW-Computational Basics“ – und damit das vorliegende Skriptum – stellen sich die Aufgabe, die entsprechenden Mindestkenntnisse zu vermitteln.

Was heißt Programmieren?

Unter Programmieren versteht man ganz allgemein die Tätigkeit, einem Computer Anweisungen zu geben. Das Eintippen einer Rechenanweisung in einen Taschenrechner lässt sich also schon als Programmieren betrachten. Man erklärt dem Computer die Berechnung, die man gerne durchführen möchte, und der Computer erledigt die Rechenarbeit. Einfache Taschenrechner kennen jedoch nur wenige Befehle, meist nur Grundrechenarten. Teurere kennen zum Beispiel Winkelfunktionen oder

¹ „Crawling the micro-causal web by way of simulation“, Bedau M.A. (1997) Weak emergence. Philosophical Perspectives 11: 375–399.

die Möglichkeit, Zahlen zu speichern und wieder abzurufen. Will man noch komplexere Berechnungen durchführen, ist man auf „echte“ Computer angewiesen.

Im Unterschied zum Taschenrechner mit seinen Tasten für jede Grundrechenart, muss dem Computer nun allerdings der Umstand, dass zum Beispiel die Wurzel einer Zahl berechnet werden soll, mithilfe einer Programmiersprache mitgeteilt werden. Dies könnte zum Beispiel der Befehl `sqrt(x)` sein, den es wohl – abhängig von der so genannten *Syntax* einer Sprache – in der einen oder anderen Form in jeder Programmiersprache gibt. Das hat den Vorteil, dass die Kenntnis einer Programmiersprache in der Regel dafür sorgt, dass auch andere Programmiersprachen leicht erlernt werden. Darüber hinaus steigern Programmierkenntnisse die allgemeine Problemlösungskompetenz, die Abstraktionsfähigkeit und die präzise Ausdrucksweise im wissenschaftlichen Kontext.

Die Programmiersprache Python

Die Vorlesung – und damit dieses Skriptum – beziehen sich auf die Programmiersprache **Python**. Dies hat mehrere Gründe: Python (<https://www.python.org>) ist eine universelle, vielseitig einsetzbare Programmiersprache, die

1. in ihrer klaren und übersichtlichen Syntax als leicht erlernbar gilt,
2. damit in vielen Disziplinen mittlerweile zu einem wissenschaftlichen Standard geworden ist,
3. in all ihren Grundlagen offen und damit kostenlos zu verwenden ist,
4. eine umfangreiche Standardbibliothek und zahlreiche frei zugängliche Spezialmodule umfasst,
5. mit dem Modul *Jupyter Notebook* (Erklärung dazu weiter unten) eine leicht installierbare und komfortabel im Browser laufende Programmierumgebung zur Verfügung stellt,
6. und von einer sehr großen Community beständig weiterentwickelt und reichhaltig mit überaus hilfreichen Einsteiger- und Fortgeschrittenen-Tipps versorgt wird.

Was tut eine Programmiersprache?

Eine Programmiersprache dient dazu, Aufgaben, die sich ein Mensch stellt, an eine digitale Maschine zu vermitteln, die diese Aufgaben ausführen soll. Digitale Maschinen arbeiten auf der Basis von Binärzuständen, d.h. sie reagieren im Prinzip auf Abfolgen von elektrischen Pulsen, die keine Gewichtung (stärker oder schwächer), sondern nur *an* oder *aus* kennen, also etwa die Zustände „Strom fließt“ oder „Strom fließt nicht“. Auf einer ersten Ebene (Maschinensprache) werden diese elektrischen Pulse in Binärzahlen (0 oder 1) übersetzt. Da diese „Sprache“ allerdings für Menschen schwer zu lesen ist, wird diese Codierung in weiteren Vermittlungsstufen an ein von Menschen verstehbares Niveau herangeführt. Dabei stellt sich die Aufgabe, einen Kompromiss zu finden zwischen „hinreichend allgemein, um möglichst viele verschiedene Dinge ausführen zu können“ und „hinreichend konkret, um nicht alles von Anfang an selbst programmieren zu müssen“. Darüber hinaus soll dieser Kompromisssprache dann auch noch leicht verständlich und leicht erlernbar sein.

Dies ist kein leichtes Unterfangen: einen solchen Kompromiss stellt eben die Programmiersprache Python dar.

Python-Versionen – 2.x oder 3.x

Python wird, wie viele Open Source-Programmiersprachen, beständig und sehr intensiv weiter entwickelt. Dies ist einerseits gut, weil die Sprache damit stets auf neuestem Stand bleibt, auf Fehler

überprüft und korrigiert wird, und neue Features hinzugefügt werden. Andererseits sind neuere Versionen unter Umständen nicht immer mit älteren kompatibel. Was Python betrifft so gibt es aktuell zwei Entwicklungsstränge, Python 2.x und Python 3.x, die zwar sehr ähnlich sind, aber doch nicht in allen Punkten ident und damit auch nicht kompatibel (d.h. Programme die z.B in Python 2.7 geschrieben sind, laufen nicht notwendigerweise in der gleichen Weise auch in Python 3.6.). Dies ist bei Installation zu beachten (siehe unten dazu mehr). Genaueres zu den Unterschieden findet sich unter anderem² unter: <https://wiki.python.org/moin/Python2orPython3>

Für unsere Zwecke sind die Unterschiede aber weitgehend vernachlässigbar. Wir bemühen uns überdies, die Programmierbeispiele in diesem Skriptum so zu verfassen, dass sie in beiden Entwicklungssträngen gleichermaßen verwendet werden können.

Module (Pakete)

Python ist, wie die meisten Programmiersprachen, modular aufgebaut. Was heißt das?

Programmcodes lassen sich zu kleinen oder größeren Einheiten zusammenbinden, um dann als solche immer wieder verwendet zu werden. Das funktioniert ähnlich wie in unserer Alltagssprache, wo die Buchstaben zu wiederverwendbaren Worten zusammengestellt werden, um dann aus diesen Worten wiederum Sätze zu bauen, und aus den Sätzen längere Texte, und aus diesen Bücher, usw.

Wenn mit Python zum Beispiel die Aufgabe, den Absolutwert einer Zahl zu bilden, wiederholt ausgeführt werden soll, so macht es Sinn, nicht jedes Mal einzeln die Zahl zunächst zu quadrieren um sodann wieder die Wurzel daraus zu ziehen, sondern das Quadrieren und Wurzelziehen zu einer sogenannten Funktion zusammenzuziehen und z.B. unter dem Namen *abs* abzuspeichern, sodass sie jedes Mal, wenn ein Absolutwert benötigt wird, leicht zur Anwendung gebracht werden kann. In Python könnte dies wie folgt aussehen (Keine Angst, dies soll hier einstweilen nur ein erstes Beispiel liefern, um die Modularität zu erklären. Der Code wird später erklärt):

```
# hier wird die Funktion abs allgemein definiert
def abs(x):
    return (x**2)**(0.5)

# hier wird die Funktion abs mit einer konkreten Zahl (-17) zur Anwendung gebracht
abs(-17)
```

17.0

Das heißt, Python lässt sich verwenden, um wiederverwendbare Einheiten zu bauen, die auf das Ausführen bestimmter Aufgaben spezialisiert sind. Genau dies wird seit Anbeginn der Python-Entwicklung von einer großen Community von Programmierern getan. Diese Programmierer bauen kleinere Python-Code-Elemente zu größeren funktional-spezialisierten Einheiten zusammen. Solche Einheiten haben je nach Größe und Ordnungsniveau unterschiedliche Namen (z.B. *function*, *class*, ...), folgen aber stets dem gleichen Prinzip: große Einheiten bestehen aus kleineren, die ihrerseits wieder aus noch kleineren Einheiten zusammengesetzt sind, usw. Das ist das Prinzip der Modularität.

Auf recht hoher Ebene werden solche Module auch Pakete oder *packages* bzw. *libraries* genannt und ihrerseits nochmals zu Paket-Sammlungen zusammengestellt. Eine solche Paket-Sammlung, die sehr

² Aufgrund der großen Anwender-Community ist es im Allgemeinen sehr leicht, Informationen zu speziellen Aspekten von Python, insbesondere auch zu Programmier-Fragen, zu finden. Man google einfach zum Beispiel „unterschied python 2 und 3“

prominent ist und viele solche *packages* vereint, und die darüber hinaus im Prinzip auch alles enthält, was wir hier zum Programmieren-Lernen mit Python brauchen, heißt **Anaconda**.

Anaconda

Anaconda (<https://www.continuum.io>) ist eine sogenannte Python-Distribution, die eine Vielzahl von Werkzeugen vereint, die für das Programmieren mit Python notwendig sind. Anaconda ist frei (d.h. kostenlos), sehr leicht zu installieren, und verfügbar für Windows, Mac-OS und Linux. Die letzte Version (4.4.0 vom 31.5.2017³) enthält mit über 150 mitgelieferten Modulen (*packages*) im Prinzip alles (und viel mehr), was zum Programmieren im Bereich der USW-Systemwissenschaften nötig ist. Vor allem enthält Anaconda bereits auch eine sehr komfortable Ein- und Ausgabe-Umgebung (d.h. eine Entwicklungsumgebung) namens **Jupyter Notebook**, die einfach im gewohnten Browser läuft, wie er für das tägliche Surfen im Internet verwendet wird (Google-Chrome, Firefox, Internetexplorer ...). Das heißt, nach Installation von Anaconda sind im Prinzip keine zusätzlichen oder ungewohnten Programme mehr notwendig, um sofort mit Python zu arbeiten zu beginnen.

Im Folgenden wird nun der Installationsvorgang für Anaconda mit Python 3.6. auf einem Windows-Computer beschrieben.

Eine weitere Windows-Installationsbeschreibung findet sich hier: <https://docs.continuum.io/anaconda/install-windows>, für die Mac-OS-Installation siehe hier: <https://docs.continuum.io/anaconda/install-macOS>, für Linux siehe hier: <https://docs.continuum.io/anaconda/install-linux>

Anaconda-Installation für Python 3.6.

- Öffnen Sie den Link www.continuum.io/downloads in Ihrem Internet Browser und scrollen sie ein Stück nach unten bis sie die folgende Abbildung sehen.

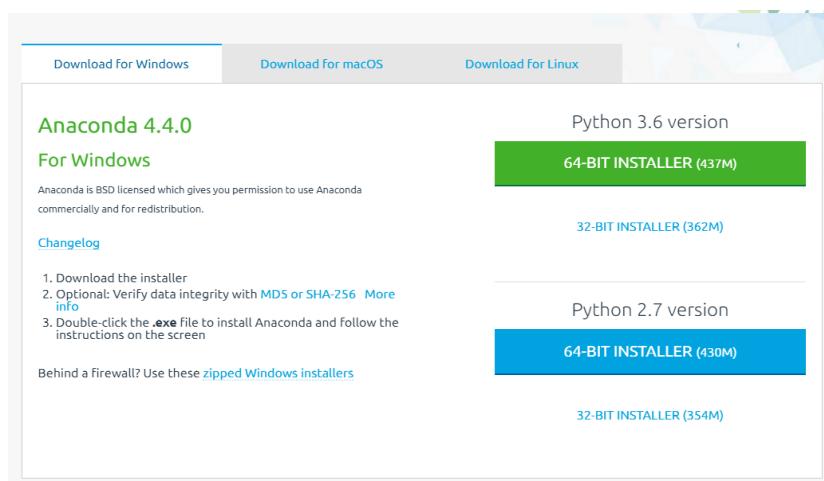


Abbildung 1: Download "Python 3.6 version"

³ Die aktuelle Anaconda-Versionen (ab 4.4.0) enthalten zusätzlich zu den Python- *packages* weitere spezialisierte Daten-Analyse-*packages* für R und für Stata. Für dieses Skriptum werden diese *packages* nicht benötigt. Wer auf seinem Computer Platz sparen will/muss, kann alternativ auch die Version **Miniconda** (<https://conda.io/miniconda.html>) installieren. Eine Kurz-Anleitung dazu findet sich hier: <https://conda.io/docs/install/quick.html>. In diesem Fall muss allerdings das Paket für das Jupyter Notebook (siehe unten) noch extra installiert werden. Eine Anleitung dazu findet sich hier: <http://jupyter.readthedocs.io/en/latest/install.html>.

- Klicken sie, wenn sie einen entsprechenden Computer⁴ haben, auf den Button „64-BIT INSTALLER“ um Python 3.6 Version zu erhalten. Ansonsten wählen sie die 32-BIT-Version. Daraufhin startet der Download der Installationsdatei. Dies kann einige Minuten in Anspruch nehmen.
- Achten sie darauf, dass der Pfad zu dem Ordner, in dem sie ihre downloads speichern, und auch der Ordnername selbst keine Sonderzeichen (wie Umlaute etc.) oder Leerzeichen enthält. Sollte dies der Fall sein, so kopieren sie die heruntergeladene Datei bevor sie sie ausführen in einen entsprechenden Ordner.
- Doppelklicken Sie auf die .exe Datei, die sie heruntergeladen haben.
- Klicken Sie im automatisch neu geöffneten Fenster auf den Button „Ausführen“.
- Folgen sie den weiteren Dialog-Fenstern.
- Im *Licence Agreement*-Fenster klicken Sie auf den Button „I Agree“, wenn sie den Lizenzbedingungen zustimmen.
- Folgen sie erneut den Dialog-Fenstern. Wir empfehlen die jeweils vorgeschlagenen Angaben zu übernehmen.
- Im Fenster *Choose Install Location* sollten sie erneut darauf achten, dass der Pfad zu dem Ordner, in den sie Anaconda installieren wollen, und auch der Ordnername selbst keine Sonderzeichen (Umlaute etc.) oder Leerzeichen enthalten. Mit einem Klick auf den Button „Browse...“ können Sie den automatisch vorgeschlagenen Ordner ändern und Ihren gewünschten Speicherort auswählen. Klicken sie erneut „Next >“ um mit der Installation fortzufahren.
- Klicken Sie auf den Button „Install“ und schließlich „Finish“ um die Installation fertigzustellen.

Nach erfolgreicher Installation sollten sie in Ihrem Windows Startmenü einen Menüpunkt *Anaconda* sehen, der seinerseits einen Menü-Unterpunkt *Jupyter Notebook* enthält.

Jupyter Notebook

Das Jupyter Notebook ist eine sehr komfortable Ein- und Ausgabe-Umgebung für die Programmiersprache Python, d.h. eine Entwicklungsumgebung, die im bereits auf ihrem Computer installierten Browser läuft. Alle in diesem Skriptum vorkommenden Beispiele wurden mit dem Jupyter Notebook geschrieben und werden auch in diesem angezeigt.

Klicken Sie, um das Jupyter Notebook zu starten, im Windows-Startmenü unter Anaconda auf den Menüpunkt Jupyter Notebook.

Es öffnet sich ein Fenster, das zuerst schwarz ist, und sich dann mit weißem Text füllt. Dieses Fenster darf nicht geschlossen werden, solange Sie mit dem Jupyter Notebook arbeiten möchten. Es handelt sich hierbei um eine Anzeige für den so genannte *Kernel*, also den Prozess, in dem die eigentlichen Berechnungen durchgeführt werden. Zusätzlich öffnet sich ihr Browser, bzw. ein neuer Tab in ihrem Browser.

⁴ Im Startmenü können sie unter >Systemsteuerung >System >Systemtyp feststellen, welche Art von Betriebssystem sie haben.

Um ein neues Jupyter Notebook zu öffnen, klicken sie im rechten Bereich dieses Browser-Tabs auf das Dropdown-Menü „New“ und wählen sie, je nachdem, was sie installiert haben „Python 2“ oder „Python 3“ aus.

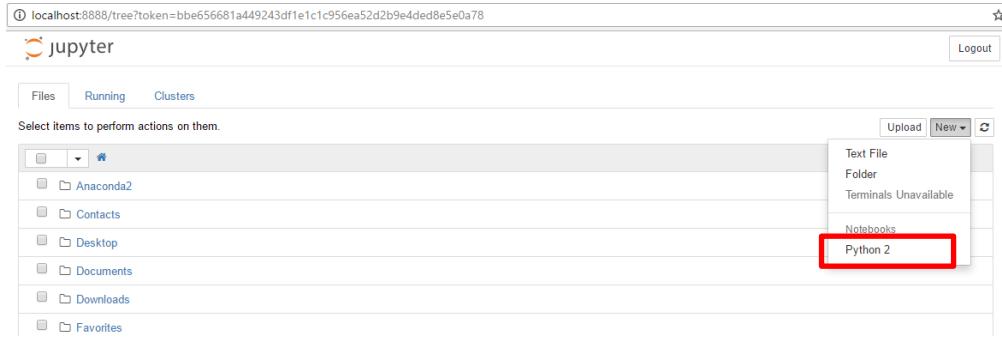


Abbildung 2: Jupyter Dashboard

Ein neues Browserfenster mit der Notebook Umgebung wird geöffnet.

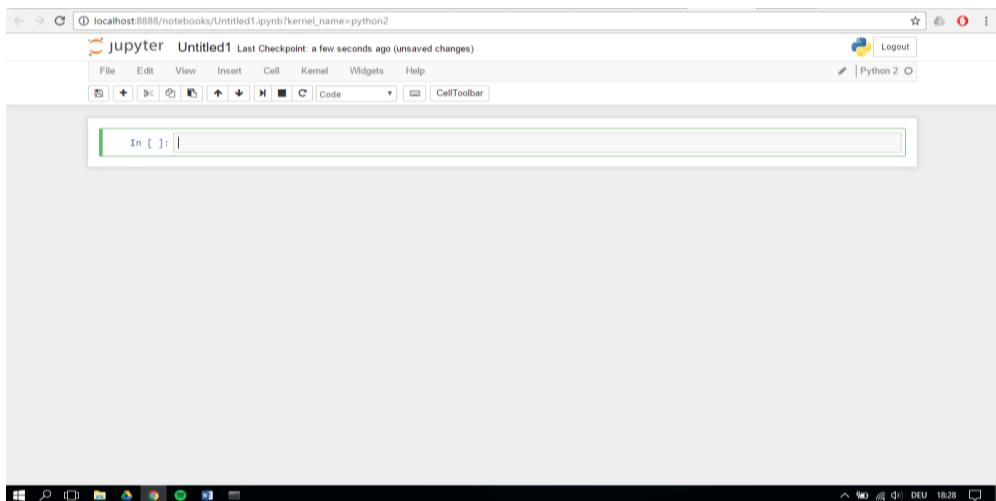
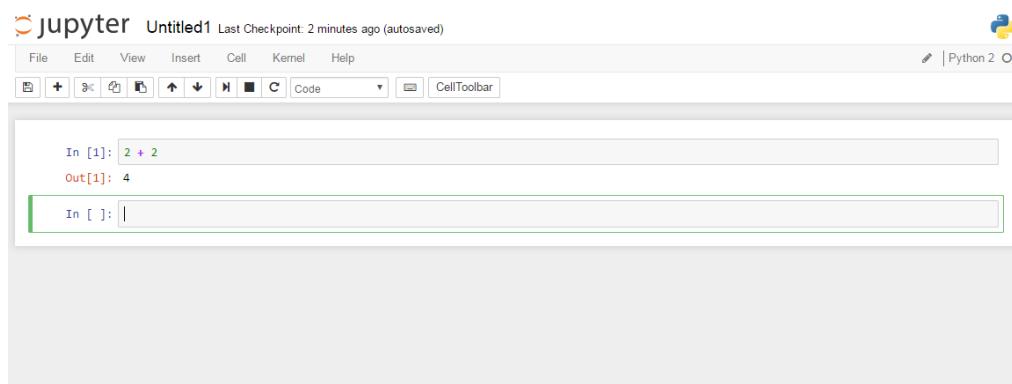


Abbildung 3: Jupyter Notebook Eingabe-Fenster

Um das Notebook zu testen, schreiben sie in dieses Fenster, dort wo der blinkende Cursor steht, die Aufgabe $2 + 2$ und drücken sie gleichzeitig die Shift- (\downarrow) und die Entertaste (\leftarrow) auf ihrem Keyboard.



Kapitel 1 – Einführung in Python – Der Froschteich

Verwenden von Jupyter-Notebooks

Jupyter-Notebooks bestehen aus Zellen. Es gibt drei Typen von Zellen:

- Textzellen, so wie diese hier, in welchen man Texte schreiben und formatieren kann
- In-Zellen, in die man Python-Code schreiben kann
- Out-Zellen, die das Ergebnis der darüberstehenden In-Zelle ausgeben

Um eine In-Zelle auszuführen klickt man einfach in die gewünschte Zelle und drückt die Shift- und die Enter-Taste gleichzeitig. Es wird automatisch eine Out-Zelle produziert, in der das Ergebnis der In-Zelle steht.

Auch ohne jegliche Programmierkenntnisse kann man so schon Python benutzen, wenn auch nur als Taschenrechner:

```
In [1]: 7 * 7 - 7
Out[1]: 42
```

```
In [2]: ((12 + 144 + 20 + 3 * 4**0.5) / 7) + 5 * 11
# Exponenten werden in Python als x**y angegeben. 4**0.5 bedeutet also 4 hoch 0.5.
# Der Text hinter einer Raute (#) wird vom Computer ignoriert und ist als Information für den Leser gedacht.
```

```
Out[2]: 81.0
```

Python kann aber natürlich viel mehr als ein Taschenrechner. In einer Zelle kann man nicht nur einzelne Rechnungen schreiben, sondern ganze Programme oder vollständige Simulationen. Versuchen wir eine ganz simple Simulation zu erstellen, um die Grundbefehle von Python zu lernen. Nehmen wir an, wir wollen die Populationsentwicklung in einem Froschteich simulieren. In allerster Näherung gehen wir davon aus, dass sich jedes Jahr 3 neue Frösche im Teich ansiedeln.

Variablen definieren

Um Zahlen (wie z.B. die Anzahl der Frösche in einem Teich) zu speichern muss eine so genannte Variable angelegt werden. Das ist sozusagen der Name unter dem die Zahl gespeichert ist. Die Zuweisung von Variablenname zur Zahl passiert in Python mit dem = Zeichen. Wichtig ist, dass der Name links vom = Zeichen steht, die Zahl die dort gespeichert werden soll, rechts. Wir starten unsere Teichsimulation also indem wir die Anzahl der Frösche auf 0 setzen.

```
In [3]: froschanzahl = 0
```

Variablen verändern

Um eine Variable zu ändern kann man einfach den aktuellen Wert mit dem neuen Wert überschreiben. Auch das geschieht mit dem = Zeichen.

```
In [4]: # Jahr 1:
froschanzahl = 3
```

Variablen abrufen

Um Variablen wieder anzuzeigen benutzt man den Befehl print. Dieser Befehl schreibt den aktuellen Wert der abgefragten Variable in die Ausabezeile. Schreiben wir also print(froschanzahl), sollte das das Ergebnis 3 bringen.

```
In [5]: print(froschanzahl)
3
```

Das ist sehr hilfreich für unsere weitere Simulation, denn so können wir die aktuelle Zahl der Frösche benutzen, um die neue zu berechnen. Laut unserer Annahme ist die Zahl der Frösche in jedem Jahr um 3 höher als zuvor, mathematisch ausgedrückt also froschanzahl + 3. Somit sieht unsere Froschsimulation so aus:

```
In [6]: froschanzahl = 0
#Jahr 1
froschanzahl = froschanzahl + 3
#Jahr 2
froschanzahl = froschanzahl + 3
#Jahr 3
froschanzahl = froschanzahl + 3
#Jahr 4
froschanzahl = froschanzahl + 3
#Jahr 5
froschanzahl = froschanzahl + 3
#Um ein Ergebnis auszugeben benutzen wir den Befehl "print"
print(froschanzahl)
```

```
15
```

In dieser Version ist unsere Froschsimulation nicht nur sehr unspektakulär, sondern auch recht umständlich. Sie ist jedoch ein guter Ausgangspunkt für eine bessere Simulation. Als ersten Schritt wäre es schön, wenn wir die Anzahl der Jahre, die simuliert werden sollen, einstellen könnten und nicht immer exakt 5 Jahre simulieren müssen. Dazu benötigen wir ein wichtiges Konzept: die For-Schleife.

For-Schleifen

For-Schleifen werden benutzt um einen Befehl mehrmals hintereinander ausführen zu lassen, in unserem Fall das Hinzuzählen von Fröschen. So würde unser Programm mit einer For-Schleife aussehen:

```
In [7]: froschanzahl = 0
for i in range(5):
    froschanzahl = froschanzahl + 3
print(froschanzahl)
```

15

Das ist deutlich kompakter, dadurch aber auch ein wenig komplizierter. Die erste Zeile ist exakt gleich, wir setzen froschanzahl auf 0. In der nächsten Zeile leitet der Befehl for die For-Schleife ein. Lesen könnte man diese Zeile als: Mache folgenden Befehl für jede ganze Zahl i, die kleiner ist als 5, also insgesamt 5 mal (für 0,1,2,3,4). Nach dem Doppelpunkt kommt in der nächsten Zeile der Befehl, der ausgeführt werden soll. Zum Schluss lassen wir uns wieder das Ergebnis ausgeben.

Einrücken in Python

Woher weiß Python jetzt aber, dass wir den letzten Befehl (das print) nicht auch 5 mal ausgeführt haben wollen? Das passiert mittels Einrückungen (tab-Taste). Befehle, die direkt untereinander stehen gehören für Python zusammen. Das macht Pythoncode automatisch gut lesbar und man benötigt keine Klammern. Würden wir die Ausgabe wirklich gerne nach jedem Jahr, und nicht erst am Schluss haben, können wir den print-Befehl mit der Tabulatortaste einrücken. So gehört er zur For-Schleife:

```
In [8]: froschanzahl = 0
for i in range(5):
    froschanzahl = froschanzahl + 3
    print(froschanzahl)
```

3
6
9
12
15

Ob Befehle innerhalb oder außerhalb einer For-Schleife stehen macht einen großen Unterschied und ist eine häufige Fehlerquelle. Würde man beispielsweise auch die Zeile froschanzahl = 0 in die For-Schleife einbauen, würde die Zahl der Frösche bei jedem Durchlauf der Schleife auf 0 gesetzt werden:

```
In [9]: for i in range(5):
    froschanzahl = 0
    froschanzahl = froschanzahl + 3
    print(froschanzahl)
```

3
3
3
3
3

Dennoch bietet das Verwenden einer For-Schleife einen riesigen Vorteil: Wir müssen nur mehr eine einzige Zahl ändern um die Zahl der simulierten Jahre zu verändern. Wollen wir beispielweise 10 Jahre simulieren:

```
In [10]: froschanzahl = 0
for i in range(10):
    froschanzahl = froschanzahl + 3
print(froschanzahl)
```

30

Zu einem schönen Stil beim Programmieren gehört es, solche Zahlen, die man öfter ändern möchte an den Anfang des Programms zu stellen. Wir definieren also eine neue Variable, damit wir die Zahl nicht mitten im Code ändern müssen, sondern schön übersichtlich am Anfang. Mit einem Kommentar wissen auch zukünftige Benutzer, was diese Variable genau macht.

```
In [11]: simulationszeit = 10
#simulationszeit: Die Zeit in Jahren, die der Froschteich simuliert wird

froschanzahl = 0
for i in range(simulationszeit):
    froschanzahl = froschanzahl + 3
print(froschanzahl)
```

30

Mit diesem Programm können wir nun auch sehr lange Zeitbereiche simulieren:

```
In [12]: simulationszeit = 1000
#simulationszeit: Die Zeit in Jahren, die der Froschteich simuliert wird

froschanzahl = 0
for i in range(simulationszeit):
    froschanzahl = froschanzahl + 3
print(froschanzahl)
```

3000

In dieser Simulation ist das Endergebnis weniger spannend als die zeitliche Entwicklung. Es wäre also interessant die Froschanzahl zu jedem Zeitpunkt auszugeben. Am einfachsten verschieben wir den `print` Befehl in die For-Schleife:

```
In [13]: simulationszeit = 30
#simulationszeit: Die Zeit in Jahren, die der Froschteich simuliert wird

froschanzahl = 0
for i in range(simulationszeit):
    froschanzahl = froschanzahl + 3
    print(froschanzahl), #ein Beistrich hinter einem Print-Befehl schreibt die Ergebnisse, statt untereinander, in eine Zeile

3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 75 78 81 84 87 90
```

Auf diese Weise sieht man wie sich die Froschpopulation entwickelt, da wir jeden Wert einzeln ausgegeben haben. Der Nachteil an dieser Variante: Wirklich gespeichert haben wir immer nur eine Zahl. Am Ende der Simulation sehen wir zwar viele Zahlen, gespeichert (und somit zum Weiterarbeiten verfügbar) ist aber nur die letzte. Eine schönere Lösung ist das Verwenden von einer Liste.

Listen in Python

Listen bestehen aus mehreren aufeinanderfolgenden Werten. Um eine Liste zu erstellen, fängt man am besten mit einer leeren Liste an und hängt dann immer wieder neue Elemente an. Auch Listen bekommen, so wie Variablen, einen eindeutigen Namen. Leere Listen erstellt man mit `name = []` und neu Einträge fügt man mit dem `append`-Befehl hinzu. Für unser Beispiel:

```
In [14]: simulationszeit = 30
#simulationszeit: Die Zeit in Jahren, die der Froschteich simuliert wird

froschanzahl = 0
froschanzahl_liste = [] #Leere Liste wird erstellt
froschanzahl_liste.append(froschanzahl) #erster Eintrag (0 Frösche) wird angehängt

for i in range(simulationszeit):
    froschanzahl = froschanzahl + 3
    #am Ende jedes Schleifendurchgangs wird die aktuelle Zahl an die Liste angehängt
    froschanzahl_liste.append(froschanzahl)
print(froschanzahl_liste)

[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90]
```

Nun kann man schon grob erkennen was passiert, viel vorstellen kann man sich aber noch nicht. Schön wäre eine grafische Darstellung. Und das ist in Python zum Glück sehr einfach.

Grafiken in Python

Der Befehl zum Erstellen von Plots lautet "plot". Dieser Befehl ist jedoch nicht standardmäßig in jedem Pythonprogramm vorhanden, sondern muss in der Regel erst aus einem sogenannten Paket importiert werden. Pakete sind sozusagen Sammlungen von Befehlen.

Der `plot` Befehl selbst funktioniert dann ganz einfach: In Klammer schreibt man einfach die Liste von Zahlen, die man darstellen möchte.

```
In [15]: import matplotlib.pyplot as plt #wir importieren das Modul matplotlib.pyplot und geben ihm die Abkürzung plt
# die nächste Zeile bewirkt, dass die Grafiken direkt in der Zelle angezeigt werden
%matplotlib inline

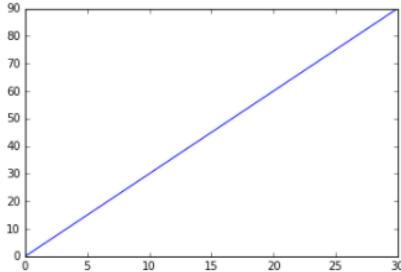
simulationszeit = 30
#simulationszeit: Die Zeit in Jahren, die der Froschteich simuliert wird

froschanzahl = 0
froschanzahl_liste=[] #leere Liste wird erstellt
froschanzahl_liste.append(froschanzahl) #erster Eintrag (0 Frösche) wird angehängt

for i in range(simulationszeit):
    froschanzahl = froschanzahl + 3
    #am Ende jedes Schleifendurchgangs wird die aktuelle Zahl an die Liste angehängt
    froschanzahl_liste.append(froschanzahl)

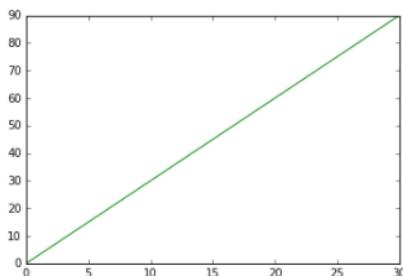
plt.plot(froschanzahl_liste)
```

Out[15]: [`<matplotlib.lines.Line2D at 0xae24ef0>`]



Eine Grafik ist schon viel anschaulicher als eine Liste von Zahlen. Natürlich kann man diese Grafik noch deutlich verbessern. Bei Fröschen würde es sich beispielsweise anbieten, die Linie grün darzustellen. Solche zusätzlichen Optionen kann man im Plot-Befehl einfach nach einem Beistrich anfügen:

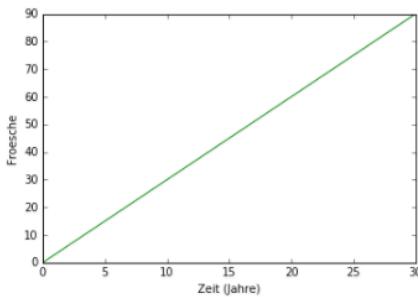
```
In [16]: plt.plot(froschanzahl_liste, color='green')
Out[16]: [<matplotlib.lines.Line2D at 0xb0f9828>]
```



Man beachte, dass man die Farbe selbst unter Anführungsstrichen schreiben muss, da das Programm sonst nach einer Variable suchen würde, die green heißt.

Eine weitere mögliche Verbesserung wäre eine Beschriftung der Achsen:

```
In [17]: plt.plot(froschanzahl_liste, color='green')
plt.xlabel('Zeit (Jahre)')
plt.ylabel('Froesche')
Out[17]: <matplotlib.text.Text at 0xb16b828>
```



Zusammenfassung

Variablen

Durch Variablen können wir Werte einem Namen zuordnen. Mit

```
varname = 42
```

weisen wir der Variable varname den Wert 42 zu und überschreiben den aktuell dort gespeicherten Wert, wenn dort schon etwas gespeichert ist.

Mit

```
varname = varname + 1
```

erhöhen wir den Wert, der unter varname gespeichert ist um 1.

Den aktuellen Wert der Variable varname kann mit

```
print(varname)
```

angezeigt werden.

For-Schleifen

Mit For-Schleifen ist es möglich gleiche oder ähnliche Befehl oftmals hintereinander auszuführen. Um einen Befehl also beispielsweise 100 mal abarbeiten zu lassen, verwenden wir:

```
for i in range(100):
    befehl
```

Man beachte die Einrückung des Befehls, die zeigt, dass sich der Befehl innerhalb der Schleife befindet.

Listen

In Listen können mehrere Werte unter nur einem Namen abgespeichert werden. Leere Listen erstellt man mit

```
listename = []
```

Einen neuen Eintrag (z.B. neueselement) zur Liste hinzufügen kann man dann mit

```
listename.append(neueselement)
```

Grafiken

Um innerhalb eines Jupyter-Notebooks den Plotbefehl benutzen zu können, verwenden wir die Zeilen

```
import matplotlib.pyplot as plt
```

```
%matplotlib inline
```

am Anfang des Programms. Danach können wir mit dem Befehl

```
plt.plot(listenname)
```

die Liste mit dem Namen `listenname` grafisch darstellen. Zusatzoptionen wie Farbe können nach einem Beistrich übergeben werden:

```
plt.plot(listenname, color='green')
```

Kapitel 2 – Populationsentwicklungen

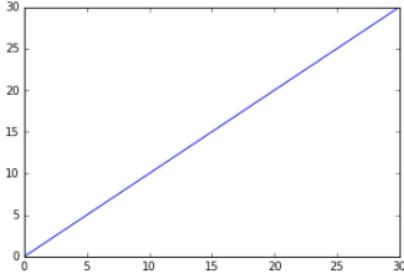
In dieser Einheit möchten wir uns genauer mit verschiedenen Möglichkeiten der Populationsentwicklung auseinandersetzen. Mit dem Beispiel des Froschteiches haben wir bereits eine sehr einfache Form der Populationsentwicklung kennengelernt, so genanntes:

Lineares Wachstum

Wir ziehen nun als weiteres Beispiel die Vermehrung von Kaninchen heran.

```
In [1]: import matplotlib.pyplot as plt #wir importieren das paket matplotlib.pyplot und geben ihm die abkürzung plt  
#die nächste Zeile bewirkt, dass die Grafiken direkt in der Zelle angezeigt werden  
%matplotlib inline  
  
simulationszeit = 30  
#simulationszeit: Die Zeit in Jahren, die die Kaninchen simuliert werden  
  
kaninchenanzahl = 0  
kaninchenanzahl_liste = [] #leere Liste wird erstellt  
kaninchenanzahl_liste.append(kaninchenanzahl) #erster Eintrag (0 Kaninchen) wird angehängt  
  
for i in range(simulationszeit):  
    kaninchenanzahl = kaninchenanzahl + 1  
    #am Ende jedes Schleifendurchgangs wird die aktuelle Zahl an die Liste angehängt  
    kaninchenanzahl_liste.append(kaninchenanzahl)  
  
plt.plot(kaninchenanzahl_liste)
```

Out[1]: [<matplotlib.lines.Line2D at 0xadada20>]



Diese Form von Wachstum war für einen Froschteich zwar eine gute Näherung, wenn wir annehmen, dass die Frösche dort einfach zwandern. Unsere Kaninchenpopulation vermehrt sich aber durch Fortpflanzung. Das heißt, je mehr Kaninchen es gibt, um so schneller kommen neue hinzu.

Eine interessante Näherung, um die Vermehrung von Kaninchen zu modellieren, ist die sogenannte

Fibonacci-Folge

In diesem Modell errechnet sich die Anzahl der Kaninchen im aktuellen Jahr aus der Anzahl der Kaninchen aus dem Vorjahr plus der Anzahl der Kaninchen aus dem vorletzten Jahr. Um so etwas zu programmieren, müssen wir aber zuerst mehr über Listen in Python lernen:

Listen und Schleifen für Fortgeschritten

Bislang wissen wir, wie man leere Listen erstellt und Elemente zu Listen hinzufügt. Man kann aber natürlich auch spezifische Listeneinträge abfragen. Möchte man beispielsweise das siebente Element der Liste kaninchenanzahl_liste wissen, kann man es mit kaninchenanzahl_liste[6] abrufen. Warum nicht 7? In Programmiersprachen ist es üblich, das erste Element einer Liste stets mit der Nummer 0 zu speichern, in diesem Fall also als kaninchenanzahl_liste[0]. Das ist anfangs zwar verwirrend, für spätere Anwendungen jedoch eine recht sinnvolle Konvention.

```
In [2]: kaninchenanzahl_liste[7]
```

Out[2]: 7

For-Schleifen haben wir schon kennen gelernt, eine wichtige Eigenschaft haben wir aber bislang ignoriert. Die Zahl *i*, die sozusagen mitzählt wie oft ein Befehl schon ausgeführt worden ist, darf auch innerhalb eines Befehls benutzt werden:

```
In [3]: for i in range(5):  
    print(i)
```

0

1

2

3

4

Wenn wir diese beiden Kenntnisse nun kombinieren, haben wir eine Möglichkeit gefunden, wie wir innerhalb einer Schleife auf vorherige Listeneinträge zugreifen können. Das ist genau das, was wir für unsere Fibonacci-Folge benötigen.

Eine weitere wichtige Eigenschaft von for-Schleifen ist, dass *i* nicht unbedingt bei 0 anfangen muss. Möchten wir beispielsweise erst ab Jahr 2 simulieren, so können wir die Schleife auch erst bei Jahr zwei beginnen lassen:

```
In [4]: for i in range(2,5):
    print(i)

2
3
4

In [5]: import matplotlib.pyplot as plt
%matplotlib inline

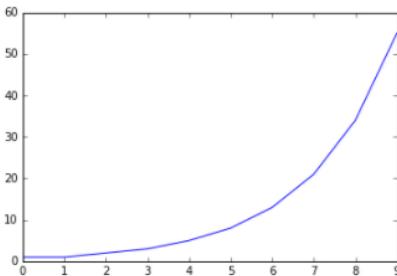
simulationszeit = 10
# Simulationszeit: Die Zeit in Jahren, die die Kaninchenfortpflanzung simuliert wird

kaninchenanzahl = 1
kaninchenanzahl_liste = [] #leere Liste wird erstellt
kaninchenanzahl_liste.append(kaninchenanzahl) #erster Eintrag (1 Kaninchen) wird angehängt
kaninchenanzahl = 1
kaninchenanzahl_liste.append(kaninchenanzahl) #zweiter Eintrag (1 Kaninchen) wird angehängt
#wir benötigen hier zwei Kaninchen, damit sie sich auch vermehren können

for i in range(2, simulationszeit):
    letztesjahr = kaninchenanzahl_liste[i - 1] #Kaninchenanzahl vom letzten Jahr wird berechnet
    vorletztesjahr = kaninchenanzahl_liste[i - 2] #Kaninchenanzahl vom vorletzten Jahr wird berechnet
    kaninchenanzahl = letztesjahr + vorletztesjahr #neue Kaninchenanzahl wird berechnet
    #am Ende jedes Schleifendurchgangs wird die aktuelle Zahl an die Liste angehängt
    kaninchenanzahl_liste.append(kaninchenanzahl)

plt.plot(kaninchenanzahl_liste)

Out[5]: [<matplotlib.lines.Line2D at 0xb05ff28>]
```



So haben wir mit sehr einfachen Mitteln schon eine relativ gute Simulation der Kaninchenvermehrung erstellt. Die Fibonacci-Folge hat aber einen gravierenden Nachteil: Die Kaninchen vermehren sich immer gleich schnell. Wir haben keine Möglichkeit Umwelteinflüsse (Vorhandensein von Nahrung oder Fressfeinden) einzubauen. Aus diesem Grund werden wir unser Model nun verbessern.

Wir haben gesehen, dass das Model realistischer wurde, sobald wir die Vermehrung der Kaninchen von der je aktuellen Zahl der Kaninchen abhängig machen. Das geht auf viele verschiedene Arten. Eine Möglichkeit ist, dass sich die Anzahl der Kaninchen jedes Jahr verdoppelt.

Mathematisch ausgedrückt: kaninchen = kaninchen * 2

```
In [6]: import matplotlib.pyplot as plt
%matplotlib inline

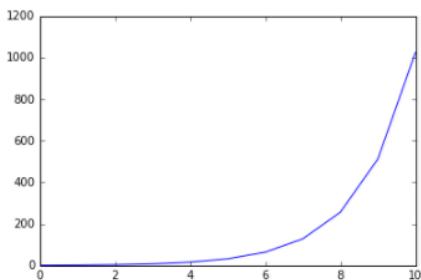
simulationszeit = 10
#simulationszeit: Die Zeit in Jahren, die die Kaninchen simuliert werden

kaninchenanzahl = 1
kaninchenanzahl_liste = [] #leere Liste wird erstellt
kaninchenanzahl_liste.append(kaninchenanzahl) #erster Eintrag (1 Kaninchen) wird zur leeren Liste hinzugefügt

for i in range(simulationszeit):
    kaninchenanzahl = kaninchenanzahl * 2 #neue Kaninchenanzahl wird berechnet
    #am Ende jedes Schleifendurchgangs wird die aktuelle Zahl an die Liste angehängt
    kaninchenanzahl_liste.append(kaninchenanzahl)

plt.plot(kaninchenanzahl_liste)

Out[6]: [<matplotlib.lines.Line2D at 0xb299518>]
```



So eine Art von Wachstum nennt man:

Exponentielles Wachstum

Im Vergleich zur Fibonacci-Folge kommt man so auf sehr viel mehr Kaninchen. Natürlich ist aber "verdoppeln" nicht die einzige mögliche Variante. Es könnten zum Beispiel auch jedes Jahr nur 10% hinzukommen.

Mathematisch ausgedrückt: kaninchen = kaninchen + kaninchen * 0.1

```
In [7]: import matplotlib.pyplot as plt
%matplotlib inline

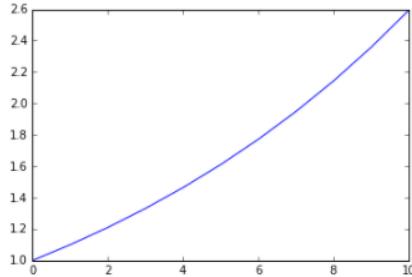
simulationszeit = 10
#simulationszeit: Die Zeit in Jahren, die die Kaninchen simuliert werden

kaninchenanzahl = 1
kaninchenanzahl_liste = [] #leere Liste wird erstellt
kaninchenanzahl_liste.append(kaninchenanzahl) #erster Eintrag (1 Kaninchen) wird angehängt

for i in range(simulationszeit):
    kaninchenanzahl = kaninchenanzahl + kaninchenanzahl * 0.1 #neue kaninchenanzahl wird berechnet
    #am ende jedes schleifendurchgangs wird die aktuelle zahl an die liste angehängt
    kaninchenanzahl_liste.append(kaninchenanzahl)

plt.plot(kaninchenanzahl_liste)
```

Out[7]: [<matplotlib.lines.Line2D at 0xb4dada0>]



Hier wächst die Zahl der Kaninchen eher langsam. Dieser eine Faktor (hier 2 oder 1.1) bestimmt also, wie schnell sich die Kaninchen vermehren. Man spricht in diesem Fall von einem "Parameter", im Speziellen von einem **Wachstumsparameter**, also einem Parameter, der angibt, wie schnell die Population anwächst.

Unser Modell funktioniert in ähnlicher Weise für jeden beliebigen Wachstumsparameter. Somit können wir damit sowohl schnelles, als auch langsames Wachstum beschreiben.

```
In [8]: import matplotlib.pyplot as plt
%matplotlib inline

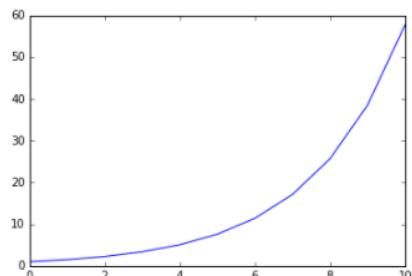
simulationszeit = 10
#simulationszeit: Die Zeit in Jahren, die die Kaninchen simuliert werden
wachstumsparameter = 0.5
#wie schnell vermehren sich die kaninchen

kaninchenanzahl = 1
kaninchenanzahl_liste = [] #leere Liste wird erstellt
kaninchenanzahl_liste.append(kaninchenanzahl) #erster Eintrag (1 Kaninchen) wird angehängt

for i in range(simulationszeit):
    kaninchenanzahl = kaninchenanzahl + kaninchenanzahl * wachstumsparameter #neue kaninchenanzahl wird berechnet
    #am ende jedes schleifendurchgangs wird die aktuelle zahl an die liste angehängt
    kaninchenanzahl_liste.append(kaninchenanzahl)

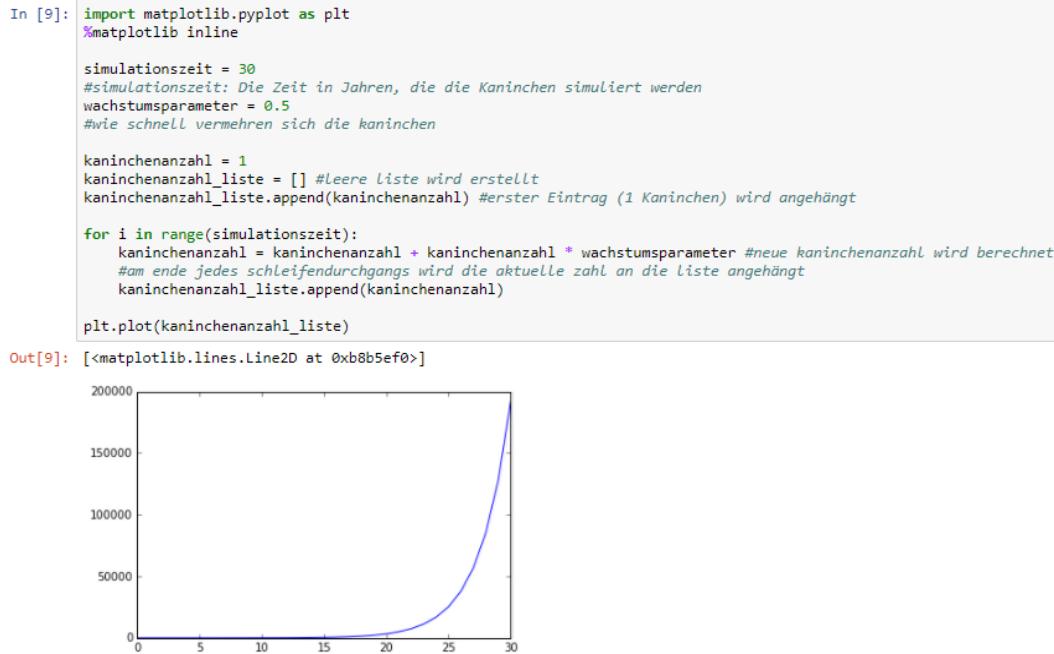
plt.plot(kaninchenanzahl_liste)
```

Out[8]: [<matplotlib.lines.Line2D at 0xb6f92b0>]



Wir können die Kaninchen nun also schon relativ realistisch simulieren und können sogar durch den Parameter festlegen, wie schnell sich die Kaninchen (aufgrund von Umwelteinflüssen) vermehren können.

Eine entscheidende Schwachstelle unseres Modells sieht man allerdings, sobald man die Simulationszeit etwas erhöht:



Es gibt keine obere Grenze für das Kaninchenwachstum! Die Kaninchen werden immer mehr! Nach 30 Jahren gibt es schon 200000 Kaninchen. Das ist natürlich ein sehr unrealistisches Verhalten. Irgendwann wird wohl jede Kaninchenpopulation langsamer wachsen, wenn zum Beispiel nur mehr wenig Nahrung oder Unterschlupf vorhanden ist. Und eventuell könnte die Kaninchenpopulation sogar konstant bleiben, wenn die natürliche Kapazität des Lebensraums ihre Grenze, die so genannte **Umweltkapazitätsgrenze** (Engl.: *carrying capacity*) erreicht hat.

Wenn wir auch diesen Effekt in unser Modell einbauen, erhalten wir ein so genanntes:

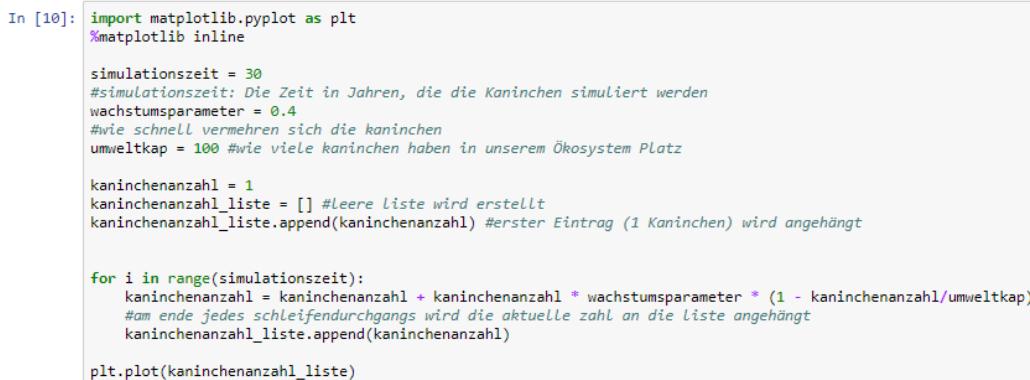
Logistisches Wachstum

Hierbei gibt es zusätzlich zum Wachstumsparameter, der ausdrückt wie schnell sich die Kaninchen vermehren, auch den Parameter "Umweltkapazität", der beschreibt, wie vielen Individuen das Ökosystem Platz bietet.

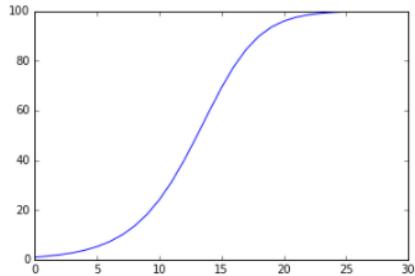
Die mathematische Formel für logistisches Wachstum ist ein wenig komplizierter. In Worte erklärt lautet sie in etwa: Der Bestand an Kaninchen vermehrt um einen Prozentsatz der Zahl der aktuell vorhandenen Kaninchen ($\text{kaninchen} * \text{wachstumsparameter}$). Und dieser Prozentsatz wird zusätzlich mit einer Größe multipliziert, die gegen 0 tendiert, je näher die Population der Umweltkapazitätsgrenze kommt ($1 - \text{kaninchen}/\text{umweltkap}$).

In einer Formel also

$$\text{kaninchen} = \text{kaninchen} + \text{kaninchen} * \text{wachstumsrate} * (1 - \text{kaninchen}/\text{umweltkap})$$



```
Out[10]: []
```



Diese Art von logistischem Wachstum kommt in der Natur sehr oft vor. Viele Populationsentwicklungen kann man mit einem solchen Modell abbilden. Auch sehr viel kompliziertere Modelle, wie beispielsweise **Räuber-Beute Modelle** bauen auf dieser Art von Wachstum auf.

Zusammenfassung

Lineares Wachstum

Bei linearem Wachstum kommt zu einer Population in jedem Zeitschritt eine konstante Anzahl hinzu:

```
kaninchen = kaninchen + x
```

wobei x angibt wie viele Individuen hinzukommen, also wie schnell die Population wächst.

Fibonacci-Folge

Bei der Fibonacci-Folge errechnet sich die neue Zahl immer aus den beiden letzten Zahlen. Um so etwas programmieren zu können, müssen wir mehr über for-Schleifen wissen:

Die Laufvariable (im Beispiel `i`) kann innerhalb der Schleife benutzt werden. So liefert die Schleife

```
for i in range(3):
    print(i)
das Ergebnis
0
1
2
```

Außerdem müssen wir wissen, wie man einzelne Elemente von Listen abruft:

Um die einzelnen Elemente einer Liste zu verwenden oder auszugeben, benutzt man eckige Klammern:

```
listename[4]
```

liefert zum Beispiel das dritte(!) Element. Achtung: Das erste Element in der Liste hat stets den Index 0, das zweite den Index 1, und so weiter.

Damit können wir nun Fibonacci-Zahlen berechnen:

```
kaninchenanzahl = kaninchenanzahl_liste[i - 1] + kaninchenanzahl_liste[i - 2]
```

Exponentielles Wachstum

Bei exponentiellem Wachstum kommt zur Population kein konstanter Wert hinzu, sondern ein Wert, der direkt von der aktuellen Population abhängt. Die genaue Abhängigkeit von Wachstum und momentaner Population gibt der so genannte **Wachstumsparameter** an:

```
kaninchen = kaninchen + kaninchen * wachstumsparameter
```

Logistisches Wachstum

Das logistische Wachstum verhält sich ähnlich wie das exponentielle Wachstum, nur gibt es hierbei eine **Umweltkapazitätsgrenze**, die beschränkt, wie groß eine Population maximal werden kann. Die Zahl der Individuen, die hinzukommen, wird mit einer Zahl multipliziert ($1 - \text{individuen}/\text{umweltkap}$), die mit der Annäherung an die Umweltkapazitätsgrenze gegen 0 tendiert. Ist diese Kapazitätsgrenze erreicht, so kommen keine neuen Individuen mehr hinzu und die Population bleibt stabil.

```
kaninchen = kaninchen + kaninchen * wachstumsrate * (1 - kaninchen/umweltkap)
```

Kapitel 3 – Wachstum mit Feedback – Die If-Bedingung

Während wir im vorhergehenden Kapiteln Wachstumsprozesse von nur einer Tierpopulation betrachtet haben, wollen wir uns nun ein vereinfachtes Ökosystem, dass aus zwei Tierarten besteht, ansehen. Es ist dies noch kein richtiges **Räuber-Beute-System**, wie wir es als Differenzen-, bzw. Differentialgleichungssystem ab Kapitel 8 in diesem Skriptum kennenlernen werden. Hier betrachten wir fürs erste ein stark vereinfachtes Modell, das aber trotzdem viele wichtige Eigenschaften eines echten Räuber-Beute-Systems zeigt.

Wir beginnen damit, dass wir zwei Tierarten betrachten, Hasen und Luchse, die sich beide exponentiell vermehren. Beide tun dies von unterschiedlichen Anfangswerten aus und mit unterschiedlichen **Wachstumsraten**. Die Simulationszeit soll 1 Jahr betragen, mit einer Genauigkeit von einzelnen Tagen.

```
In [1]: # Importieren von matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# Startbedingungen für Hasen
hasen = 1000          # wie viele Hasen sind am Anfang vorhanden
hasen_liste = []        # leere Liste
hasen_liste.append(hasen) # speichert den ersten Eintrag in der Liste
hasen_wachstum = 0.01   # Wachstumsrate, wie schnell vermehren sich die Hasen

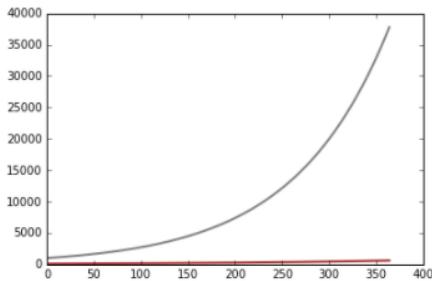
# Startbedingungen für Luchse
luchse = 100           # wie viele Luchse sind am Anfang vorhanden
luchs_liste = []        # leere Liste
luchs_liste.append(luchse) # speichert den ersten Eintrag in der Liste
luchs_wachstum = 0.005  # Wachstumsrate, wie schnell vermehren sich die Luchse

for it in range(365): # Schleife über 365 Tage
    # Populationsgleichungen:
    # population = population + wachstum
    # population = population + wachstumsrate * population
    hasen = hasen + hasen_wachstum * hasen
    luchse = luchse + luchs_wachstum * luchse

    # je aktuelle Population wird an die Listen angefügt
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Listen werden geplottet (lw gibt die Stärke der Linien an)
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)
```

Out[1]: [`<matplotlib.lines.Line2D at 0x980f860>`]



Das exponentielle Wachstum dieser Populationen ist eher unrealistisch. Es berücksichtigt nicht, dass Tiere auch sterben. Wir bauen deshalb, zusätzlich zur Wachstumsrate, auch eine **Sterberate** in unsere Gleichungen ein:

$$\text{Population} = \text{Population} + \text{Wachstumsrate} * \text{Population} - \text{Sterberate} * \text{Population}$$

```
In [2]: #Importieren von matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# Startbedingungen für Hasen
hasen = 1000          # wie viele Hasen sind am Anfang vorhanden
hasen_liste = []        # leere Liste
hasen_liste.append(hasen) #speichert den ersten Eintrag in der Liste
hasen_wachstum = 0.01   # wie schnell vermehren sich die Hasen
hasen_sterben = 0.001  # Sterberate, wie viele Hasen sterben

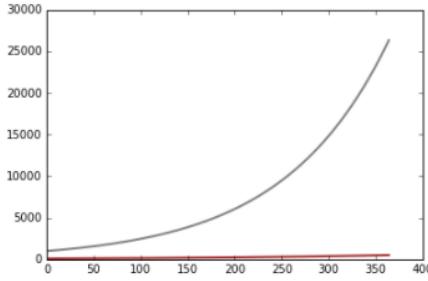
# Startbedingungen für Luchse
luchse = 100           # wie viele Luchse sind am Anfang vorhanden
luchs_liste = []        # leere Liste
luchs_liste.append(luchse) #speichert den ersten Eintrag in der Liste
luchs_wachstum = 0.005  # wie schnell vermehren sich die Luchse
luchs_sterben = 0.0005 # Sterberate, wie viele Luchse sterben

for it in range(365): #Schleife über 365 Tage
    # Populationsgleichungen:
    # population = population + wachstum - sterben
    # population = population + wachstumsrate * population - sterberate * population
    hasen = hasen + hasen_wachstum * hasen - hasen_sterben * hasen
    luchse = luchse + luchs_wachstum * luchse - luchs_sterben * luchse

    # je aktuelle Population wird an die Listen angefügt
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

#Listen werden geplottet
plt.plot(hasen_liste,color = "gray", lw = 2)
plt.plot(luchs_liste,color = "brown", lw = 2)
```

Out[2]: [<matplotlib.lines.Line2D at 0xb019e10>]



Die sich ergebenden Kurven sehen zwar ähnlich aus, wie zuvor, wachsen aber nicht so hoch an, wie der Blick auf die Werte an der y-Achse zeigt.

In dieser Variante unserer Gleichungen leben Hasen und Luchse noch vollkommen unabhängig voneinander. Wir könnten nun aber berücksichtigen, dass sich Luchse von Hasen ernähren und sich damit schneller vermehren, wenn es gerade viele Hasen gibt. Dazu benötigen wir eine neue Programmierstruktur, die:

If- Abfrage

If-Abfragen sind "Wenn-Dann-Fragen", mit denen wir Befehle an Bedingungen knüpfen können. Sie haben stets folgenden Aufbau:

```
if BEDIKUNG:
    BEFEHL
```

BEFEHL meint hierbei irgendeine Anweisung an Python, also zum Beispiel das Ändern einer Variable, einen Printbefehl oder etwas Ähnliches. Eine If-Abfrage kann auch mehrere Befehle enthalten.

Eine BEDIKUNG kann alles sein, was entweder wahr oder falsch sein kann. Oftmal benutzt man zur Definition einer Bedingung eine Gleichung. Man überprüft also etwa, ob es wahr ist, dass eine Variable x gleich groß ist wie eine Variable y. Noch häufiger werden Ungleichungen benutzt. Man überprüft also, ob es wahr ist, dass eine Variable x größer (oder kleiner) als eine Variable y ist.

Im folgenden Beispiel betrachten wir eine If-Abfrage in einer For-Schleife. Die For-Schleife erhöht die Zahl der Hasen fortlaufend um 1, die If-Abfrage führt einen Printbefehl aus, allerdings nur dann, wenn die Zahl der Hasen größer ist als 100.

```
In [3]: hasen = 0
for it in range(105):
    hasen = hasen + 1
    if hasen > 100:
        print("Es gibt jetzt mehr als 100 Hasen!")
```

Es gibt jetzt mehr als 100 Hasen!
Es gibt jetzt mehr als 100 Hasen!

Alternativ kann man die If-Abfrage auch benutzen, um festzustellen, wann es exakt 100 Hasen gibt. Dafür benutzt man ein doppeltes =-Zeichen, also ==.

Merk: In Programmier-Sprachen wird das Zeichen == zumeist in der Bedeutung "Ist-gleich" verwendet, während das Zeichen = für eine Zuweisung, zum Beispiel einer Zahl zu einer Variable, steht.

```
In [4]: hasen = 0 # Variablen-Definition, d.h. Zuweisung des Wertes 0 an die Variable hasen
```

```
for it in range(105):
    hasen = hasen + 1
    if hasen == 100: # Vergleich des Wertes der Variable hasen mit dem Wert 100
        print("Es gibt jetzt EXAKT 100 Hasen!")
```

Es gibt jetzt EXAKT 100 Hasen!

Dieses Beispiel demonstriert den wichtigen Unterschied zwischen dem einfachen = Zeichen und dem doppelten = Zeichen noch einmal: Das einfache = wird benutzt um einer Variable (hasen) einen Wert zu zuweisen. Das doppelte = Zeichen wird dagegen für einen Vergleich verwendet.

Es können auch mehrere Befehle unter einer If-Abfrage gestellt werden. Ähnlich wie bei der For-Schleife wird hier durch Einrücken signalisiert, ob ein Befehl Teil der If-Abfrage ist, oder nicht:

```
In [5]: hasen = 95
for it in range(10):
    hasen = hasen + 1
    if hasen > 100:
        print("Es gibt jetzt mehr als 100 Hasen!")
        print("Die exakte Zahl der Hasen ist ")
        print(hasen)

Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
101
Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
102
Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
103
Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
104
Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
105
Wenn wir die beiden zuletzt stehenden Befehle nicht einrücken, dann sind sie nicht mehr Teil der If-Abfrage und werden in jedem Fall ausgeführt, egal wie viele Hasen es gibt:
```

```
In [6]: hasen = 95
for it in range(10):
    hasen = hasen + 1
    if hasen > 100:
        print("Es gibt jetzt mehr als 100 Hasen!")
    print("Die exakte Zahl der Hasen ist ")
    print(hasen)

Die exakte Zahl der Hasen ist
96
Die exakte Zahl der Hasen ist
97
Die exakte Zahl der Hasen ist
98
Die exakte Zahl der Hasen ist
99
Die exakte Zahl der Hasen ist
100
Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
101
Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
102
Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
103
Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
104
Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
105
Wenn wir die beiden Befehle noch einmal nach außen verschieben, stehen sie auch außerhalb der For-Schleife und werden somit nur einmal ausgeführt, und zwar nachdem die For-Schleife abgeschlossen ist:
```

```
In [7]: hasen = 95
for it in range(10):
    hasen = hasen + 1
    if hasen > 100:
        print("Es gibt jetzt mehr als 100 Hasen!")
print("Die exakte Zahl der Hasen ist ")
print(hasen)

Es gibt jetzt mehr als 100 Hasen!
Die exakte Zahl der Hasen ist
105
```

Wir können nun unsere neuen Erkenntnisse über If-Abfragen in unsere Gleichung einbauen. Wir wollen dabei berücksichtigen, dass sich die Luchse besser vermehren können, wenn es sehr viele Hasen gibt. Dazu nehmen wir an, dass sich die Wachstumsrate der Luchse auf 0.01 erhöht, wenn es 1500 oder mehr Hasen gibt.

```
In [8]: # Importieren von matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# Startbedingungen für Hasen
hasen = 1000          # wie viele Hasen sind am Anfang vorhanden
hasen_liste = []        # leere Liste
hasen_liste.append(hasen) # speichert den ersten Eintrag in der Liste
hasen_wachstum = 0.01   # wie schnell vermehren sich die Hasen
hasen_sterben = 0.001  # wie viele Hasen sterben

# Startbedingungen für Luchse
luchse = 100           # wie viele Luchse sind am Anfang vorhanden
luchs_liste = []        # leere Liste
luchs_liste.append(luchse) # speichert den ersten Eintrag in der Liste
luchs_wachstum = 0.005  # wie schnell vermehren sich die Luchse
luchs_sterben = 0.0005 # wie viele Luchse sterben

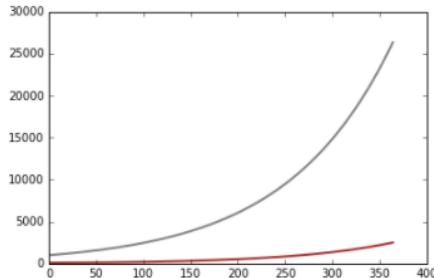
for it in range(365): #Schleife über 365 Tage
    # Populationsgleichungen:
    # population = population + wachstum - sterben
    # population = population + wachstumrate * population - sterberate * population
    hasen = hasen + hasen_wachstum * hasen - hasen_sterben * hasen
    luchse = luchse + luchs_wachstum * luchse - luchs_sterben * luchse

    # viele Hasen
    if hasen >= 1500:      # Wenn es 1500 oder mehr Hasen gibt
        luchs_wachstum = 0.01 # können sich die Luchse besser vermehren

    # je aktuelle Population wird an die Listen angefügt
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Listen werden geplottet
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)
```

Out[8]: [`<matplotlib.lines.Line2D at 0x3b6af0>`]



Mit dieser Erweiterung wächst die Zahl der Luchse ab einer gewissen Hasen-Dichte stärker an. Was wird unter diesen Bedingungen nun mit den Hasen passieren? Die Hasen werden stärker bejagt, ihre Sterberate wird sich also erhöhen. In den Systemwissenschaften nennt man einen solchen Effekt **Feedback**.

Feedbacks

Ein Feedback liegt immer dann vor, wenn die Änderung einer Größe direkt oder indirekt zu einer weiteren Änderung der Größe führt. Unser Beispiel impliziert gleich mehrere Feedbacks: Einerseits kann man die Vermehrung der Hasen und Luchse für sich bereits als Feedback betrachten: je mehr Hasen (Luchse) es gibt, desto mehr Hasen (Luchse) werden geboren. Diesbezüglich spricht man von einem **positiven Feedback**.

Es gibt allerdings auch **negative Feedbacks**. Diese kommen zu tragen, wenn eine positive Änderung einer Variable im Weiteren zu einer negativen Änderung der gleichen Variable führt. Auch ein solches negatives Feedback ist in unserem Beispiel vertreten: Wenn es viele Hasen gibt, gibt es auch viele Luchse. Viele Luchse führen aber dazu, dass mehr Hasen gefressen werden. Indirekt führt die positive Änderung der Hasenpopulation zu einer negativen Änderung der Hasenpopulation.

Bauen wir diesen Effekt nun (wieder mit einer If-Abfrage) in unser Modell ein. Wir nehmen dazu an, dass sich die Sterberate der Hasen auf 0.02 erhöht, wenn die Luchs-Population auf den Wert 200 (und darüber) anwächst.

```
In [9]: #Importieren von matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# Startbedingungen für Hasen
hasen = 1000          # wie viele Hasen sind am Anfang vorhanden
hasen_liste = []        # leere Liste
hasen_liste.append(hasen) #speichert den ersten Eintrag in der Liste
hasen_wachstum = 0.01   # wie schnell vermehren sich die Hasen
hasen_sterben = 0.001   # wie viele Hasen sterben

# Startbedingungen für Luchse
luchse = 100           # wie viele Luchse sind am Anfang vorhanden
luchs_liste = []        # leere Liste
luchs_liste.append(luchse) #speichert den ersten Eintrag in der Liste
luchs_wachstum = 0.005  # wie schnell vermehren sich die Luchse
luchs_sterben = 0.0005  # wie viele Luchse sterben

for it in range(365): #Schleife über 365 Tage
    # Populationsgleichungen:
    # population = population + wachstum - sterben
    # population = population + wachstumrate * population - sterberate * population
    hasen = hasen + hasen_wachstum * hasen - hasen_sterben * hasen
    luchse = luchse + luchs_wachstum * luchse - luchs_sterben * luchse

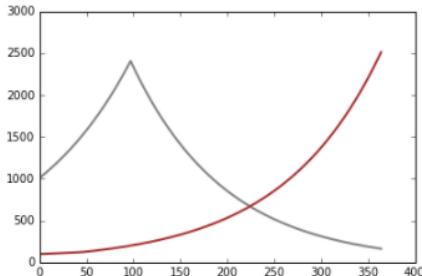
    #viele Hasen
    if hasen >= 1500:      # WENN es 1500 oder mehr Hasen gibt
        luchs_wachstum = 0.01  # können sich die Luchse besser vermehren

    #viele Luchse
    if luchse >= 200:       # WENN es 200 oder mehr Luchse gibt
        hasen_sterben = 0.02  # werden mehr Hasen gefressen

    # die je aktuellen Populationen werden an die Listen angefügt
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Listen werden geplottet
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)
```

Out[9]: [<matplotlib.lines.Line2D at 0x3b90390>]



Hier passiert nun offensichtlich etwas Spannendes, das wir aus unseren bisherigen Wachstums-Betrachtungen so noch nicht kennen: Die Hasen-Population beginnt plötzlich wieder abzunehmen. Wenn es sehr viele Luchse gibt, werden mehr Hasen gefressen als geboren werden. Die Population schrumpft. Die Luchse werden jedoch immer mehr. Am Ende der Simulation gibt es sogar mehr Luchse als Hasen. Was ist hier passiert?

Zuerst steigt die Anzahl der Hasen. Sobald eine kritische Größe (1500) überschritten wird, beginnen sich die Luchse stärker zu vermehren und die Sterberate der Hasen erhöht sich ebenfalls, wenn es 200 und mehr Luchse gibt. So weit, so gut. Das haben wir in unseren Überlegungen bedacht.

Die erhöhte Vermehrungsrate der Luchse bleibt allerdings nun bestehen, auch wenn es, später in der Simulation, wieder weniger Hasen gibt. Wir sollten also genau genommen eine weitere If-Abfrage einbauen, die überprüft, ob es irgendwann wieder weniger als 1500 Hasen gibt und damit die Vermehrungsrate der Luchse wieder auf den normalen Wert zurückfallen sollte. Es gibt für diesen Fall, in dem wir `hasen >= 1500` und `hasen < 1500` unterscheiden, eine elegante Methode, die so genannte:

If-else-Abfrage

Die If-else-Abfrage ist eine Erweiterung der If-Abfrage. Sie erweitert diese Struktur um zusätzliche Befehle, die ausgeführt werden, wenn die Bedingung 'nicht wahr' (also falsch) ist. Der Aufbau sieht so aus:

```
if BEDINGUNG:
    BEFEHL WENN WAHR
else:
    BEFEHL WENN FALSCH
```

Diese Struktur können wir nun verwenden, um die Wachstums- und Sterberaten wieder auf ihre ursprünglichen Werte zurückzusetzen, wenn die Hasen- und Luchs-Populationen wieder unter ihre kritischen Werte fallen.

```
In [10]: # Importieren von matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# Startbedingungen für Hasen
hasen = 1000          # wie viele Hasen sind am Anfang vorhanden
hasen_liste = []        # leere Liste
hasen_liste.append(hasen) # speichert den ersten Eintrag in der Liste
hasen_wachstum = 0.01   # wie schnell vermehren sich die Hasen
hasen_sterben = 0.001   # wie viele Hasen sterben

# Startbedingungen für Luchse
luchse = 100           # wie viele Luchse sind am Anfang vorhanden
luchs_liste = []        # leere Liste
luchs_liste.append(luchse) # speichert den ersten Eintrag in der Liste
luchs_wachstum = 0.005  # wie schnell vermehren sich die Luchse
luchs_sterben = 0.0005 # wie viele Luchse sterben

for it in range(365): # Schleife über 365 Tage
    # Populationsgleichungen:
    # population = population + wachstum - sterben
    # population = population + wachstumrate * population - sterberate * population
    hasen = hasen + hasen_wachstum * hasen - hasen_sterben * hasen
    luchse = luchse + luchs_wachstum * luchse - luchs_sterben * luchse

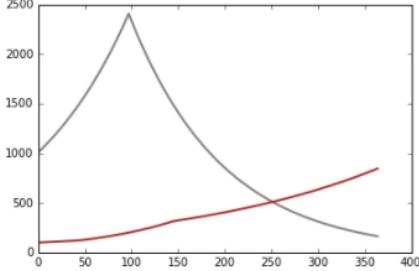
    # viele Hasen
    if hasen >= 1500:      # WENN es 1500 oder mehr Hasen gibt
        luchs_wachstum = 0.01 # können sich die Luchse besser vermehren
    else:                  # SONST
        luchs_wachstum = 0.005 # vermehren sie sich normal

    # viele Luchse
    if luchse >= 200:      # WENN es 200 oder mehr Luchse gibt
        hasen_sterben = 0.02 # werden mehr Hasen gefressen
    else:                  # SONST
        hasen_sterben = 0.001 # ist das Hasensterben normal groß

    # die je aktuellen Populationen werden an die Listen angefügt
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Listen werden geplottet
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)
```

Out[10]: [`<matplotlib.lines.Line2D at 0xb5e9780>`]



Mit dieser Änderung normalisiert sich das Wachstum der Luchse wieder, wenn die Hasen-Population zu schrumpfen beginnt.

Ein wichtiger Effekt fehlt uns aber noch in unserem Modell: Wenn es wieder weniger Hasen gibt, dann werden auch die Luchse wieder weniger Beute machen. Einige von ihnen werden nicht genug Nahrung finden. Ihre Sterberate erhöht sich. Auch diesen Effekt können wir mit einer If-Else-Bedingung in unser Modell einbauen:

Wir nehmen dazu an, dass sich die Sterberate der Luchse auf 0.01 erhöht, wenn es 500 oder weniger Hasen gibt.

```
In [11]: # Importieren von matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# Startbedingungen für Hasen
hasen = 1000          # wie viele Hasen sind am Anfang vorhanden
hasen_liste = []        # leere Liste
hasen_liste.append(hasen) # speichert den ersten Eintrag in der Liste
hasen_wachstum = 0.01   # wie schnell vermehren sich die Hasen
hasen_sterben = 0.001   # wie viele Hasen sterben

# Startbedingungen für Luchse
luchse = 100           # wie viele Luchse sind am Anfang vorhanden
luchs_liste = []        # leere Liste
luchs_liste.append(luchse) # speichert den ersten Eintrag in der Liste
luchs_wachstum = 0.005  # wie schnell vermehren sich die Luchse
luchs_sterben = 0.0005 # wie viele Luchse sterben

for it in range(365): # Schleife über 365 Tage
    # Populationsgleichungen:
    # population = population + wachstum - sterben
    # population = population + wachstumrate * population - sterberate * population
    hasen = hasen + hasen_wachstum * hasen - hasen_sterben * hasen
    luchse = luchse + luchs_wachstum * luchse - luchs_sterben * luchse
```

```

# viele Hasen
if hasen >= 1500:          # WENN es 1500 oder mehr Hasen gibt
    luchs_wachstum = 0.01  # können sich die Luchse besser vermehren
else:                      # SONST
    luchs_wachstum = 0.005 # vermehren sie sich normal

# viele Luchse
if luchse >= 200:          # WENN es 200 oder mehr Luchse gibt
    hasen_sterben = 0.02  # werden mehr Hasen gefressen
else:                      # SONST
    hasen_sterben = 0.001 # ist das Hasensterben normal groß

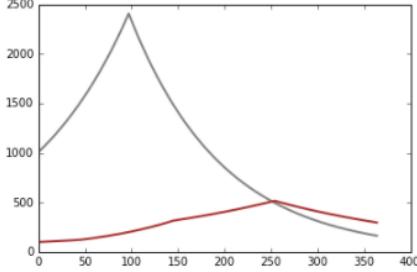
# wenig Hasen
if hasen <= 500:           # WENN es 500 oder weniger Hasen gibt
    luchs_sterben = 0.01  # verhungern viele Luchse
else:                      # SONST
    luchs_sterben = 0.0005 # ist das Luchssterben normal groß

# die je aktuellen Populationen werden an die Listen angefügt
hasen_liste.append(hasen)
luchs_liste.append(luchse)

# Listen werden geplottet
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)

```

Out[11]: [`<matplotlib.lines.Line2D at 0xadbd56a0>`]



Das Verhalten dieser beiden voneinander abhängigen Populationen ist nun schon einigermaßen komplex: Zuerst steigt die Zahl der Hasen, bis ihre Zahl nach etwa 100 Tagen ein Maximum erreicht und wieder abzunehmen beginnt. In Folge dessen wird nach etwa 250 Tagen das Nahrungsangebot für die Luchse knapp. Auch sie erreichen ihr Maximum. Beide Populationen weisen nun ein negatives Wachstum auf (d.h. sie schrumpfen).

Es stellt sich an dieser Stelle die Frage, ob Hasen und Luchse somit aussterben werden. Um diese Frage zu beantworten, erhöhen wir in unserem Modell einfach die Simulationszeit von einem auf sieben Jahre und beobachten, wie sich das System weiterentwickelt.

```

In [12]: # Importieren von matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# Startbedingungen für Hasen
hasen = 1000          # wie viele Hasen sind am Anfang vorhanden
hasen_liste = []        # leere Liste
hasen_liste.append(hasen) # speichert den ersten Eintrag in der Liste
hasen_wachstum = 0.01   # wie schnell vermehren sich die Hasen
hasen_sterben = 0.001   # wie viele Hasen sterben

# Startbedingungen für Luchse
luchse = 100           # wie viele Luchse sind am Anfang vorhanden
luchs_liste = []        # leere Liste
luchs_liste.append(luchse) # speichert den ersten Eintrag in der Liste
luchs_wachstum = 0.005  # wie schnell vermehren sich die Luchse
luchs_sterben = 0.0005  # wie viele Luchse sterben

for it in range(7*365): #Schleife über 7 * 365 Tage
    # Populationsgleichungen:
    # population = population + wachstum - sterben
    # population = population + wachstumrate * population - sterberate * population
    hasen = hasen + hasen_wachstum * hasen - hasen_sterben * hasen
    luchse = luchse + luchs_wachstum * luchse - luchs_sterben * luchse

    # viele Hasen
    if hasen >= 1500:          # WENN es 1500 oder mehr Hasen gibt
        luchs_wachstum = 0.01  # können sich die Luchse besser vermehren
    else:                      # SONST
        luchs_wachstum = 0.005 # vermehren sie sich normal

    # viele Luchse
    if luchse >= 200:          # WENN es 200 oder mehr Luchse gibt
        hasen_sterben = 0.02  # werden mehr Hasen gefressen
    else:                      # SONST
        hasen_sterben = 0.001 # ist das Hasensterben normal groß

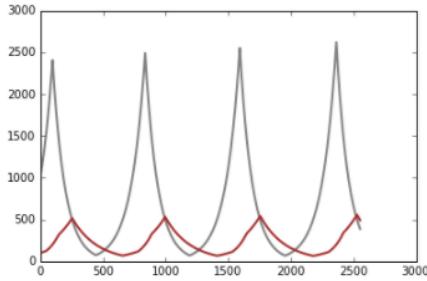
    # wenig Hasen
    if hasen <= 500:           # WENN es 500 oder weniger Hasen gibt
        luchs_sterben = 0.01  # verhungern viele Luchse
    else:                      # SONST
        luchs_sterben = 0.0005 # ist das Luchssterben normal groß

    # die je aktuellen Populationen werden an die Listen angefügt
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Listen werden geplottet
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)

```

```
Out[12]: [<matplotlib.lines.Line2D at 0xb042780>]
```



Wie wir sehen können, überleben in unserem Modell beide Arten, und das auch auf lange Sicht. Immer wenn die Hasen-Population einen niedrigen Wert erreicht, beginnt auch die Population der Luchse wieder zu schrumpfen und die Hasenpopulation kann sich wieder erholen. Beide Populationen pendeln zwischen einem Minimum und einem Maximum hin und her. Wir werden sehen, dass dies ein Verhalten ist, das für viele einfache Räuber-Beute Systeme charakteristisch ist.

Einen weiteren Effekt möchten wir noch einbauen: Wenn es sehr wenige Luchse gibt, fühlen sich die Hasen weniger bedroht und vermehren sich stärker. Wieder benutzen wir eine If-Else-Abfrage:

Wir nehmen dazu an, dass die Vermehrungs- oder Wachstumsrate der Hasen auf 0.011 steigt, wenn es weniger als 100 Luchse gibt.

```
In [13]:
```

```
import matplotlib.pyplot as plt
%matplotlib inline

# Startbedingungen für Hasen
hasen = 1000 # wie viele Hasen sind am Anfang vorhanden
hasen_liste = [] # leere Liste
hasen_liste.append(hasen) # speichert den ersten Eintrag in der Liste
hasen_wachstum = 0.01 # wie schnell vermehren sich die Hasen
hasen_sterben = 0.001 # wie viele Hasen sterben

# Startbedingungen für Luchse
luchs = 100 # wie viele Luchse sind am Anfang vorhanden
luchs_liste = [] # leere Liste
luchs_liste.append(luchs) # speichert den ersten Eintrag in der Liste
luchs_wachstum = 0.005 # wie schnell vermehren sich die Luchse
luchs_sterben = 0.0005 # wie viele Luchse sterben

for it in range(7*365): # Schleife über 7 * 365 Tage
    # Populationsgleichungen:
    # population = population + wachstum - sterben
    # population = population + wachstumrate * population - sterberate * population
    hasen = hasen + hasen_wachstum * hasen - hasen_sterben * hasen
    luchs = luchs + luchs_wachstum * luchs - luchs_sterben * luchs

    # viele Hasen
    if hasen >= 1500: # WENN es 1500 oder mehr Hasen gibt
        luchs_wachstum = 0.01 # können sich die Luchse besser vermehren
    else: # SONST
        luchs_wachstum = 0.005 # vermehren sie sich normal

    # viele Luchse
    if luchs >= 200: # WENN es 200 oder mehr Luchse gibt
        hasen_sterben = 0.02 # werden mehr Hasen gefressen
    else: # SONST
        hasen_sterben = 0.001 # ist das Hasensterben normal groß

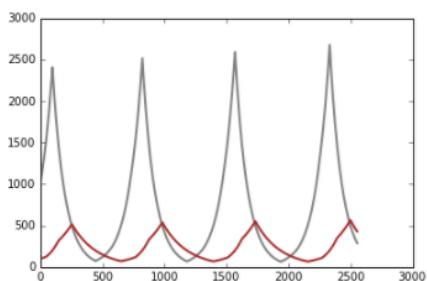
    # wenig Hasen
    if hasen <= 500: # WENN es 500 oder weniger Hasen gibt
        luchs_sterben = 0.01 # verhungern viele Luchse
    else: # SONST
        luchs_sterben = 0.0005 # ist das Luchssterben normal groß

    # wenig Luchse
    if luchs <= 100: # WENN es 100 oder weniger Luchse gibt
        hasen_wachstum = 0.011 # können sich die Hasen schneller vermehren
    else: # SONST
        hasen_wachstum = 0.01 # vermehren sie sich normal

    # die je aktuellen Populationen werden an die Listen angefügt
    hasen_liste.append(hasen)
    luchs_liste.append(luchs)

# Listen werden geplottet
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)
```

```
Out[13]: [<matplotlib.lines.Line2D at 0xb042f28>]
```



Dieses System scheint relativ stabil zu sein: Die Populationen oszillieren in ihrer Größe, aber keine Art ist vom Aussterben bedroht. Wir könnten nun aber auch Einflüsse von außen betrachten. Was würde zum Beispiel passieren, wenn die Hasen über einen bestimmten Zeitraum hinweg von Menschen bejagt werden.

Nehmen wir an, dass für einen kurzen Zeitraum, Jagd auf Hasen gemacht wird, und in diesem Zeitraum damit die Hasenpopulation zusätzlich sinkt. Wir nehmen diesen Zeitraum von Tag 1500 bis Tag 1580 an. Wir müssen also in unser Modell eine Abfrage einbauen, die überprüft, ob der je aktuelle Tag mindestens der 1500ste ist, aber gleichzeitig auch höchstens der 1580ste.

Dafür können wir den Umstand ausnutzen, dass man If-Abfragen **verschachteln** kann, dass man also If-Abfragen auch innerhalb von If-Abfragen stellen kann. Schematisch könnte dies zum Beispiel so aussehen:

```
if it >= 1500:
    if it <= 1580:
        HASENSAISON!
```

Für den Fall, dass zwei Bedingungen in dieser Weise gleichzeitig vorliegen, gibt es allerdings auch eine elegantere Methode. Wir können die beiden Bedingungen mit `and` verknüpfen, und so festlegen, dass beide Bedingungen gleichzeitig erfüllt sein müssen.

Darüber hinaus lassen sich Bedingungen auch mit `or` verknüpfen, und so sicherstellen, dass zumindest eine der beiden Bedingungen erfüllt, also wahr, sein muss.

Benutzen wir nun diese elegantere Methode, um die Hasenjagd in unsere Simulation einzubauen.

```
In [14]: # Importieren von matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# Startbedingungen für Hasen
hasen = 1000          # wie viele Hasen sind am Anfang vorhanden
hasen_liste = []        # Leere Liste
hasen_liste.append(hasen) # speichert den ersten Eintrag in der Liste
hasen_wachstum = 0.01   # wie schnell vermehren sich die Hasen
hasen_sterben = 0.001  # wie viele Hasen sterben

# Startbedingungen für Luchse
luchse = 100           # wie viele Luchse sind am Anfang vorhanden
luchs_liste = []        # Leere Liste
luchs_liste.append(luchse) # speichert den ersten Eintrag in der Liste
luchs_wachstum = 0.005 # wie schnell vermehren sich die Luchse
luchs_sterben = 0.0005 # wie viele Luchse sterben

for it in range(7*365): #Schleife über 7 * 365 Tage
    # Populationsgleichungen:
    # population = population + wachstum - sterben
    # population = population + wachstumrate * population - sterberate * population
    hasen = hasen + hasen_wachstum * hasen - hasen_sterben * hasen
    luchse = luchse + luchs_wachstum * luchse - luchs_sterben * luchse

    # viele Hasen
    if hasen >= 1500:          # WENN es 1500 oder mehr Hasen gibt
        luchs_wachstum = 0.01  # können sich die Luchse besser vermehren
    else:                      # SONST
        luchs_wachstum = 0.005 # vermehren sie sich normal

    # viele Luchse
    if luchse >= 200:          # WENN es 200 oder mehr Luchse gibt
        hasen_sterben = 0.02   # werden mehr Hasen gefressen
    else:                      # SONST
        hasen_sterben = 0.001  # ist das Hasensterben normal groß

    # wenig Hasen
    if hasen <= 500:          # WENN es 500 oder weniger Hasen gibt
        luchs_sterben = 0.01   # verhungern viele Luchse
    else:                      # SONST
        luchs_sterben = 0.0005 # ist das Luchssterben normal groß

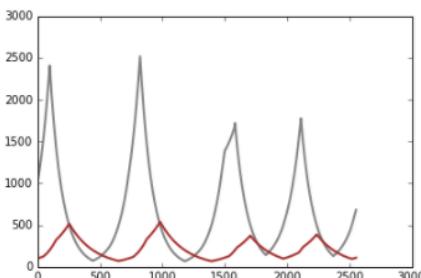
    # wenig Luchse
    if luchse <= 100:          # WENN es 100 oder weniger Luchse gibt
        hasen_wachstum = 0.011  # können sich die Hasen schneller vermehren
    else:                      # SONST
        hasen_wachstum = 0.01   # vermehren sie sich normal

    #Hasensaison
    #WENN der Tag 'it' 1500 oder größer ist UND GLEICHZEITIG 1580 oder kleiner
    if it >= 1500 and it <= 1580:
        hasen = hasen - 10

    # die je aktuellen Populationen werden an die Listen angefügt
    hasen_liste.append(hasen)
    luchs_liste.append(luchse)

# Listen werden geplottet
plt.plot(hasen_liste, color = "gray", lw = 2)
plt.plot(luchs_liste, color = "brown", lw = 2)
```

Out[14]: [`<matplotlib.lines.Line2D at 0xb0c04278>`]



Der Effekt, den eine solche Jagd auf das Ökosystem hat, ist nun gut zu sehen. Die Population der Hasen und Luchse wird damit nicht nur kurzzeitig reduziert, sondern kann sich auf lange Sicht nicht mehr erholen. Der ursprüngliche Bestand wird auch nach Ende der Jagd, also nach dem 1580sten Tag, nicht mehr erreicht.

In Kapitel 9 werden wir auf Räuber-Beute-Systeme zurückkommen und mathematische Beschreibungen solcher Systeme mittels gekoppelter Differenzen- bzw. Differentialgleichungen kennen lernen.

Zusammenfassung

If-Abfragen

If-Abfragen können benutzt werden, um Anweisungen und Befehle nur unter gewissen Bedingungen ausführen zu lassen. Sie haben den Aufbau

```
if BEDINGUNG:  
    BEFEHL
```

Die Bedingungen sind meist Ungleichungen (hasen >= 100) oder Gleichungen (hasen == 100), die entweder **wahr** oder **falsch** sein können. Wichtig ist, bei einem Vergleich das doppelte =-Zeichen zu verwenden.

If-Else Abfragen

Die If-Else-Abfrage ist eine Erweiterung der If-Abfrage. Sie erweitert diese Struktur um zusätzliche Befehle, die ausgeführt werden sollen, wenn die Bedingung **nicht wahr** ist. Sie hat den Aufbau:

```
if BEDINGUNG:  
    BEFEHL WENN WAHR  
else:  
    BEFEHL WENN FALSCH
```

and / or

Mit and und or kann man mehrere Bedingungen miteinander verknüpfen, entweder so, dass alle Bedingungen erfüllt sein müssen, oder so, dass nur eine erfüllt werden muss.

- wahr and wahr = wahr
- wahr and falsch = falsch
- wahr or wahr = wahr
- wahr or falsch = wahr

Kapitel 4 – Zufallszahlen – Ein Energiemix

In den bisher betrachteten Beispielen systemwissenschaftlicher Modellbildung sind wir davon ausgegangen, dass wir die Daten, die wir dafür benötigen, kennen, dass wir also zum Beispiel wissen, wie groß die Wachstumsrate oder auch die Anfangsgröße einer Tierpopulation ist. Dies ist leider aber nicht immer der Fall. Oftmals haben wir bestenfalls eine mehr oder weniger gute Vorstellung über ein Intervall, innerhalb dessen sich die benötigten Daten bewegen. In diesen Fällen können wir **Zufallszahlen** heranziehen, um unser Modell an realistische Gegebenheiten anzunähern. Zum Glück stellt uns auch dafür Python die nötigen Werkzeuge zur Verfügung.

Um dies zu demonstrieren, wollen wir uns im Folgenden mit einem Energienetzwerk beschäftigen, bestehend aus Wind-, Wasser- und Solarkraftwerken. Wir werden die Produktion dieser Kraftwerke dem Verbrauch gegenüberstellen und Strategien finden, wie man den Bedarf möglichst gut decken kann, ohne Strom aus fossilen Brennstoffen ankaufen zu müssen.

Wir beginnen unser diesbezügliches Modell ganz einfach mit einer For-Schleife, die Stunde für Stunde über den Simulationszeitraum iteriert. In dieses Grundgerüst bauen wir auch schon ein Wasserkraftwerk ein, das konstant Strom liefert. Um den produzierten Strom grafisch darzustellen, verwenden wir einen so genannten **Stack-Plot**:

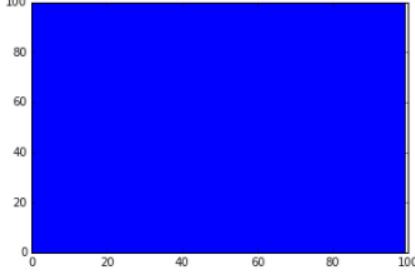
```
In [1]: import matplotlib.pyplot as plt
%matplotlib inline

simulationszeit = 100
wasserkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = 100
    wasserkraft_liste.append(wasserkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste)
```

Out[1]: [`<matplotlib.collections.PolyCollection at 0xad5f908>`]



Ein Wasserkraftwerk, das konstant die selbe Menge Strom liefert ist zwar parktisch, leider aber nicht realistisch. In der Realität wird es immer kleine Schwankungen geben, die wir auch in unsere Simulation einbauen sollten. Dazu benutzen wir **Zufallszahlen**.

Für unser Wasserkraftwerk nehmen wir an, dass die Leistung meistens etwa 100 MW beträgt. Kleine Abweichungen werden häufiger sein, große Abweichungen seltener. Wir haben es also mit einer so genannten **Normal-** oder auch **Gauß-Verteilung** zu tun. Gauß-verteilte Zufallszahlen generiert man in Python mit dem Befehl `random.gauss(mittelwert, standardabweichung)` aus dem Packet `random`, das wir zunächst importieren müssen. Als Mittelwert verwenden wir 1 (mal 100 MW), als Standardabweichung können wir 0.1 benutzen (also 10%). Vereinfacht gesagt heißt das, dass die Zahlen, die damit generiert werden, im Durchschnitt 10% vom Mittelwert entfernt liegen, und zwar, je größer die Abweichung ist, umso seltener.

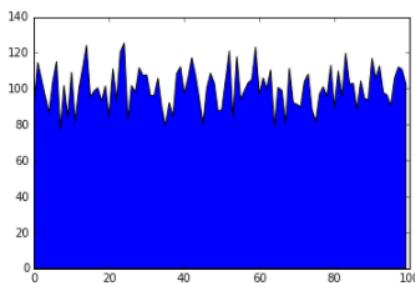
```
In [2]: import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
wasserkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste)
```

Out[2]: [`<matplotlib.collections.PolyCollection at 0xb0177f0>`]



Neben einem Wasserkraftwerk möchten wir nun auch ein Windkraftwerk in unser Stromnetz einbauen. Die Leistung von Windkraftwerken ist stärker vom Zufall abhängig: bei Windstille produzieren sie gar nichts, aber auch zu starker Wind macht ihnen Probleme. Ein Gauß-Verteilung ist hier also nicht sinnvoll, wenn wir abbilden wollen, dass ein Windkraftwerk hin und wieder auch keinen Strom liefert.

Wir können dazu aber **gleich-verteilte** Zufallszahlen heranziehen. Bei gleichverteilten Zufallszahlen ist jede Zahl gleich wahrscheinlich. Der folgende Plot illustriert den Unterschied zwischen Gauß-verteilten und gleich-verteilten Zufallszahlen. Wir verwenden den Befehl `random.uniform(0, 20)`, um reelle Zahlen zwischen 0 und (ausschließlich) 20 zu erzeugen. Diese werden zu 90 hinzu addiert, um eine ebenfalls um 100 variiierende Verteilung zu erzeugen. Zum Plotten verwenden wir hier den Befehl `hist`, der ein Histogramm der erzeugten Daten zeichnet.

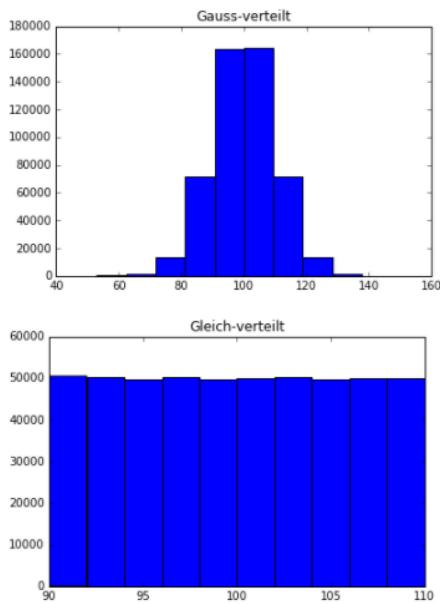
```
In [3]: gauss=[]
gleich=[]

for it in range(500000):
    gauss.append(random.gauss(100, 10))
    gleich.append(90 + random.uniform(0, 20))

plt.hist(gauss)
plt.title('Gauss-verteilt')

plt.figure()
plt.hist(gleich);
plt.title('Gleich-verteilt')
```

Out[3]: <matplotlib.text.Text at 0xeae37f0>



Unser Windkraftwerk soll eine maximale Leistung von 7 MW besitzen. Bei Windstille kann es aber vorkommen, dass auch gar kein Strom geliefert wird. Wir verwenden für die Leistung des Windkraftwerks also eine Zufallszahl zwischen 0 und 7, die wir mit dem Befehl `random.uniform()` erzeugen. Pythons `Stackplot` liefert uns die bequeme Möglichkeit, die zusätzlich zu den 100% Wasserkraft erzeugte Windenergie einfach grafisch auf die Wasserkraft auf zu addieren. Zur Unterscheidung der beiden Energiearten verwenden wir eine zusätzliche Farbe, ein helleres Blau, das mit dem Kürzel `#cceeef` erzeugt wird.

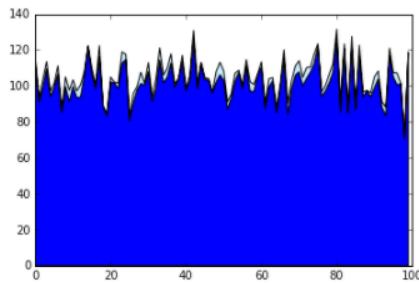
```
In [4]: import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = random.uniform(0, 7)
    windkraft_liste.append(windkraft)

# '#cceeef' steht für ein etwas helleres blau
plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, colors = ('blue', '#cceeef'))
```

```
Out[4]: [<matplotlib.collections.PolyCollection at 0xe843470>,
 <matplotlib.collections.PolyCollection at 0xe6c3080>]
```



Wir sehen, dass der Beitrag eines einzelnen Windkraftwerks, im Vergleich zu einem Wasserkraftwerk, nur eher wenig ausmacht. Wir sollten unser Modell also ausbauen, sodass wir die Zahl der jeweiligen Kraftwerke bestimmen können. Das wird es uns später erlauben zu bestimmen, wie viele Kraftwerke wir von welchem Typ benötigen.

Wir gehen hier im Folgenden zunächst von 10 Windkraftwerken und einem Wasserkraftwerk aus.

```
In [5]: import matplotlib.pyplot as plt
import random
%matplotlib inline

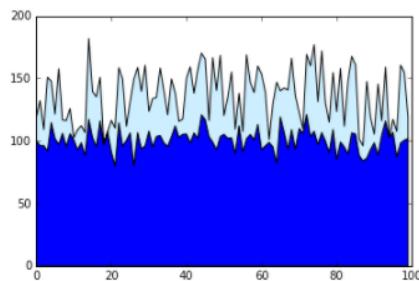
simulationszeit = 100
windkraft_anzahl = 10
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = windkraft_anzahl * random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, colors=('blue', '#cceeef'))
```

```
Out[5]: [<matplotlib.collections.PolyCollection at 0xe54eb0>,
 <matplotlib.collections.PolyCollection at 0xea00c88>]
```



Sollten wir nun als Vorgabe haben, dass zu jeder Zeit mindestens 150 MW Strom verfügbar sein sollen, so sehen wir, dass die Anzahl unserer Windkraftwerke noch nicht ausreicht:

```
In [6]: import matplotlib.pyplot as plt
import random
%matplotlib inline

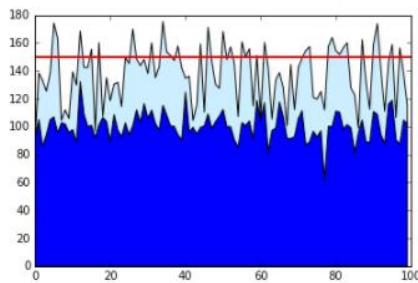
simulationszeit = 100
windkraft_anzahl = 10
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = windkraft_anzahl * random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, colors = ('blue', '#cceeef'))
# zeichne eine rote Linie in der Höhe von 150
plt.plot((0, 100),(150, 150), c = "red", lw = 2)
```

```
Out[6]: [<matplotlib.lines.Line2D at 0xe54e9e8>]
```



Wir müssen die Anzahl an Windkraftwerken also noch ein wenig erhöhen:

```
In [7]: import matplotlib.pyplot as plt
import random
%matplotlib inline

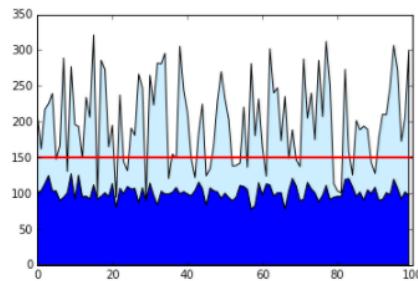
simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = windkraft_anzahl * random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, colors = ('blue', '#cceeff'))
plt.plot((0, 100), (150, 150), c = "red", lw = 2)
```

```
Out[7]: [<matplotlib.lines.Line2D at 0xe54e2b0>]
```



Wir sehen, dass, auch wenn wir 30 Windkraftwerke vorsehen und wir damit zu manchen Zeiten doppelt so viel Strom produzieren wie benötigt, es immer noch Zeiten gibt, zu denen wir die 150 MW Marke nicht erreichen. Wo liegt das Problem?

Das Problem liegt in der Eigenart der Windenergie. Bei Windstille produziert keines der 30 Windkraftwerke Strom. Wir haben beim Erstellen unseres Modells implizit angenommen, dass alle Windkraftwerke in der selben Region liegen und damit von den selben Windverhältnissen betroffen sind. In unserem Programmcode verwenden alle Kraftwerke die gleiche Zufallszahl: Wenn also in einer Stunde "Windstille" gewürfelt wird, dann gilt das für alle Kraftwerke.

Wir könnten aber auch annehmen, dass die Kraftwerke so weit voneinander entfernt stehen, dass es für jedes Kraftwerk andere Windverhältnisse gibt. Aber ändert das etwas? Wird dann im Mittel nicht genau gleich viel Strom produziert?

Probieren wir es aus: Ändern wir unser Programm, sodass für jedes Kraftwerk eine eigene Zufallszahl vorgibt, wieviel Strom gerade produziert wird.

```
In [8]: import matplotlib.pyplot as plt
import random
%matplotlib inline

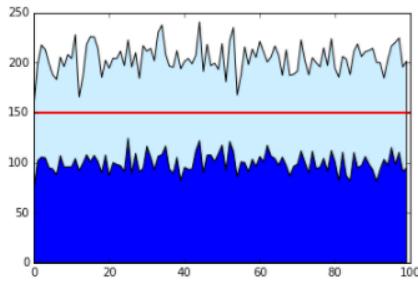
simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, colors = ('blue', '#cceeff'))
plt.plot((0, 100), (150, 150), c = "red", lw = 2)
```

```
Out[8]: [<matplotlib.lines.Line2D at 0xf0debe0>]
```



Wir sehen: obwohl im Mittel gleich viel Strom produziert wird, macht die Annahme, dass die Windkraftwerke von unterschiedlichen Windverhältnissen betroffen sind, einen großen Unterschied: wenn wir für jedes Kraftwerk eine eigene Zufallszahl erzeugen, mittelt sich das Ergebnis besser. Es gibt weniger Spitzen und weniger Senken, die Stromproduktion ist näher am Mittelwert.

Das ist eine grundlegende Eigenschaft von Zufallszahlen: Wenn man sehr viele davon verwendet, ist das Ergebnis immer sehr nahe am Mittelwert der möglichen Resultate.

Mit dem nun vorliegenden Modell schaut es also so aus, als würden wir sicherstellen können, dass immer 150 MW Strom verfügbar sind. Was passiert aber, wenn wir miteinbeziehen, dass Windkraftwerke einmal defekt werden können? Wie bauen wir so ein Zufallereignis in unsere Simulation ein?

Zufallereignisse mit Python

Wir haben bisher gelernt, wie man zufällige Zahlen erstellt. Was aber, wenn wir keine Zahl erzeugen wollen, sondern ein Ereignis - z.B. einen Defekt in einem Windkraftwerk -, das mit bestimmter Wahrscheinlichkeit auftritt? Wir können dazu Zufallszahlen mit `if`-Abfragen kombinieren.

Dazu benutzen wir den Umstand, dass 10% der Zahlen von 1 bis 100 kleiner oder gleich 10 sind. Und 50% der Zahlen sind kleiner gleich 50. Im Code könnten wir also schreiben:

```
In [9]: chance = 50
zufallszahl = random.uniform(0, 100)
if zufallszahl <= chance:
    print "KOPF!"
else:
    print "ZAHL!"
```

ZAHL!

Das heißt, wir vergleichen hier die prozentuelle Chance eines Zufallereignisses mit einer Zufallszahl und können somit sicher sein, dass dieses Ereignis auch mit genau dieser Chance passiert.

Bauen wir so eine Chance in unsere Simulation ein. Wir nehmen an, dass jede Stunde eine 10%-Chance besteht, dass ein Defekt im Windkraftwerk für einen Ausfall der Stromversorgung sorgt:

```
In [10]: import matplotlib.pyplot as plt
import random
%matplotlib inline

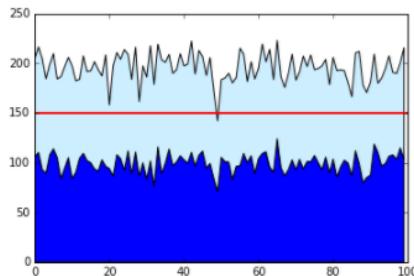
simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        if random.uniform(0, 100) <= 10:
            # STORUNG: kein Strom wird produziert
            windkraft = windkraft + 0
        else:
            # Normalbetrieb:
            windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, colors = ('blue', '#cceeff'))
plt.plot((0, 100), (150, 150), c = "red", lw = 2)
```

```
Out[10]: [<matplotlib.lines.Line2D at 0xf263748>]
```



Wir sehen, dass wir, obwohl es manchmal schon knapp wird, die Stromversorgung meistens noch sicherstellen können. Erhöhen wir aber die Ausfallchance zum Beispiel auf 50%, so sieht die Sache anders aus.

```
In [11]: import matplotlib.pyplot as plt
import random
%matplotlib inline

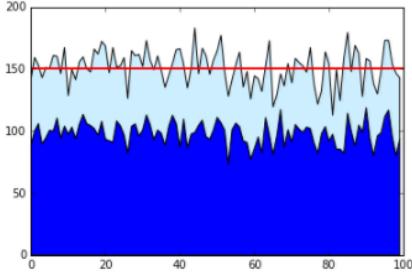
simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        if random.uniform(0, 100) <= 50:
            # STÖRUNG: kein Strom wird produziert
            windkraft = windkraft + 0
        else:
            # Normalbetrieb:
            windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)

plt.stackplot(range(simulationszeit), wasserkraft_liste, windkraft_liste, colors = ('blue', '#cceeef'))
plt.plot((0, 100), (150, 150), c = "red", lw = 2)

Out[11]: [<matplotlib.lines.Line2D at 0xb0ae9e8>]
```



Um noch etwas mehr über Zufallsgeneratoren zu lernen, wollen wir im Folgenden noch ein weiteres Kraftwerk in unser Stromnetz einbauen. Wir nehmen dazu ein eher unzuverlässiges Kohlekraftwerk an, das nur drei Betriebsmodi hat: "Stillstand" (0 MW), "Normalbetrieb" (20 MW) und "Hochbetrieb" (70 MW). In unserem Beispiel treten diese drei Betriebsmodi zufällig auf, kein anderer MW-Wert kann produziert werden.

Wie würden wir das am besten umsetzen? Eine Zufallszahl zwischen 0 und 70 wäre schon einmal falsch, 3 aufeinanderfolgende 33%-Chancen wären auch nicht richtig, da in dem Fall ja mehrere Betriebsmodi auf einmal möglich wären. In Python gibt es glücklicherweise eine ganz einfache Lösung für solche Probleme: der Befehl `random.choice((a, b, c))` trifft eine zufällige Entscheidung zwischen a, b und c, wobei a, b und c nicht einmal Zahlen sein müssen. Bauen wir damit das Kraftwerk in unsere Simulation ein.

```
In [12]: import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

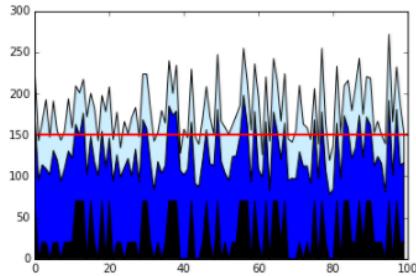
wasserkraft_liste = []
windkraft_liste = []
kohlekraft_liste = []

for zeit in range(simulationszeit):
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        if random.uniform(0, 100) <= 50:
            # STÖRUNG: kein Strom wird produziert
            windkraft = windkraft + 0
        else:
            # Normalbetrieb:
            windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)

    kohlekraft = random.choice((0, 20, 70))
    kohlekraft_liste.append(kohlekraft)

plt.stackplot(range(simulationszeit), kohlekraft_liste, wasserkraft_liste, windkraft_liste, colors = ('black', 'blue','#cceeef'))
plt.plot((0, 100), (150, 150), c = "red", lw = 2)
```

```
Out[12]: [<matplotlib.lines.Line2D at 0xf143d68>]
```



Freilich gibt es noch Probleme, die wir mit unserer bisherigen Kenntnis von Zufallszahlen noch nicht darstellen können: Nehmen wir etwa an, es gäbe innerhalb des Simulationszeitraumes eine Stunde, in der das Stromnetz eine besondere Last erfährt. Der Verbrauch steigt hierbei auf 200 MW. Wir wissen jedoch nicht, wann genau diese Stunde sein wird, wir wissen nur dass es eine solche Stunde gibt.

Wäre diese Stunde ganz am Anfang der Simulationszeit, wäre es ein einfaches Problem:

```
In [13]: import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []
kohlekraft_liste = []
verbrauch_liste = []

for zeit in range(simulationszeit):
    verbrauch_liste.append(150)
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        if random.uniform(0, 100) <= 50:
            # STÖRUNG: kein Strom wird produziert
            windkraft = windkraft + 0
        else:
            #Normalbetrieb:
            windkraft = windkraft + random.uniform(0, 7)
        windkraft_liste.append(windkraft)

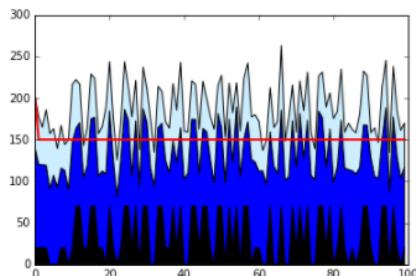
    kohlekraft= random.choice((0, 20, 70))
    kohlekraft_liste.append(kohlekraft)

plt.stackplot(range(simulationszeit), kohlekraft_liste, wasserkraft_liste, windkraft_liste, colors = ('black', 'blue', '#cceeff'))

#erstes Element von verbrauch_Liste auf 200 setzen:
verbrauch_liste[0] = 200

plt.plot(verbrauch_liste, c= "red", lw = 2)
```

```
Out[13]: [<matplotlib.lines.Line2D at 0xf76df0>]
```



Was aber, wenn diese überhöhte Energienachfrage einfach irgendwann auftritt? In diesem Fall hilft uns der Python-Befehl `random.shuffle(liste)`. Dieser mischt die Einträge einer Liste durcheinander. Damit kann man nicht nur Karten mischen...

```
In [14]: karten = ['Herz 2', 'Herz 3', 'Herz 4', 'Herz 5', 'Herz 6', 'Herz 7']
random.shuffle(karten)
print(karten)
```

```
['Herz 3', 'Herz 7', 'Herz 6', 'Herz 4', 'Herz 2', 'Herz 5']
```

... sondern auch unser oben genanntes Problem lösen:

```
In [15]: import matplotlib.pyplot as plt
import random
%matplotlib inline

simulationszeit = 100
windkraft_anzahl = 30
wasserkraft_anzahl = 1

wasserkraft_liste = []
windkraft_liste = []
kohlekraft_liste = []
verbrauch_liste = []

for zeit in range(simulationszeit):
    verbrauch_liste.append(150)
    wasserkraft = wasserkraft_anzahl * 100 * random.gauss(1, 0.1)
    wasserkraft_liste.append(wasserkraft)
    windkraft = 0
    for kraftwerke in range(windkraft_anzahl):
        if random.uniform(0, 100) <= 50:
            # STÖRUNG: kein Strom wird produziert
            windkraft = windkraft + 0
        else:
            # Normalbetrieb:
            windkraft = windkraft + random.uniform(0, 7)
    windkraft_liste.append(windkraft)

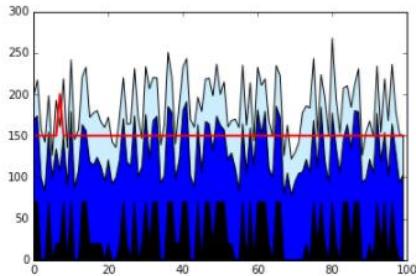
    kohlekraft = random.choice((0, 20, 70))
    kohlekraft_liste.append(kohlekraft)

plt.stackplot(range(simulationszeit), kohlekraft_liste, wasserkraft_liste, windkraft_liste, colors=('black','blue','#cceeef'))

# erstes Element der verbrauch_Liste auf 200 setzen:
verbrauch_liste[0]=200
# shuffeln der gesamten Liste
random.shuffle(verbrauch_liste)

plt.plot(verbrauch_liste, c = "red", lw = 2)
```

Out[15]: [`<matplotlib.lines.Line2D at 0xe578f98>`]



Zusammenfassung

Zufallszahlen

Zufallszahlen werden in Python mit dem Paket `random` generiert. Es gibt verschiedene Befehle, die unterschiedliche Verteilungen von Zufallszahlen erzeugen.

Gleich-verteilte (reelle) Zufallszahlen erstellt man mit `random.uniform(a, b)`, wobei `a` die kleinste mögliche Zahl ist und `b` die obere Grenze markiert, bis zu der (aber ausschließlich der) Zufallszahlen erzeugt werden. Alle Zahlen von `a` bis (exklusive) `b` treten gleich wahrscheinlich auf. Somit lassen sich auch prozentuelle Wahrscheinlichkeiten leicht implementieren: Da 10% aller Zahlen zwischen 0 und 100 kleiner als 10 sind, und in gleichverteilten Zufallszahlen alle Zahlen gleich wahrscheinlich sind, wird die `if`-Abfrage `if random.uniform(0, 100) < 10` in etwa 10 Prozent der Fälle `true` liefern.

Normal-verteilte Zufallszahlen erstellt man mit `random.gauss(m, s)`, wobei `m` den Mittelwert der erzeugten Verteilung angibt und `s` die Standardabweichung. Der genaue Wertebereich ist also schwer abzuschätzen, auch bei `random.gauss(100, 10)` ist es theoretisch möglich, einen Wert kleiner als 50 zu bekommen. Die Chance dafür ist aber verschwindend gering. Je näher die Zahl am Mittelwert ist, umso wahrscheinlicher tritt sie auf. Rund 70% der Werte werden im Intervall $m \pm s$ liegen, also in diesem Beispiel zwischen 90 und 110. Etwa 95% liegen im Intervall $m \pm 2s$ (also 80 bis 120) und über 99% im Intervall $m \pm 3s$ (also 70 bis 130).

Bei jedem Aufruf einer Zufallsfunktion wird eine neue Zahl generiert. Wenn man ein Programm mit Zufallszahlen mehrmals hintereinander ausführt, bekommt man im Allgemeinen unterschiedliche Ergebnisse.

Zufallsergebnisse Auch zufällige Ereignisse erzeugt man am besten mit Zufallszahlen: Da $x\%$ aller Zahlen von 1 bis 100 kleiner sind als `x`, können wir mit `if random.uniform(0, 100) <= chance` jede beliebige prozentuelle Chance simulieren.

Random Choice `random.choice` wählt ein zufälliges Element von mehreren Elementen aus. Jedes Element ist gleich wahrscheinlich und die Elemente müssen nicht zwangsläufig Zahlen sein.

Random Shuffle `random.shuffle` mischt eine Liste. Die Elemente bleiben gleich, die Position der Elemente ändert sich. Das ist hilfreich, wenn bekannt ist, welcher Wert der Liste wie oft auftritt, die genaue Position des Wertes aber zufällig sein soll.

Kapitel 5 – Vektoren und Matrizen – Die Bewirtschaftung eines Waldes

Eine sehr zentrale Methode für den Umgang mit Daten und Inhalten in den Systemwissenschaften stellt das Rechnen mit Vektoren und Matrizen dar. Python bietet auch hierfür die benötigten Werkzeuge. Um diese kennenzulernen, beschäftigen wir uns im folgenden Beispiel mit der nachhaltigen Bewirtschaftung eines Waldes. Uns interessiert insbesondere, wie der Wald (nach)wächst.

Beginnen wir mit einem ganz einfachen, eindimensionalem Wald-Modell: Wir betrachten also nur eine "Reihe" von Bäumen. In unserem Modell brauchen Bäume Platz um gut wachsen zu können, zwei Bäume dürfen also nie an benachbarten Stellen stehen. Dazu legen wir mit Python eine Liste an, in die wir eine Eins schreiben, wenn sich an einer entsprechenden Stelle ein Baum befindet, und eine Null, wenn nicht. Der Einfachheit halber wechseln wir zunächst einfach ab: Baum, kein Baum, Baum, kein Baum,....

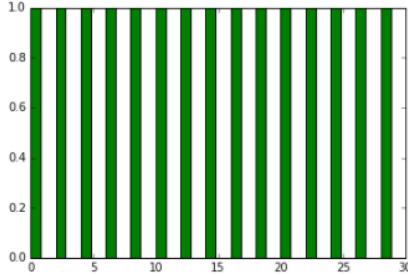
```
In [1]: import matplotlib.pyplot as plt
%matplotlib inline

waldid = []

for posx in range(15):
    waldid.append(1)
    waldid.append(0)

plt.bar(range(len(waldid)), waldid, color='green')
```

Out[1]: <Container object of 30 artists>



So sieht der Wald aber noch sehr "künstlich" aus. Wir könnten den Wald auch ein wenig zufälliger wachsen lassen: Wir könnten links beginnen und dann Feld für Feld mit einer gewissen Wahrscheinlichkeit einen Baum wachsen lassen:

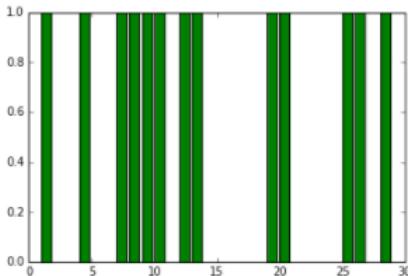
```
In [2]: import matplotlib.pyplot as plt
# ein Paket zum Erzeugen von und Rechnen mit Zufallszahlen
import random
%matplotlib inline

waldid = []

for posx in range(30):
    if random.uniform(0, 100) <= 50:
        waldid.append(1)
    else:
        waldid.append(0)

plt.bar(range(len(waldid)), waldid, color = 'green')
```

Out[2]: <Container object of 30 artists>



So sieht der Wald zwar zufällig aus, aber noch nicht richtig natürlich: Bäume wachsen nicht so gerne direkt nebeneinander, da sie sich dabei gegenseitig Schatten machen. Wie können wir diesen Umstand in unser Waldmodell einbauen? Die Wahrscheinlichkeit, dass an einer bestimmten Position ein Baum wachsen kann, hängt scheinbar davon ab, ob an der Position daneben schon ein Baum steht oder nicht. Für die Entscheidung für einen Baum an Position x , brauchen wir also Daten zu Position $x - 1$.

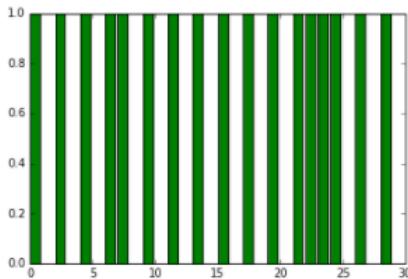
Dies ist kein Problem. Wir haben unseren ganzen Wald in einer Liste, also mathematisch gesehen in einem **Vektor** gespeichert. Wir erhalten diese Information also, indem wir einfach das je vorherige Vektorelement betrachten. Nehmen wir an, dass ein Baum nur noch eine 20%-Chance hat zu wachsen, wenn an der vorhergehenden Position schon ein Baum steht. Dagegen könnte die Chance auf 90% steigen, wenn das Nachbarfeld frei ist. Das könnten wir wie folgt in unseren Python-Code einbauen:

```
In [3]: import matplotlib.pyplot as plt
import random
%matplotlib inline

wald1d = [1]

for posx in range(1, 30):
    # Beachte: das Zeichen == steht in den meisten Programmiersprachen für eine tatsächliche Gleichheit,
    # während das Zeichen = für eine Zuweisung eines Werte zu einer Variable steht.
    if wald1d[posx - 1] == 1:
        if random.uniform(0,100) <= 20:
            wald1d.append(1)
        else:
            wald1d.append(0)
    else:
        if random.uniform(0,100) <= 90:
            wald1d.append(1)
        else:
            wald1d.append(0)
plt.bar(range(len(wald1d)), wald1d, color = 'green')

out[3]: <Container object of 30 artists>
```



Das wäre also unser ein-dimensionaler Wald. Wir wollen ihn nun auf zwei Dimensionen erweitern.

Bisher haben wir unseren "Wald" in einer Liste, bzw. mathematisch in einem **Vektor** gespeichert. In der Mathematik ist ein Vektor ein Schema aus untereinanderstehenden Zahlen. Ein ganzer Wald besteht aber in der Regel aus mehreren Reihen von Bäumen. Um ihn darzustellen, benötigen wir für jede Baumreihe einen eigenen Vektor. Einfacher und übersichtlicher ist es, wenn wir diese Zahlenreihen nebeneinander schreiben. Damit entsteht ein zweidimensionales Zahlenschema, eine sogenannte **Matrix**:

Ein 4x4 Wald in Vektorschreibweise:

$$\vec{v}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \quad \vec{v}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad \vec{v}_4 = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Der selbe 4x4 Wald in Matrixschreibweise:

$$\vec{v}_1 = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Eine Matrix ist im Prinzip nichts anderes, als eine Liste von Listen. Mit dem Python-Paket `numpy` bekommen wir Zugriff auf zahlreiche "Werkzeuge" zum erleichterten Arbeiten mit Matrizen. Das Anlegen einer leeren 4x4 Matrix würde beispielsweise ohne `numpy` so aussehen:

```
wald2d = [[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]]
```

was für größere Matrizen zunehmend unübersichtlich würde. Einfacher geht es mit `numpy`:

```
wald2d = np.zeros((4,4))
```

Das Lesen und Ändern von Matrixeinträgen funktioniert ähnlich wie das Lesen und Ändern von Vektor-, sprich Listenelementen: Der erste Eintrag eines Vektors (einer Liste) ist der Eintrag `vektorname[0]`. In einer Matrix lautet der erste Eintrag links oben `matrixname[0][0]`.

Wenden wir dieses Wissen über Matrizen jetzt auf unser Waldmodell an: Erstellen wir zuerst einen leeren Wald, also eine 10x10 Matrix, die mit Nullen gefüllt ist.

Im nächsten Schritt sollte an jede Stelle eine zufällige Zahl (entweder 1 oder 0) gesetzt werden. `int(random.uniform(0, 2))` erzeugt uns genau so eine Zahl (genaugenommen eine Zahl zwischen 0 und 2, die dann abgerundet wird). Aber wie machen wir das für jeden Eintrag der Matrix?

Der Befehl ist immer der gleiche, aber das Ziel des Befehls ändert sich fortlaufend, nämlich vom Element `[0][0]`, zu `[0][1]`, zu `[0][2]` und so weiter. Um also alle Matrizenstellen mit Zufallszahlen füllen, macht es Sinn eine sogenannte **Doppelschleife** zu verwenden.

Eine Doppelschleife ist eine Schleife innerhalb einer Schleife. Die innere Schleife führt den gewünschten Befehl für eine ganze Zeile der Matrix aus, die äußere Schleife sorgt dafür, dass die innere Schleife für jede Zeile ausgeführt wird. Wichtig dabei ist, dass die Durchlaufvariablen (`it`) nicht den gleichen Namen haben dürfen.

Wenn der Wald fertig erstellt ist, geben wir ihn mit einem einfachen Printbefehl an den Bildschirm aus.

```
In [4]: import matplotlib.pyplot as plt
import random
import numpy as np
%matplotlib inline

# erzeuge eine 'leere', sprich mit Nullen gefüllte Matrix
wald2d=np.zeros((10,10))

# erste Schleife für die verschiedenen Reihen
for it in range(10):
    # zweite Schleife für die Elemente in den Reihen
    for jt in range(10):
        wald2d[it][jt] = int(random.uniform(0, 2))

print(wald2d)

[[ 1.  1.  0.  1.  0.  1.  0.  1.  0.  1.]
 [ 1.  0.  1.  0.  1.  1.  0.  1.  0.  0.]
 [ 1.  0.  1.  0.  1.  0.  1.  1.  0.  0.]
 [ 1.  1.  1.  0.  1.  1.  1.  1.  0.  1.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  1.  0.  1.  0.  1.  0.  0.]
 [ 0.  1.  1.  1.  0.  0.  0.  1.  1.  0.]
 [ 0.  0.  1.  1.  0.  1.  1.  1.  0.  1.]
 [ 0.  0.  1.  0.  0.  0.  1.  1.  0.  0.]
 [ 1.  0.  0.  1.  0.  1.  0.  0.  1.  1.]]
```

Man kann sich so schon ein wenig vorstellen, wie der Wald aussehen könnte. Besser wäre aber eine grafische Ausgabe. Ein Befehl, den wir zum Plotten von Matrizen verwenden können, heißt `plt.matshow`. Damit können wir einerseits angeben, welche Matrix geplottet werden soll, andererseits auch in welcher Farbe. Die Farbdarstellung in dieser Art von Plots erlaubt es Farben in Abhängigkeit der Größe eines Matrizeintrages zu gestalten. dazu wird ein Farbverlauf definiert. Obwohl wir hier einstweilen nur Nullen und Einsen als Einträge haben, wählen wir einen Farbverlauf, der von Weiß nach Grün geht. Er heißt `plt.cm.Greens`.

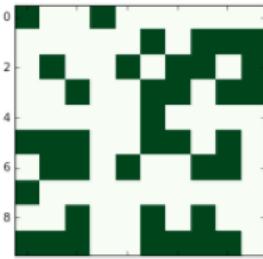
```
In [5]: import matplotlib.pyplot as plt
import random
import numpy as np
%matplotlib inline

wald2d = np.zeros((10, 10))

for it in range(10):
    for jt in range(10):
        wald2d[it][jt] = int(random.uniform(0, 2))

plt.matshow(wald2d, cmap = plt.cm.Greens)
```

Out[5]: <matplotlib.image.AxesImage at 0xb7c72b0>



Bisher berücksichtigen wir zur Darstellung unseres Waldes nur die Werte 0 und 1 (kein Baum, Baum). Wenn wir aber zum Beispiel auch Daten über die Größe oder die Dichte der Bäume in unserem Wald hätten, so könnten wir dies mit Werten zwischen 0 und 1 repräsentieren. Zur Darstellung müssen wir dank Farbverlauf sonst nichts an unserem Programm ändern.

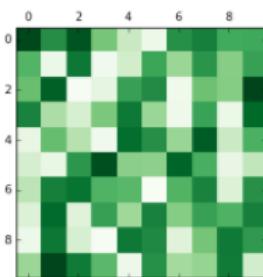
```
In [6]: import matplotlib.pyplot as plt
import random
import numpy as np
%matplotlib inline

wald2d=np.zeros((10, 10))

for it in range(10):
    for jt in range(10):
        wald2d[it][jt] = random.uniform(0, 1)

plt.matshow(wald2d, cmap = plt.cm.Greens)
```

Out[6]: <matplotlib.image.AxesImage at 0xb960b70>



In diesem Plot sehen wir nun dunklere und hellere Bereiche des Waldes. Anhand dieser Darstellung könnten wir uns überlegen, in welchen Bereichen es Sinn machen würde, Holz aus dem Wald zu entfernen. Praktisch wäre es, die besonders dunklen Bereiche des Waldes durch Fällen von Bäumen zu lichten, damit dort wieder neue Bäume wachsen können.

Wie finden wir nun aber zuverlässig die dunklen Bereiche des Waldes? Einfach nur eine große Zahl in der Matrix zu suchen, reicht nicht aus: Damit könnte es sein, dass wir einen vereinzelten großen Baum fällen, der gar keine Nachbarn hat. Wir müssen also bei unserer Suche sicherstellen, dass auch auf den je benachbarten Bereichen große Bäume stehen.

Dazu müssen wir zuerst eine Grenze festlegen, ab der wir einen Baum als "groß" einstufen. In diesem Beispiel nehmen wir den Wert 0.75 an. Nachdem wir den Wald generiert haben, iterieren wir in einer weiteren Doppelschleife durch alle Positionen des Waldes und überprüfen, ob ein Baum an dieser Stelle "groß" ist. Sodann überprüfen wir zusätzlich alle vier Nachbarn an dieser Position. Steht an einem dieser Nachbarorte auch ein großer Baum, so wird der ursprüngliche, also der mittlere Baum gefällt.

In unserer Matrendarstellung sind die vier Nachbarn des Baumes `wald2d[it][jt]`

- `wald2d[it+1][jt]`
- `wald2d[it-1][jt]`
- `wald2d[it][jt+1]`
- `wald2d[it][jt-1]`

Um nun den Wald vor und nach diesem Auslichten zu betrachten, erstellen wir zwei Plots. Wir verwenden dazu den Befehl `plt.show()`, der den Plot zeichnet, bevor das restliche Programm ausgeführt wird.

```
In [7]: import matplotlib.pyplot as plt
import random
import numpy as np
%matplotlib inline

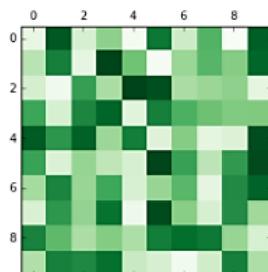
wald2d = np.zeros((10, 10))

# Wald wir generiert
for it in range(10):
    for jt in range(10):
        wald2d[it][jt] = random.uniform(0, 1)

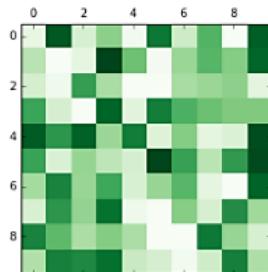
# erste Darstellung
plt.matshow(wald2d, cmap = plt.cm.Greens)
plt.show()

# Wald wird ausgelichtet
for it in range(1, 9):
    for jt in range(1, 9):
        if wald2d[it][jt] > 0.75:
            if wald2d[it+1][jt] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it-1][jt] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it][jt+1] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it][jt-1] > 0.75:
                wald2d[it][jt] = 0

# zweite Darstellung
plt.matshow(wald2d, cmap = plt.cm.Greens)
```



Out[7]: <matplotlib.image.AxesImage at 0xc4c04e0>



Um in diesen Plots zu sehen, wo genau die Bäume gefällt wurden, muss man sehr genau hinschauen. Lassen sich die Orte, an denen gefällt wurde, einfacher finden? Gesucht ist also eine Methode zur Darstellung des Unterschieds zwischen erstem und zweiten Plot.

Unterschiede sind mathematisch ausgedrückt Differenzen. Das Rechnen mit Matrizen bietet Vorteil, eine Matrix einfach von einer anderen subtrahieren zu können. Wir bauen diese Möglichkeit in das Programm ein und erstellen einen Plot der uns sagt, wo Bäume gefällt wurden.

Dazu müssen wir den zunächst generierten Wald unter einem eigenen Namen zwischenspeichern, damit wir ihn beim Fällen nicht überschreiben.

```
In [8]: import matplotlib.pyplot as plt
import random
import numpy as np
%matplotlib inline

wald2d = np.zeros((10, 10))
for it in range(10):
    for jt in range(10):
        wald2d[it][jt] = random.uniform(0, 1)

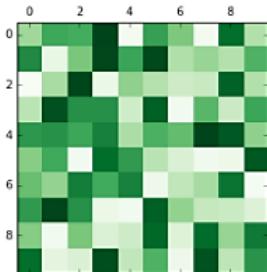
# erste Darstellung, der generierte Wald
plt.matshow(wald2d, cmap = plt.cm.Greens)
plt.show()

# Zwischenspeichern
wald2dalt = np.array(wald2d)

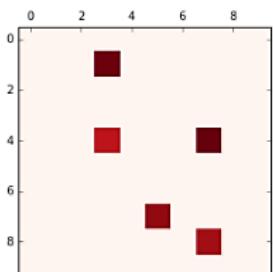
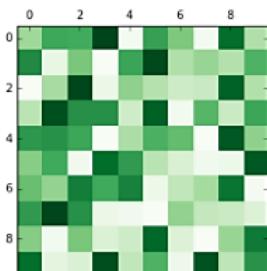
for it in range(1, 9):
    for jt in range(1, 9):
        if wald2d[it][jt] > 0.75:
            if wald2d[it+1][jt] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it-1][jt] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it][jt+1] > 0.75:
                wald2d[it][jt] = 0
            if wald2d[it][jt-1] > 0.75:
                wald2d[it][jt] = 0

# zweite Darstellung, der gelichtete Wald
plt.matshow(wald2d, cmap = plt.cm.Greens)

# dritte Darstellung, die Differenz, diesmal in Rot-Tönen
plt.matshow(wald2dalt - wald2d, cmap = plt.cm.Reds)
```



Out[8]: <matplotlib.image.AxesImage at 0xc7fa2e8>



Zusammenfassung

Vektoren

Vektoren sind so etwas wie Listen, deren Einträge in der Mathematik gewöhnlich übereinandergeschrieben werden. $\vec{v}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$

Durch Vektoren lässt sich genauso iterieren, wie durch Listen. Beachte, dass das erste Element (der erste Eintrag) in einem Vektor als 'nullter' Eintrag gezählt wird (`vektorname[0]`). Der zweite Eintrag hat somit die Bezeichnung `vektorname[1]`, usw.

Matrizen

Matrizen sind Vektoren von Vektoren, oder am Computer: Listen von Listen. $\vec{v}_1 = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$

Matrizen-Einträge werden mit `matrixname[0][0]`, `matrixname[0][1]`, ..., `matrixname[n][m]` adressiert. Zum Iterieren durch eine Matrix kann eine Doppelschleife verwendet werden.

Auf Matrizen können eine Reihe von Grundrechenarten (Addieren, Subtrahieren, Multiplizieren ...) angewendet werden.

Random

Zum Erzeugen von Zufallszahlen kann das Python-Paket (bzw. Modul) `random` verwendet werden.

Kapitel 6 - Rekursionen und Zellulare Automaten

Um die Besonderheiten von Phänomenen, die in den Systemwissenschaften relevant sind, gut zu verstehen, bleiben wir im Folgenden zunächst noch im Bereich der **diskreten** Entwicklungen und wenden uns in diesem Kapitel einer eher einfachen, aber sehr grundlegenden Art der Entwicklung zu, der **Rekursion**.

Systemwissenschaftlich sind Rekursionen relevant, weil sie das **EIGENVERHALTEN** von Systemen vor Augen führen.

In der Mathematik bezeichnet man als Rekursion eine Operation, die auf das Produkt, das sie hervorbringt, neuerlich - und wiederholt - angewandt wird. Jede wiederholte Anwendung wird als **Iteration** bezeichnet. Beispiele dafür haben wir schon mit der Fibonacci-Reihe (Kapitel 2 dieses Skriptums) kennengelernt. Rekursionen sind vom Prinzip her **Differenzengleichungen**.

Ein einfaches und grundlegendes Beispiel für eine Rekursion stellt die folgende Operation op dar:

$$op = \frac{op}{2} + 1$$

oder genauer - nämlich als Differenzengleichung - formuliert:

$$op_{t+1} = \frac{op_t}{2} + 1$$

In Python lässt sich dies wie folgt berechnen:

```
In [1]: # definiere eine Liste, die die Zahl der Iterationen enthält
T = range(10)      # T = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# bestimme einen Anfangswert. Probiere auch andere Werte!
op = 4 #3 #-1 ...

# definiere ein Schleife zum Wiederholen (Iterieren) der Operation
for t in T:
    # definiere die Operation.
    # Merke: in der Version Python 2.x muss, um eine Division korrekt auszuführen,
    # stets durch reelle Zahlen dividiert werden. Daher der Punkt nach 2.
    op = (op/2.) + 1
    # schreibe das Ergebnis nach jedem Schleifendurchlauf an
    print(op)

3.0
2.5
2.25
2.125
2.0625
2.03125
2.015625
2.0078125
2.00390625
2.001953125
```

Die Rekursion der Operation op scheint auf die Zahl 2 zu zustreben. Wir können dies verifizieren, indem wir die Zahl der Iterationen erhöhen, z.B. auf $T = range(20)$.

Wenn wir die Anfangswerte, von denen aus op gestartet wird, variieren, sehen wir, dass op offensichtlich von vielen unterschiedlichen Anfangswerten aus auf 2 zustrebt.

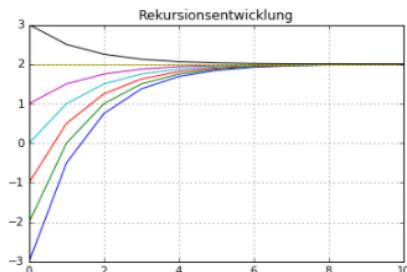
Deutlich lässt sich dies mit Python zeigen, wenn die Ergebnisse dieser unterschiedlichen Versuche nicht angeschrieben, sondern gezeichnet (geplottet) werden. Wir definieren dazu zwei Listen, eine namens AW , die verschiedenen Anfangswerte enthält, und sodann in jedem Schleifendurchlauf eine weitere Liste namens OP , in die die vom jeweiligen Anfangswert aus errechneten Iterationsergebnisse eingetragen werden, um sodann, ebenfalls in jedem Schleifendurchlauf, gezeichnet zu werden. Überdies benötigen wir für diese Berechnung die Python-Module `numpy` (für numerisches Rechnen) und `matplotlib.pyplot` (für Zeichnen).

```
In [2]: # importiere die benötigten Module (Pakete)
import numpy as np
import matplotlib.pyplot as plt
# zeichne direkt in das Notebook
%matplotlib inline

# definiere eine Liste mit Anfangswerten, hier von -3 bis (exklusive) 4
AW = np.arange(-3, 4)
# Simulationszeit
T = range(10)

# definiere eine (erste) Schleife zum Wiederholen der Rekursion mit unterschiedlichen Anfangswerten
for aw in AW:
    # definiere eine Liste und schreibe den jeweiligen Anfangswert hinein
    OP = [aw]
    # definiere ein (zweite) Schleife zum Wiederholen (Iterieren) der Operation
    for t in T:
        # definiere die Operation und schreibe das Ergebnis jeweils ans Ende der Liste OP
        OP.append((OP[t]/2.) + 1)
    # zeichne den Inhalt der Liste OP
    plt.plot(OP)

# gib der Zeichnung einen Titel
plt.title('Rekursionsentwicklung')
# füge ein Koordinatensystem (ein Grid) zur leichteren optischen Auflösung hinzu
plt.grid()
```



Die Operation $op = \frac{op}{2} + 1$ strebt für die unterschiedlichen Anfangswerte auf die Zahl 2 zu. Man spricht diesbezüglich davon, dass die Operation op in 2 einen **EIGENWERT** hat. Sie zeigt damit charakteristisches **Systemverhalten**, d.h. op tut was sie tut unabhängig vom jeweiligen Ausgangszustand. Der Ausgangszustand ist eigentlich irrelevant für das Verhalten des Systems. Das System hat ein **EIGENVERHALTEN**, es folgt seinem Eigenwert.

Systeme sind dadurch charakterisiert, dass sie ein Eigenverhalten zeigen, d.h. dass sie ein von ihren Ausgangszuständen oder vom Zustand ihrer Komponenten unabhängiges Verhalten an den Tag legen, das sich nicht ohneweiteres vorhersagen lässt. Dieses Eigenverhalten und der Versuch, es vorherzusagen, ist Gegenstand der **Systemwissenschaften**.

Funktionen

Wie eingangs in diesem Skriptum erwähnt, ist Python, wie nahezu alle Programmiersprachen, **modular** aufgebaut. D.h. die Operation op lässt sich als 'Werkzeug' mit einer Schnittstelle für einen Inputwert definieren, als so genannte **Funktion**, die sich sodann in dieser Definition wiederholt verwenden lässt. Diese Funktionendefinition sieht in Python wie folgt aus:

```
In [3]: # Definition einer Funktion
def op(x):
    return (x/2.) + 1
```

Der `return`-Befehl bestimmt das, was die Funktion als Output 'zurückliefert'.

Nachdem sie definiert wurde, lässt sich eine Funktion wie folgt verwenden:

```
In [4]: # drei einzelne Iterationsschritte vom Anfangswert 4 weg
x = op(4)
print(x)
x = op(x)
print(x)
x = op(x)
print(x)

3.0
2.5
2.25
```

Auch solche Funktionen können **rekursiv** aufgerufen werden. Aber **Vorsicht(!)**: sie benötigen eine Stoppbedingung, um nicht, ähnlich der Rückkoppelung eines pfeifenden Mikrofons, in einen unendlichen Regress zu verfallen.

Das Folgende zeigt zwei Beispiele für solche Rekursionen mit Stoppbedingungen.

```
In [5]: # Beispiel 1
x = 4    # Anfangswert
n = 0    # ein Zählerwert zum Erfüllen der Stopbedingung

# eine while-Schleife (lies: während (oder solange) die Bedingung n < 10 erfüllt ist, tue alles was eingerückt darunter steht)
while n < 10:
    x = op(x)    # rechne und weise x das Ergebnis zu
    print(x),    # schreibe das Ergebnis an, in diesem Fall in einer Zeile hintereinander
    n += 1        # inkrimiere den Zählerwert um 1

print '\n' # füge eine Leerzeile ein

# Beispiel 2
x = 4    # Anfangswert

while x > 2.003:
    x = op(x)
    print(x),

3.0 2.5 2.25 2.125 2.0625 2.03125 2.015625 2.0078125 2.00390625 2.001953125
3.0 2.5 2.25 2.125 2.0625 2.03125 2.015625 2.0078125 2.00390625 2.001953125
```

Countdown

Um das Prinzip solcher Funktionsdefinitionen und die Möglichkeit ihres **Selbstaufrufs** noch deutlicher zu machen, sei hier noch das bekannte Beispiel eines Countdown angeführt:

```
In [6]: def countdown(n):
    if n == 0:
        print("Start!")
    else:
        print(n)
        countdown(n-1)

countdown(10)

10
9
8
7
6
5
4
3
2
1
Start!
```

Ein 1-dimensionales Zellulares Automaton

Ein bekanntes Beispiel für Rekursionen stellen **zelluläre Automaten** dar, auf Englisch *Cellular Automata* oder kurz CAs. (siehe auch: http://systems-sciences.uni-graz.at/etextbook/sw2/ca_1d.html). Zu anderen Beispielen für Rekursionen siehe auch: <http://systems-sciences.uni-graz.at/etextbook/sw2/recursions.html>

CAs sind räumlich und zeitlich diskrete mathematische Systeme, charakterisiert durch lokale Interaktion und parallele Evolution. In CAs hängt der Zustand der Zellen, aus denen sie bestehen, deterministisch vom Zustand der jeweiligen Nachbarzellen ab. Im hier betrachteten Fall elementarer ein-dimensionaler CAs sind diese Zustände einfach binär definiert (1 oder 0 oder schwarz oder weiß). Eine typische Regel, die diese Zustände im zeitlichen Ablauf bestimmt, könnte z.B. lauten:

Wenn eine Zelle x im Zeitschritt t den Wert 1 hat und die linke Nachbarzelle $x - 1$ ebenfalls den Wert 1 hat, die rechte Nachbarzelle $x + 1$ dagegen den Wert 0 hat, so erhält Zelle x im darauffolgenden Zeitschritt $t + 1$ den Wert 0.

$x - 1$	x_t	$x + 1$	$>$	x_{t+1}
1	1	0	>	0

Gewöhnlich werden solche Regeln in Form von **Regelsets** definiert, die für sämtliche Permutationen der Nachbarzellen die entsprechenden Zustände im Zeitpunkt $t + 1$ angeben. Zum Beispiel:

$x - 1$	x_t	$x + 1$	$>$	x_{t+1}
1	1	1	>	0
1	1	0	>	0
1	0	1	>	0
1	0	0	>	1
0	1	1	>	0
0	1	0	>	1
0	0	1	>	1
0	0	0	>	0

In der Ausführung von solchen CAs werden zunächst zufällige, binäre Ausgangsreihen, also z.B. zufallsgegeneerte Reihen von Nullen und Einsen, als **Anfangswerte** erzeugt (siehe den Output des nächsten Code-Segments), auf die dann iterativ - Zeile für Zeile, also von oben nach unten - das jeweilige Regelset angewandt wird. Auch hier zeigt sich, dass der Effekt der Interaktionen von den konkreten Anfangswerten unabhängig ist, die einzelnen Regelsets also ein **Eigenverhalten** zeigen.

Die Anfangswertreihen werden als zu Ringen zusammengeschlossen vorgestellt, d.h. in der Berechnung wird der erste Wert ganz links in der Reihe als benachbart mit dem letzten Wert ganz rechts in der Reihe vorgestellt.

Da die Permutationen der Zellzustände von $x - 1$, x_t und $x + 1$ für unterschiedliche Regelsets gleichbleiben, braucht ein Regelset im Prinzip immer nur die Zellzustände im Zustand $t + 1$ anführen. Die Regelsets bestehen damit aus Listen, die die acht Binärzustände auflisten. Als Einsen und Nullen gelesen können diese Listen als Binärzahlen interpretiert werden. Die Regelsets (oder einfach: die Regeln) werden deshalb nach dieser Binärzahl benannt. Das oben angeführte Regelset stellt zum Beispiel die **Regel 22** dar, weil die Binärzahl 00010110 im Dezimalsystem der Zahl 22 entspricht.

Mit Python lässt sich ein solches ein-dimensionales Cellulare Automaton wie folgt generieren:

Im ersten Schritt erzeugen wir eine Anfangswertreihe:

```
In [7]: # importieren das Python-Modul random zum Erzeugen von Zufallszahlen
import random as rd

# erzeuge eine zufällige Anfangswertreihe AW aus Einsen und Nullen mit der Länge L
AW = []
L = 30
for n in range(L):
    # füge L-mal zufällig eine 0 oder 1 hinten an Liste AW an
    AW.append(rd.randint(0,1))
print(AW)

[1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1]
```

Im nächsten Schritt definieren wir - zunächst per Hand - eine Regel, in Form einer Liste, deren Einträge als Binärzahl gelesen wird.

```
In [8]: # Regel 22
R = [0, 0, 0, 1, 0, 1, 1, 0]
```

Im nächsten Schritt definieren wir eine Funktion mit Namen *iteration*, die die Anfangswertreihe (Line) und die Regel (Rule) als Input aufnimmt. Im ersten Teil dieser Funktion wird dafür gesorgt, dass die ersten und letzten Zellen jeder Reihe (jeder Liste) als benachbart behandelt werden. Im zweiten Teil werden einfach die Ergebnisse der Regelanwendung in eine Liste namens Newline eingetragen.

```
In [9]: def iteration(Line, Rule):
    Newline = []
    for n, z in enumerate(Line):
        # schließe die Zellenreihe zu einem Ring zusammen
        if n == 0:                                # im Fall der ersten Zelle
            zm1 = Line[-1]                         # die Zelle vor der betrachteten Zelle ('Zelle minus 1')
            zp1 = Line[n + 1]                        # die Zelle nach der betrachteten Zelle ('Zelle plus 1')
        if n == len(Line) - 1:                      # im Fall der letzten Zelle
            zm1 = Line[n - 1]                       # die Zelle vor der betrachteten Zelle ('Zelle minus 1')
            zp1 = Line[0]                           # die Zelle nach der betrachteten Zelle ('Zelle plus 1')
        if n != 0 and n != len(Line) - 1:           # für alle anderen Zellen
            zm1 = Line[n - 1]
            zp1 = Line[n + 1]

        # Regelset
        if zm1 == 1 and z == 1 and zp1 == 1:
            # füge den Null-ten Eintrag der Regel zu Newline hinzu
            Newline.append(Rule[0])
        if zm1 == 1 and z == 1 and zp1 == 0:
            Newline.append(Rule[1])
        if zm1 == 1 and z == 0 and zp1 == 1:
            Newline.append(Rule[2])
        if zm1 == 1 and z == 0 and zp1 == 0:
            Newline.append(Rule[3])
        if zm1 == 0 and z == 1 and zp1 == 1:
            Newline.append(Rule[4])
        if zm1 == 0 and z == 1 and zp1 == 0:
            Newline.append(Rule[5])
        if zm1 == 0 and z == 0 and zp1 == 1:
            Newline.append(Rule[6])
        if zm1 == 0 and z == 0 and zp1 == 0:
            Newline.append(Rule[7])

    return(Newline)
```

Im letzten Schritt brauchen wir nun nur noch die Funktion `iteration` rekursiv (in diesem Fall 20 mal) ausführen.

```
In [10]: # schreibe zuerst die Anfangswertreihe an
print(AW)
# nimm diese Anfangswertreihe als Ausgangspunkt für die Regelanwendung
Newline = AW

for i in range(20):
    # wende die Regel an
    Newline = iteration(Newline, R)
    # schreib die Ergebnisse jeweils in neue Zeilen
    print(Newline)
```

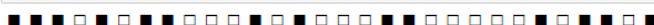
Leider ist das Ergebnis dieser Iterationen, d.h. der Output des CA für die Regel 22, in mit Einsen und Nullen gefüllten Listen nicht gut zu erkennen. Wir können deshalb versuchen, anstelle von Einsen und Nullen, schwarze und weiße Zellen zu zeichnen. Die ASCII-Codierung für schwarze und weiße Zellen lauten `unichr(0x2B1B)` und `unichr(0x2B1C)`.

Die obige Anfangswertzeile in dieser Weise dargestellt, lässt sich wie folgt generieren:

```
In [11]: Awbw = []          # Leere Liste für die neuen Anfangswerte
B = unichr(0x2B1B)      # weiße Zelle
W = unichr(0x2B1C)      # schwarze Zelle

for x in Awbw:
    if x == 1:
        Awbw.append(B)
    else:
        Awbw.append(W)

# diese Zellen lassen sich nun nicht mehr in einer Liste darstellen. Sie müssen einzeln ausgelesen werden
c = 0
while c < len(Abwb):
    print(Abwb[c]),
    c += 1
```



Nun können wir die gleichen Schritte wie oben wiederholen, nur dass wir dieses Mal die Zellzustände statt als Einsen und Nullen als B (für black square) und W (für white square) bezeichnen.

```
In [12]: def iteration_2(Line, Rule):
    Newline = []
    for n, z in enumerate(Line):
        # schließe die Zellenreihe zu einem Ring zusammen
        if n == 0:
            zm1 = Line[-1] # im Fall der ersten Zelle
            zp1 = Line[n + 1] # die Zelle vor der betrachteten Zelle ('Zelle minus 1')
        if n == len(Line) - 1:
            zm1 = Line[n - 1] # im Fall der letzten Zelle
            zp1 = Line[0] # die Zelle nach der betrachteten Zelle ('Zelle plus 1')
        if n != 0 and n != len(Line) - 1: # für alle anderen Zellen
            zm1 = Line[n - 1]
            zp1 = Line[n + 1]

        # Regelset
        if zm1 == B and z == B and zp1 == B:
            # füge den Null-ten Eintrag der Regel zu Newline hinzu
            Newline.append(Rule[0])
        if zm1 == B and z == B and zp1 == W:
            Newline.append(Rule[1])
        if zm1 == B and z == W and zp1 == B:
            Newline.append(Rule[2])
        if zm1 == B and z == W and zp1 == W:
            Newline.append(Rule[3])
        if zm1 == W and z == B and zp1 == B:
            Newline.append(Rule[4])
        if zm1 == W and z == B and zp1 == W:
            Newline.append(Rule[5])
        if zm1 == W and z == W and zp1 == B:
            Newline.append(Rule[6])
        if zm1 == W and z == W and zp1 == W:
            Newline.append(Rule[7])

    c = 0
    while c < len(Newline):
        print(Newline[c],
              c += 1
    print('\n')
    return(Newline)
```

```
In [13]: # Regel 22
#RR = [N, W, W, B, W, B, B, W]

# Regelgenerator
rule = 22
# eine Möglichkeit, Dezimalzahlen direkt in Binärzahlen umzuwandeln
Ru = list(format(rule, '08b'))
RR = [B if i=='1' else W for i in Ru]

# zeichne Anfangswertzeile
c = 0
while c < len(Awbw):
    print(Awbw[c]),
    c += 1
print('\n')

# iteriere CA
Newline = Awbw
for i in range(20):
    Newline = iteration_2(Newline, RR)
```

Noch deutlicher lässt sich unser CA allerdings darstellen, wenn wir das obige Binärzahlergebnis in eine Matrix stellen und uns diese mit Hilfe des matplotlib-Befehls `matshow` (siehe Kapitel 6 dieses Skriptums) darstellen lassen.

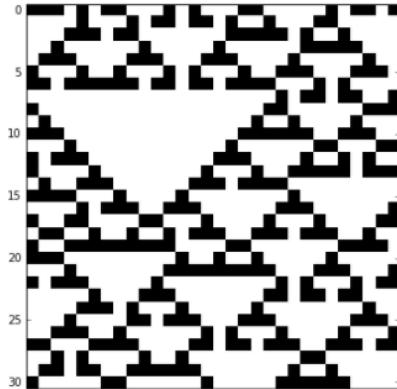
```
In [14]: import matplotlib.pyplot as plt
%matplotlib inline

####
# Regelgenerator
rule = 22
# eine Möglichkeit, Dezimalzahlen direkt in Binärzahlen umzuwandeln
Ru = list(format(rule, '08b'))
RR = [1 if i=='1' else 0 for i in Ru]
###

fig = plt.figure(figsize=(6,10))
ax1 = fig.add_subplot(111)

M = [AW]
Newline = AW
for i in range(30):
    Newline = iteration(Newline, RR)
    M.append(Newline)
ax1.matshow(M, cmap = plt.cm.Greys)
ax1.set_xticks([])
```

Out[14]: []



Zusammenfassung

Rekursionen

Rekursionen sind Operationen, die auf das Produkt, das sie hervorbringen, neuerlich - und wiederholt - angewendet werden. Jede wiederholte Anwendung wird als **Iteration** bezeichnet.

Eigenverhalten

Systeme zeigen Eigenverhalten, wenn ihre Entwicklung oder der Zustand, den sie mit der Zeit einnehmen, von ihren Ausgangszuständen oder vom Zustand ihrer Komponenten unabhängig ist und sich überdies nicht ohne weiteres vorhersagen lässt. Dieses Eigenverhalten und der Versuch, es (trotzdem) vorherzusagen, sind Gegenstand der Systemwissenschaften.

Beispiele für Eigenverhalten sind etwa die spezifischen Muster von zellulären Automaten.

while-Schleife

Eine `while`-Schleife dient - ähnlich wie die `for`-Schleife - dem Iterieren, sprich dem Wiederholen bestimmter Operationen oder Berechnungen.

Eine `while`-Schleife benötigt in der Regel eine Stop-Bedingung, um nicht unendlich weiter zu laufen (und damit u.U. den Computer zum Absturz zu bringen).

Zelluläre Automaten

Zelluläre Automaten (CAs) sind räumlich und zeitlich diskrete mathematische Systeme, die aus einzelnen Zellen mit mehreren (oftmals nur zwei) Zustandsmöglichkeiten bestehen, und ihre Zustände (z.B. schwarz oder weiß) deterministisch über ihren eigenen Zustand und den ihrer jeweiligen Nachbarzellen im vorhergehenden Zeitschritt bestimmen.

Zelluläre Automaten eignen sich gut, um systemisches Eigenverhalten zu veranschaulichen.

Kapitel 7 – Differenzieren und Integrieren – Ein Solarauto

Wie wir schon in den bisherigen Kapitel gesehen haben, interessieren sich die Systemwissenschaften vor allem für **dynamische Systeme**, d.h. für Systeme, deren Variablen oder Zustände sich verändern. Um Veränderungen, Bewegungen oder Entwicklungen zu erfassen, stellt die Mathematik eine elaborierte Methodik bereit, die **Differential- und Integralrechnung**. Beginnend mit diesem Kapitel beschäftigen wir uns mit dieser für die Systemwissenschaften so zentralen Methode, dies allerdings nicht primär aus der Perspektive der Mathematik, sondern eher aus der der Möglichkeiten, die der Computer bietet.

Als Beispiele betrachten wir in diesem Kapitel den Prozess der Stromerzeugung mithilfe eines Solarpanels und den Betrieb eines Elektrofahrzeugs. Wir nehmen - um auch dies in Python kennenzulernen - an, dass dafür das Einlesen und Bearbeiten von Daten aus externen Dateien notwendig ist, d.h. aus Dateien, die wir am Computer gespeichert haben, aber noch nicht im Rahmen des Jupyter-Notebooks berücksichtigt haben (die entsprechenden Dateien <solargrob.txt>, <solarfein.txt> und <batterie.txt> stehen im Moodle-Repositorium zum Download bereit. Laden sie diese Dateien herunter und speichern sie sie auf ihrem Computer in den Ordner, in dem sie auch dieses Jupyter-Notebook gespeichert haben).

Einlesen von Daten

Zuerst interessiert uns hier, wieviel Energie ein Solarpanel liefert. In der Datei <solargrob.txt> wurde mitprotokolliert, wie viel Leistung das Panel zu welcher Stunde der Tageszeit liefert. Wir benötigen also eine Methode, wie wir diese Datei, bzw. den Inhalt dieser Datei in ein Jupyter-Notebook laden, um ihn mit Python bearbeiten zu können. Für das so genannte Einlesen von Daten gibt es in Python vorgefertigte Pakete. Eines davon trägt den Namen `csv` (für *comma separated values*), und ermöglicht es, Inhalte aus Dateien aufzubereiten.

In einem ersten Schritt laden wir dieses Paket in unser Notebook, zusammen mit dem bereits bekannten Paket `matplotlib` für wissenschaftliches Zeichnen.

```
In [1]: import csv
import matplotlib.pyplot as plt
%matplotlib inline
```

Im nächsten Schritt greifen wir auf die besagte Textdatei zu. (Die Datei <solargrob.txt> muss dazu im gleichen Ordner wie das Jupyter-Notebook gespeichert sein).

In der Datei sind 24 Einträge, einer für jede volle Stunde eines Messtages. Jeder Eintrag gibt die Leistung des Solarpanels zu dieser Stunde in Watt an. Um eine Datei in Python zu öffnen und ihr einen Namen zu geben (z.B. `inputfile`) kann man den Befehl `with open(filename.txt) as inputfile` verwenden. Alle Befehle, die diesen Namen `inputfile` verwenden sollen, müssen dazu eingerückt werden, ähnlich wie bei einer For-Schleife.

Wir öffnen die Datei und versuchen, ihren Inhalt mit dem `print`-Befehl auszugeben:

```
In [2]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

with open('solargrob.txt') as inputfile: #solargrob.txt wird geöffnet und heißt im programm jetzt inputfile
    print(inputfile)                      #inputfile soll geprintet werden

<open file 'solargrob.txt', mode 'r' at 0x000000000A4DC50>
```

Wir sehen: der `print`-Befehl liefert nicht das gewünschte Ergebnis. Wir sehen keine 24 Zahlen, sondern nur die Speicher-Adresse, unter der diese Datei Computer-intern geführt wird. Um den Inhalt der Textdatei zu extrahieren, ist vielmehr ein weiterer Schritt notwendig. Wir müssen über die gesamte Datei, Zeile für Zeile, iterieren und jede Zeile für sich ausgeben. Dazu benutzen wir wieder eine For-Schleife, kombiniert mit dem neuen Befehl `csv.reader`, der Inhalte aus Dateien liest.

```
In [3]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

with open('solargrob.txt') as inputfile: # solargrob.txt wird geöffnet und heißt im programm jetzt inputfile
    for row in csv.reader(inputfile):      # für jede Zeile innerhalb von inputfile
        print(row)                         # wird die Zeile ausgegeben

['3.77513454428e-08'],
[1.74472887359e-06],
[5.77774851942e-05],
[0.00137095908638],
[0.0233091011429],
[0.283962983903],
[2.47875217667],
[15.503853599],
[69.4834512228],
[223.130160148],
[313.417119033],
[446.481724891],
[1000.0],
[846.481724891],
[513.417119033],
[223.130160148],
[69.4834512228],
[15.503853599],
[2.47875217667],
[0.283962983903],
[0.0233091011429],
[0.00137095908638],
[5.77774851942e-05],
[1.74472887359e-06]
```

Das Ergebnis ist schon eher das, was wir erwarten. Wir können Zahlen erkennen. Die ersten Zahlen sind allerdings sehr klein (e-08 am Ende einer Zahl bedeutet, dass die vorne stehende Zahl mit 10^-8 multipliziert wird, also mit 0.00000001). Das Maximum der Zahlen liegt beim 13ten Eintrag. Es handelt sich um die Leistungsdaten eines Solarpannels.

Dennoch ist diese Ausgabe noch nicht perfekt, denn die Zahlen stehen innerhalb einiger Sonderzeichen, die uns beim Bearbeiten, also zum Beispiel beim Zeichnen eines Leistungsverlaufs, stören. Der Umstand, dass jeder Eintrag in eckigen Klammern steht, sagt uns, dass es sich bei den Einträgen um Listen handelt mit jeweils nur einem Eintrag. Hätten wir eine Datei mit mehreren Einträgen pro Zeile, wäre das vielleicht eine vernünftige Notation. In unserem Fall hätten wir aber lieber einzelne Einträge, und keine Listen der Länge 1. Wir spezifizieren also, dass wir von der ganzen Zeile, immer nur den ersten (und einzigen) Eintrag, also den Eintrag mit dem Index 0 brauchen:

```
In [4]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

with open('solargrob.txt') as inputfile: # solargrob.txt wird geöffnet und heißt im Programm jetzt inputfile
    for row in csv.reader(inputfile): # für jede Zeile innerhalb von inputfile
        print(row[0]) # wird der erste Eintrag jeder Zeile ausgegeben
```

3.77513454428e-08
1.74472887359e-06
5.77774851942e-05
0.00137095908638
0.0233091011429
0.283962983903
2.47875217667
15.503853599
69.4834512228
223.130160148
313.417119033
446.481724891
1000.0
846.481724891
513.417119033
223.130160148
69.4834512228
15.503853599
2.47875217667
0.283962983903
0.0233091011429
0.00137095908638
5.77774851942e-05
1.74472887359e-06

Das sieht nun schon besser aus. Als nächstes sollten wir noch den **Datentyp** überprüfen. Beim Einlesen von Daten geht Python nämlich oftmals davon aus, dass es sich dabei um Text (einfach gesagt: um Buchstaben, so genannte **strings**) handelt.

Merkel: Der Datentyp gibt an, in welcher Weise ein Datum - etwa eine Zahl oder ein Buchstabe - am Computer gespeichert wird. Gewöhnlich können Programmiersprachen den Datentyp nicht selbstständig erkennen. Eine '1' kann zum Beispiel als Buchstabe (**string**) oder als ganze Zahl (**integer**) oder auch als reelle Zahl (**float**) abgespeichert sein. Was für uns nahezu identisch aussieht und leicht in seiner Bedeutung verstanden wird, sind für den Computer drei völlig verschiedene Entitäten.

Deswegen sollten wir den Typ der Variablen, die ausgegeben werden soll, mit dem Befehl `type` abfragen. Da uns hier, um den Typ festzustellen, auch schon wenige Einträge reichen, begrenzen wir unsere Ausgabe mithilfe eines Zählers `c` und einer `if`-Abfrage auf fünf.

```
In [5]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

c = 0
with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        if c < 5:
            print(row[0])
            print(type(row[0])) # type ermittelt den Typ einer Variable
        c += 1 # das Zeichen += bedeutet: inkrementiere c um 1 (d.h. zähle in jedem Schritt um eins weiter)
```

3.77513454428e-08
<type 'str'>
1.74472887359e-06
<type 'str'>
5.77774851942e-05
<type 'str'>
0.00137095908638
<type 'str'>
0.0233091011429
<type 'str'>

Wie vermutet, sind die Daten als Text (d.h. als `<type 'str'>`) abgespeichert, nicht aber als Zahlen, mit denen man rechnen kann. Wir müssen diesen Text also zuerst in Fließkomma- (sprich reelle) Zahlen verwandeln, damit wir damit arbeiten können. Dies geschieht in Python mit dem Befehl `float` (Wir demonstrieren dies neuerlich nur für die ersten fünf Einträge und verwandeln die restlichen Einträge dann erst im nächsten Schritt unten in reelle Zahlen):

```
In [6]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

c = 0
with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        if c < 5:
            print(float(row[0])) # float konvertiert den Text in eine Zahl
            print(type(float(row[0])))
        c += 1 # inkrementiere c um 1 (d.h. zähle in jedem Schritt um eins weiter)

3.77513454428e-08
<type 'float'>
1.74472887359e-06
<type 'float'>
5.77774851942e-05
<type 'float'>
0.00137095908638
<type 'float'>
0.0233091011429
<type 'float'>
```

Als letzten Schritt unserer Datenvorbehandlung möchten wir die Zahlen nicht einfach nur auf den Bildschirm schreiben, sondern in einer Liste speichern, damit wir damit auch wirklich arbeiten können. Wir erstellen also eine leere Liste und füllen sie sodann mit den umgewandelten Einträgen. Erst dann lesen wir die gesamte Liste mithilfe des `print`-Befehls aus:

```
In [7]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

solargrob = [] # eine leere Liste wird erstellt

with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        solargrob.append(float(row[0])) # der erste Eintrag jeder Zeile wird als Zahl an die Liste angehängt
print(solargrob) # die gesamte Liste wird gedruckt

[3.77513454428e-08, 1.74472887359e-06, 5.77774851942e-05, 0.00137095908638, 0.0233091011429, 0.283962983903, 2.47875217667, 15.503853599, 69.4834512228, 223.130160148, 313.417119033, 446.481724891, 1000.0, 846.481724891, 513.417119033, 223.130160148, 69.4834512228, 15.503853599, 2.47875217667, 0.283962983903, 0.0233091011429, 0.00137095908638, 5.77774851942e-05, 1.74472887359e-06]
```

Nun können wir mit diesen Messdaten arbeiten. Wir können zum Beispiel einen Plot erstellen, der uns zeigt zu welcher Tageszeit das Solarpanel wie viel Leistung erbringt. Dazu können wir den Plotbefehl `plt.fill` benutzen, der im Unterschied zu `plt.plot` die Fläche unter einer Kurve miteinfarbt.

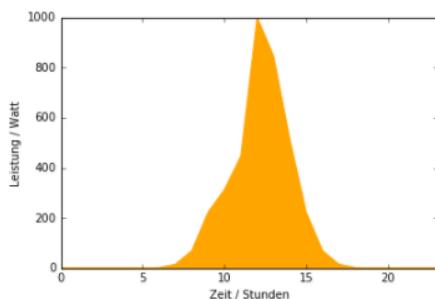
```
In [8]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

solargrob = []

with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        solargrob.append(float(row[0]))

plt.fill(range(24), solargrob, color = "orange") # fill funktioniert ähnlich wie plot, nur wird die Fläche gefüllt
plt.ylabel("Leistung / Watt")
plt.xlabel("Zeit / Stunden")
plt.xlim(0, 23)
```

Out[8]: (0, 23)



Numerisches Integrieren

Wir können nun den zeitlichen Verlauf der Leistung deutlich erkennen. Zur Spitzenzzeit um 12h liefert das Panel 1000 Watt (W), also genau 1 Kilowatt (kW). Was aber ist seine Gesamtleistung über 24 Stunden hinweg?

Wenn das Panel 24 Stunden lang stets 1000 Watt liefern würde, wäre die Berechnung der Gesamtleistung einfach. Die wäre dann einfach

$$1\text{ kW} * 24\text{ h} = 24\text{ kWh}$$

Wie wir im Plot sehen können, ist die Leistung aber zu jeder Tageszeit anders. Wir können also nicht 24 mal den gleichen Wert aufsummieren, um zur Gesamtleistung zu kommen. Wir müssen vielmehr für jede Stunde den Wert berücksichtigen, der in unserer Liste gespeichert ist. Wir haben in dieser Liste 24 Einträge und kennen somit die Leistung zu jeder Stunde der Tageszeit. Offensichtlich sollten wir also einfach die Einträge unserer Liste aufsummieren können, um die Tagesgesamtleistung zu bestimmen. Dieses Aufsummieren bedeutet praktisch, dass wir damit die gelb gezeichnete Fläche unter der Kurve näherungsweise bestimmen. (Nur näherungsweise deswegen, weil wir ja genaugenommen nur die Leistungsdaten zu jeder vollen Stunde berücksichtigen und nicht zB um 12.30h. Siehe dazu gleich unten).

Zum Berechnen der Fläche unter einer Kurve verwendet die Mathematik die **Integralrechnung**. Unser "Aufsummieren einer Liste" entspricht diesem Zweck. Man spricht diesbezüglich von der Methode der **numerischen Integration**.

Python stellt für diesen Vorgang des Aufsummierens den einfachen Befehl `sum` zur Verfügung. Im Folgenden benutzen wir diesen Befehl, um festzustellen, wieviel Leistung das Solarpanel über den Tag hinweg liefert:

```
In [9]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

solargrob = []

with open('solargrob.txt') as inputfile:
    for row in csv.reader(inputfile):
        solargrob.append(float(row[0]))

gesamtleistung = sum(solargrob) #sum addiert alle Elemente der Liste = numerische Integration
print(gesamtleistung)

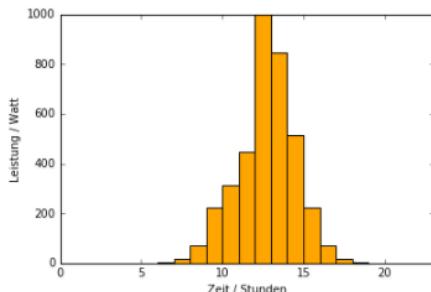
3741.60752731
```

In Summe produziert unser Solarpanel also etwa 3741 Wattstunden an einem Tag. Dieses Resultat ist freilich noch sehr ungenau. Warum? Wie gesagt, gehen wir bisher in dieser Betrachtung davon aus, dass es für jede Stunde des Tages einen fixen Leistungswert gibt. Und wir nehmen implizit an, dass dieser Wert innerhalb dieser Stunde konstant bleibt. Das ist natürlich in der Realität nicht der Fall.

Wir können den Effekt unserer Vereinfachung mit einem so genannten Bar-Plot darstellen. Damit können wir die Daten in Form von Rechtecken darstellen, also genau so, wie auch unser Aufsummieren funktioniert. Man sieht sofort, dass das nur eine Näherungslösung sein kann:

```
In [10]: plt.bar(range(24), solargrob, width = 1.0, color = "orange")
plt.xlim(0, 23)
plt.ylabel("Leistung / Watt")
plt.xlabel("Zeit / Stunden")
```

Out[10]: <matplotlib.text.Text at 0xad61630>



Erhöhen der Genauigkeit

Um die Genauigkeit unserer Ergebnisse zu erhöhen, haben wir prinzipiell zwei Möglichkeiten. Entweder wir finden bessere Integrationsmethoden (die später in diesem Skriptum behandelt werden) oder wir erhöhen einfach die Zeitauflösung unserer Messdaten.

Hier machen wir Zweiteres. Glücklicherweise können wir auf genauere Daten zurückgreifen. Zusätzlich zu der Datei, die wir schon eingelesen haben, liegt eine Datei vor, die minuten-genau Daten enthält, die also $24 * 60 = 1440$ Einträge hat. Im Folgenden lesen wir auch diese Datei in unseren Python-Code ein und erstellen einen Plot.

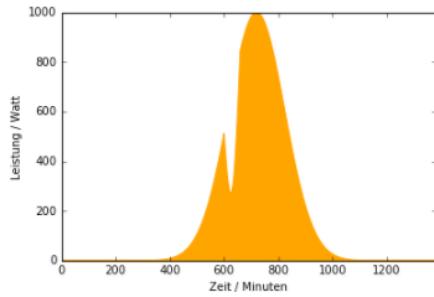
(Die Umrechnung von Stunden in Minuten muss hier nicht vorgezogen werden. Der Befehl `range(1440)` kann auch als `range(24 * 60)` geschrieben werden.)

```
In [11]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

solarfein = []
with open('solarfein.txt') as inputfile:
    for row in csv.reader(inputfile):
        solarfein.append(float(row[0]))

plt.fill(range(24 * 60), solarfein, color = "orange")
plt.ylabel("Leistung / Watt")
plt.xlabel("Zeit / Minuten")
plt.xlim(0, 23 * 60)
```

Out[11]: (0, 1380)



In dieser Auflösung sehen wir eine Reihe von Details, zum Beispiel, die Bewölkung, die offensichtlich den Vormittag kurz getrübt hat.

Es liegt nun nahe, auch mit diesen Daten eine numerische Integration durchzuführen, um die Gesamtleistung zu berechnen. Aber Vorsicht! Mit den gröberen Daten hatten wir 24 Zahlen aufsummiert, die größte davon war 1000. Nun wollen wir 1440 Zahlen aufsummieren, von denen wiederum die größte 1000 ist. Das kann kein ähnliches Ergebnis liefern. Offensichtlich haben wir etwas übersehen?

Für die gröberen Daten war unsere Argumentation wie folgt: Wenn das Solarpanel eine Stunde lang 1000 Watt liefert, produziert es eine Kilowattstunde. Deswegen war es zulässig, die Werte in der Liste einfach aufzuhaddieren.

Nun liegen uns aber minutengenaue Daten vor. Die Aussage "Wenn ein Solarpanel eine Minute lang 1000 Watt liefert, produziert es eine Kilowattstunde." wäre falsch. Richtig ist: "Wenn ein Solarpanel eine Minute lang 1000 Watt liefert, produziert es eine Kilowattminute, also ein Sechzigstel einer Kilowattstunde."

Die Werte, die in unserer Liste stehen haben also die falschen Einheit. Gegeben sind Kilowattminuten, wir hätten aber gerne Kilowattstunden, damit wir das Ergebnis der numerischen Integration (des Aufsummierens) mit unserer ursprünglichen Berechnung vergleichen können. Wir müssen diesen Umrechnungsfaktor (1/60) also miteinbeziehen und können erst danach summieren. Dazu erstellen wir eine For-Schleife, die uns jeden Wert der ursprünglichen Liste umrechnet und in einer neuen Liste speichert:

```
In [12]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

solarfein = []
with open('solarfein.txt') as inputfile:
    for row in csv.reader(inputfile):
        solarfein.append(float(row[0]))

solarfeinumgerechnet = []

for wert in solarfein:
    solarfeinumgerechnet.append(wert/60) # jeder Wert wird durch 60 dividiert und dann an die neue Liste gehängt

gesamtleistungfein = sum(solarfeinumgerechnet)
print(gesamtleistungfein)
```

4086.14193175

Wir sehen also, dass unser ursprüngliches Ergebnis (3.7 kW) noch relativ weit vom genaueren Ergebnis (4.1 kW) entfernt war. Außerdem fällt uns bei Betrachtung des Plots auf, dass der Großteil des Stroms zur Mittagszeit produziert wird. Interessant wäre nun etwa zu wissen, wie viel Prozent des Gesamtstroms wirklich zwischen 11 und 13 Uhr erzeugt werden.

Dazu erstellen wir abermals eine Summe, integrieren also numerisch. Diesmal wollen wir aber nicht das Integral über den gesamten Zeitraum, sondern nur über einen Teil. Dazu wählen wir diesen Teil der Liste aus. Wie geschieht dies?

Wir haben schon gelernt, dass wir einzelne Einträge einer Liste addressieren können. `listename[4]` liefert zum Beispiel den Eintrag mit dem Index 4, also den 5-ten Eintrag einer Liste. Wir können ganz ähnlich aber auch mehrere Elemente auswählen. Alle Elemente von 4 (einschließlich) bis 20 (ausschließlich) erhält man zum Beispiel durch `listename[4:20]`. Dieses Wissen können wir nun nutzen, um die Stromproduktion zur Mittagszeit zu berechnen:

```
In [13]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

solarfein = []
with open('solarfein.txt') as inputfile:
    for row in csv.reader(inputfile):
        solarfein.append(float(row[0]))

solarfeinumgerechnet = []

for wert in solarfein:
    solarfeinumgerechnet.append(wert/60)

gesamtleistungfein = sum(solarfeinumgerechnet) #gesamte Summe
print(gesamtleistungfein)

gesamtleistungmittag = sum(solarfeinumgerechnet[11 * 60 : 13 * 60]) #Summe der Werte zwischen 11 und 13 Uhr
print(gesamtleistungmittag)

relation = gesamtleistungmittag / gesamtleistungfein * 100      #Berechnen des prozentuellen Anteils
print("Relation in Prozent:")
print(relation)

4086.14193175
1894.1770029
Relation in Prozent:
46.3561235644
```

Etwa 46% des Gesamtstroms unseres Solarpanels werden also zur Mittagszeit produziert. Elektrische Geräte ausschließlich über das Solarpanel zu betreiben, scheint damit nicht ratsam, da diese Geräte nur zu Mittag gut funktionieren würden. Es könnte also vorteilhaft sein, den zu Mittag produzierten Strom in einer Batterie zu speichern, und dann wieder abzugeben, wann er gebraucht wird.

Wir könnten so zum Beispiel ein kleines Elektroauto betreiben.

Sehen wir uns diese Möglichkeit und speziell den Verbrauch dieses Fahrzeugs genauer an.

Verbrauch eines Elektrofahrzeugs

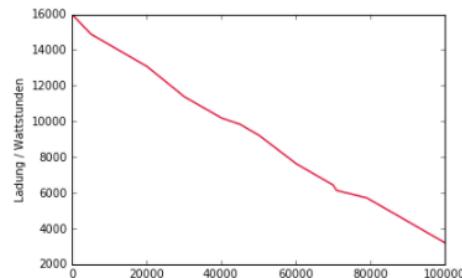
Zusätzlich zur Stromproduktion eines Solarpanels haben wir auch die Messdaten des Stromverbrauchs eines kleinen Elektrofahrzeugs zur Verfügung (laden sie die Datei batterie.txt aus dem Moodle-Repositorium herunter und speichern sie sie im Notebook-Ordner). Für eine Testfahrt wurde das Auto voll aufgeladen und dann nach jedem gefahrenen Meter der aktuelle Batteriestand protokolliert. Das Ergebnis sind zig-tausende Datenpunkte, die wir nicht mehr einfach per Hand auswerten können. In einem ersten Schritt wollen wir die Daten wieder in unseren Python-Code einlesen und grafisch darstellen:

```
In [14]: import csv
import matplotlib.pyplot as plt
%matplotlib inline

batterie = []
with open('batterie.txt') as inputfile:
    for row in csv.reader(inputfile):
        batterie.append(float(row[0]))
```

```
plt.plot(batterie, color = 'crimson', lw = 1.5)
plt.ylabel("Ladung / Wattstunden")
```

Out[14]: <matplotlib.text.Text at 0xaff485c0>



Anhand dieses Plots lässt sich schon einiges über das Testfahrzeug sagen. Bei voller Batterie besitzt es eine Ladung von 16000 Wattstunden, also 16 Kilowattstunden. Um voll aufgeladen zu sein, müsste es also etwa 4 Tage lang über unser Solarpanel laden.

Außerdem sehen wir, dass das Auto bei der Testfahrt genau 100000 Meter, also 100 Kilometer zurückgelegt hat. Nach dieser Distanz beträgt die verbleibende Ladung etwa 3 Kilowattstunden, es wurden also 13 Kilowattstunden verbraucht. Durch simple Division können wir feststellen, dass der mittlere Verbrauch des E-Autos etwa 13 Kilowattstunden auf 100 Kilometer, also 0.13 Kilowattstunden auf 1 Kilometer, beträgt.

Ebenfalls fällt auf: Der Verbrauch ist nicht durchgehend gleich. Manchmal sinkt der Batteriestand schneller, manchmal langsamer. Das kann mehrere Gründe haben: starkes Beschleunigen, aber auch das Halten einer hohen Geschwindigkeit erhöhen den Verbrauch.

Wir könnten uns nun also fragen, an welchen Abschnitten der Teststrecke der Verbrauch besonders hoch, bzw. besonders gering war. Dazu müssen wir allerdings definieren, dass mit Verbrauch die Ladungsverlust-Rate der Batterie gemeint ist. Grafisch ausgedrückt: Wie steil ist die Kurve der Funktion, die den Batterieverbrauch anzeigt. Mathematisch augendrückt: Wie groß ist die Änderung bzw. die Ableitung der Funktion an jedem Punkt.

Das heißt, wir suchen die **Ableitung der Funktion "Batterieladung"**. Die Funktion selbst sagt uns wie viel Ladung in der Batterie ist, die Ableitung sagt uns, wie sich diese Ladung verändert. Sie zeigt uns den Verbrauch. Mathematisch nennt man die Methode, um diesen Verbrauch zu ermitteln, **numerisches Differenzieren**.

Numerisches Differenzieren

Die Steigung einer Kurve (einer Funktion) auszurechnen, ist am Computer nicht schwierig. Wir kennen die Batterieladung an jedem Punkt unserer Teststrecke. Die Änderung der Batterieladung errechnet sich dann aus der Ladung an diesem Punkt minus der Ladung am unmittelbar nachfolgenden Punkt.

Auch für diese Art des numerischen Differenzierens gibt es einen Python-Befehl, der sich, zusammen mit vielen anderen Befehlen zur numerischen Berechnung im Paket `numpy` (*numerical python*) findet. Wir importieren dieses Paket und benutzen den Befehl `diff` um die Ableitung der Ladung, also den Verbrauch, auszurechnen.

Dabei ist es wichtig, auf das Vorzeichen zu achten: die Steigung unserer Kurve ist natürlich negativ (die Ladung wird immer weniger), der Verbrauch ist aber als positive Größe definiert. Man sagt

"Mein Auto verbraucht 5 Liter pro 100 Kilometer."

und nicht

"Der Tank meines Autos verändert sich um -5 Liter auf 100 Kilometer."

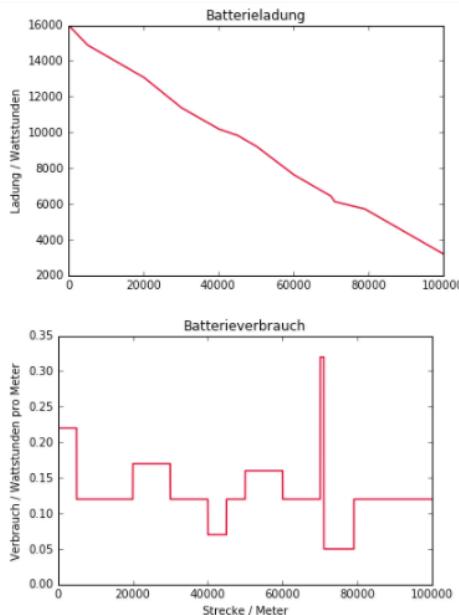
Der von uns gesuchte Verbrauch berechnet sich also als "Steigung mal -1". Im Folgenden zeichnen wir die Plots für die Batterieladung und für den Batterieverbrauch übereinander.

```
In [15]: import numpy as np
import csv
import matplotlib.pyplot as plt
%matplotlib inline

batterie = []
with open('batterie.txt') as inputfile:
    for row in csv.reader(inputfile):
        batterie.append(float(row[0]))

plt.plot(batterie, color = 'crimson', lw = 1.5)
plt.ylabel("Ladung / Wattstunden")
plt.title('Batterieladung')

verbrauch = -1 * np.diff(batterie) # np.diff berechnet die Ableitung, also die Steigung
plt.figure()
plt.plot(verbrauch, color = 'crimson', lw = 1.5)
plt.ylabel("Verbrauch / Wattstunden pro Meter")
plt.xlabel("Strecke / Meter")
plt.title('Batterieverbrauch')
```



Im direkten Vergleich sehen wir, wie aussagekräftig der Verbrauch im Vergleich zur Ladung ist. In der Verbrauchskurve können wir direkt ablesen, dass der Verbrauch meistens etwa 0.12 Wattstunden pro Meter (= 0.12 Kilowattstunden pro Kilometer) ist. Außerdem sehen wir sehr genau an welchen Stellen der Verbrauch erhöht, bzw. gering ist. Der höchste Verbrauch auf dieser Teststrecke lag bei über 0.30 Wattstunden pro Meter.

Numerisches Integral

Die Beziehung, in der in diesem Beispiel Ladung und Verbrauch der Batterie stehen, entspricht mathematisch der von **Integral** und **Differential**. Erstes gibt einen **Bestand** an (Engl. "stock"), hier den Bestand an bestimmten Punkten der Wegstrecke, zweiteres bezeichnet eine **Veränderung** (Engl. "flow"), bzw. eine **Veränderungsrate**, über diese Punkte der Wegstrecke.

So wie der Verbrauch aus der Information zur Ladung an verschiedenen Punkten der Wegstrecke errechnet werden kann, so kann auch aus dem Verbrauch zurück auf die Ladung geschlossen werden. Dazu wird wieder integriert. Konzeptionell ist dies in diesem Fall allerdings etwas Anderes, als das Integral, das beim Solarpanel benutzt wurde. Beim Solarpanel haben wir die **Fläche unter der Kurve** errechnet. Es handelt sich da um ein so genanntes **bestimmtes Integral**, das als Lösung eine **Zahl** liefert (z.B. die Gesamtstromproduktion).

Hier hatten wir allerdings die momentane Ladung zu jedem Zeitpunkt gegeben. Wir suchen hier also keine Fläche, sondern die so genannte **Stammfunktion** des Verbrauchs, also die Ladung. Es handelt sich hierbei um ein **unbestimmtes Integral**, das als Lösung eine **Funktion** hat.

Das Paket `numpy` enthält einen Befehl, der die Stammfunktion einer Funktion über eine Methode berechnet, die sich "kumulative Summe" nennt. Vereinfacht gesagt, wird dabei zu jedem Zeitschritt errechnet, wie groß der gesamte Verbrauch zu allen vorhergehenden Zeitschritten war. Der Befehl dazu lautet `cumsum` und liefert als Ergebnis wieder eine Liste. Versuchen wir damit aus dem Verbrauch zurück auf die Ladung zu rechnen:

```
In [16]: import numpy as np
import csv
import matplotlib.pyplot as plt
%matplotlib inline

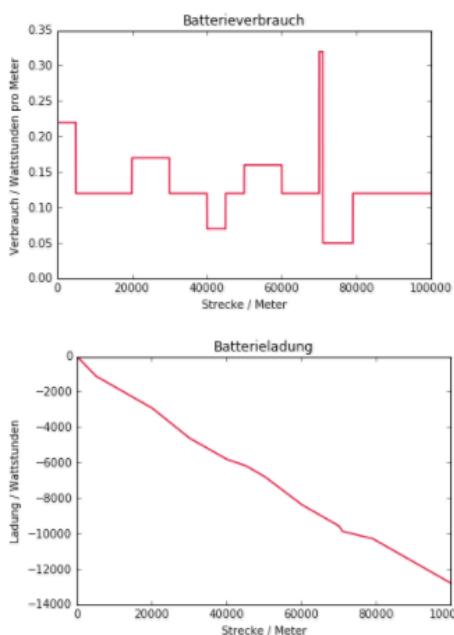
batterie = []
with open('batterie.txt') as inputfile:
    for row in csv.reader(inputfile):
        batterie.append(float(row[0]))

# 1. die Ladung
# plt.plot(batterie, color = 'crimson', lw = 1.5)
# plt.ylabel("Ladung / Wattstunden")
# plt.title('Batterieladung')

# 2. der Verbrauch
verbrauch = -1 * np.diff(batterie)
plt.figure()
plt.plot(verbrauch, color = 'crimson', lw = 1.5)
plt.ylabel("Verbrauch / Wattstunden pro Meter")
plt.xlabel("Strecke / Meter")
plt.title('Batterieverbrauch')

# 3. und wieder die Ladung
batterie2 = np.cumsum(-verbrauch) # np.cumsum berechnet die kumulative Summe, um die Stammfunktion zu bestimmen
plt.figure()
plt.plot(batterie2, color = 'crimson', lw = 1.5)
plt.ylabel("Ladung / Wattstunden")
plt.xlabel("Strecke / Meter")
plt.title('Batterieladung')
```

Out[16]: <matplotlib.text.Text at 0xcc57ba8>



Ausgehend vom Verbrauch ist es uns hier also gelungen, den Ladestand der Batterie an den einzelnen Punkten der Wegstrecke zu rekonstruieren.

Eine wichtige Information ist uns dabei allerdings verloren gegangen: wenn wir nur den Verbrauch in Betracht ziehen, können wir nicht sagen, wie viel Ladung zu Beginn in der Batterie war. In der vorliegenden Berechnung geht unser Integral davon aus, dass die Batterie am Anfang der Wegstrecke die Ladung 0 hatte, womit die Ladung insgesamt negativ wird.

Dies ist kein grundsätzlicher Fehler, sondern eine intrinsische Eigenschaft der Integralrechnung: bei Berechnung eines unbestimmten Integrals, ist das Ergebnis immer nur bis auf eine Konstante korrekt. Diese Konstante nennt man in der Mathematik "Integrationskonstante". Es ist zu beachten, dass es eine solche Konstante gibt und ein unbestimmtes Integral zwar immer die korrekte Funktion liefert, der Startwert dieser Funktion aber von Null verscheiden kann.

Zusammenfassung

Einlesen von Daten

Um Daten aus einer Datei einzulesen, kann unter anderem das Paket `csv` wie folgt benutzt werden:

```
In [17]: datenliste = []
with open('daten.txt') as inputfile:
    for row in csv.reader(inputfile):
        datenliste.append(float(row[0]))
```

Die entstehende Liste kann dann beispielsweise innerhalb einer For-Schleife weiterverarbeitet werden.

Auswählen von Listenteilen

Um aus einer Liste einen Teil der Einträge auszuwählen, steht der Befehl `listename[anfang:ende]` zur Verfügung. Um beispielsweise die ersten 10 Elemente der Liste `solarstrom` auszuwählen, schreibt man `solarstrom[0:10]` (Vorsicht: man erhält damit die Einträge einschließlich des Elements `solarstrom[0]`, aber ausschließlich des Elements `solarstrom[10]`).

Numerisches Integrieren

Um die Fläche unter einer Kurve zu berechnen, also ein **bestimmtes Integral**, steht der Befehl `sum` zur Verfügung. Das Ergebnis eines bestimmten Integrals ist eine **Zahl**. Wichtig dabei ist es, die richtigen Einheiten zu verwenden, bzw. in geometrischer Interpretation, zusätzlich zur Höhe der Rechtecke, die aufsummiert werden, auch die Breite korrekt zu bestimmen (also ob ein Rechteck eine Wattstunde breit ist, oder nur 1/60 Wattstunde).

Zusätzlich zum bestimmten Integral kann auch das **unbestimmte Integral** berechnet werden. Das unbestimmte Integral liefert als Lösung immer eine **Funktion**, also in unserem Fall eine Liste. Zum Berechnen eines unbestimmten Integrals, also einer Stammfunktion, steht der Befehl `cumsum` aus dem Paket `numpy` zur Verfügung. Die resultierende Stammfunktion ist aber immer nur bis auf eine sogenannte Integrationskonstante korrekt.

Numerisches Differenzieren

Um die Ableitung (also die Steigung in jedem Punkt) einer Funktion oder Zeitreihe zu berechnen, um also zu *differenzieren*, steht der Befehl `diff` aus dem Paket `numpy` zur Verfügung. So wie im vorliegenden Beispiel aus der Batterieladung der Verbrauch errechnet wurde, so lässt sich analog zum Beispiel auch ein zurückgelegter Weg in Geschwindigkeit oder eine Geschwindigkeit in Beschleunigung umrechnen.

Kapitel 8 – Diskret und kontinuierlich - Die Zinsrechnung

Im vorhergehenden Kapitel haben wir mit `sum` und `cumsum` zwei Python-Befehle kennengelernt, die für die Differential- und Integralrechnung verwendet werden können. In vielen Fällen lässt sich eine Integration aber auch einfach mithilfe einer For-Schleife durchführen.

Sehen wir uns dazu kurz das einfache Beispiel eines Hotels an, das vom Anfang der Saison an (Null Gäste) jeden Tag am Abend Bilanz über seine ankommenden und abreisenden Gäste zieht. Das An- und Abreisen stellt dabei die Veränderung (*flow*) dar, aus der wir im Folgenden durch Integration den jeweiligen *Tagesbestand* an Gästen (*stock*) errechnen.

```
In [1]: import random
import matplotlib.pyplot as plt
%matplotlib inline

# die Zeichenfläche wird hier genauer als in den bisherigen Beispielen definiert, u.a. in Bezug auf ihre Größe
# und die Lage der beiden plots zueinander
fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(2,1,1)
ax2 = fig.add_subplot(2,1,2)

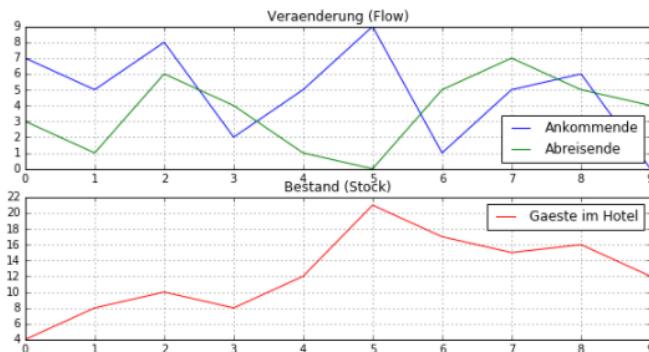
# Simulationszeit (der erste Tag wird als nullter Tag gerechnet)
T = range(10)

# Listen mit Gästefluktuation
ankommende = [7, 5, 8, 2, 5, 9, 1, 5, 6, 0]
abreisende = [3, 1, 6, 4, 1, 0, 5, 7, 5, 4]

G = [0] # Liste mit anfänglicher Zahl der Gäste im Hotel

# integrieren
for t in T:
    # Gäste am Tag t + ankommende Gäste - abreisende Gäste am Tag t
    G.append(G[t] + ankommende[t] - abreisende[t])

# Zeichnen
ax1.plot(T, ankommende, label = 'Ankommende')
ax1.plot(T, abreisende, label = 'Abreisende')
ax2.plot(T, G[1:], 'r', label = 'Gäste im Hotel')
# Ausschmücken der Plots
ax1.set_title('Veränderung (Flow)')
ax2.set_title('Bestand (Stock)')
ax1.legend(loc = 'best')
ax2.legend(loc = 'best')
ax1.grid()
ax2.grid()
```



Natürlich kommen Gäste unter Umständen auch in der Nacht in ein Hotel, oder reisen spätabends ab. Wir können uns also vorstellen, dass, wenn die Bilanz abends erstellt wird, die Tourismusbehörde aber am Vormittag die Hotelauslastung erhebt, die Informationen nicht ganz korrekt sein werden. Es könnte also wünschenswert sein, öfters am Tag zu bilanzieren.

Was hier freilich recht einfach klingt, hat weitreichende Folgen. Es berührt die komplexe Problematik des Unterschieds zwischen **diskreter** und **kontinuierlicher** Berechnung.

Wir haben bisher in diesem Skriptum nur Entwicklungen kennengelernt, die in **diskreten** Zeitschritten voranschreiten, die also zum Beispiel einmal im Jahr die Größe einer Frosch- oder einer Kaninchenpopulation in Betracht ziehen oder eben einmal am Tag die Zahl der Gäste in einem Hotel überprüft. Dies ist durchaus realistisch, da sich ja Tierpopulationen durch die Geburt einzelner Individuen, also in bestimmten (wenn auch eher unregelmäßigen) Zeitabständen vermehren und auch die Gäste zu ganz bestimmten Zeiten im Hotel eintreffen.

Viele Entwicklungen in der Natur scheinen demgegenüber aber **kontinuierlich** abzulaufen. Von der Bewegung der Planeten um die Sonne wird zum Beispiel angenommen, dass sie fortlaufend ist, also nicht in einzelnen kleinen Zeitsprüngen erfolgt, und auch das Volllaufen-Lassen einer Badewanne zum Beispiel stellen wir uns eher als Kontinuum vor. Die Mathematik hat zur Modellierung solcher **kontinuierlicher** Bewegungen oder Entwicklungen eine eigene abstrakte Methodik entwickelt, die auch in den Systemwissenschaften eine wichtige Rolle spielt.

Um diese Methodik gut zu verstehen, werden wir uns im Folgenden eine einfache und sehr gebräuchliche Alltags-Berechnung genauer ansehen:

Die Zinsesrechnung

Bankzinsen werden gewöhnlich einmal im Jahr, das heißt, in einzelnen, sprich **diskreten** Zeitschritten verrechnet. Würde es etwas ändern, wenn Banken öfters als nur einmal im Jahr abrechnen würden?

Nehmen wir an, ein Bankkonto wird mit einer Einlage von 100 Euro eröffnet und sodann, ohne weitere Einlage, jährlich von der Bank mit 3% verzinst. Wir haben die Art des Wachstums, das sich daraus ergibt, im vorigen Kapitel bereits kennengelernt. Es handelt sich um - wenn auch hier eher geringes - **exponentielles Wachstum**.

Wir schreiben B_t für die Einlage zum Zeitpunkt t , B_{t+1} für die Einlage nach einem Jahr (in dem Fall nach einem Zeitschritt), Z für den Zuwachs durch die Verzinsung in einem Zeitschritt, sowie γ für die **Wachstumsrate** der Einlage ($\gamma = 0.03$). Wir können dies als Formel wie folgt anschreiben:

$$B_{t+1} = B_t + Z$$

mit dem Zusammenhang:

$$Z = \gamma * B_t$$

somit:

$$B_{t+1} = B_t + \gamma * B_t$$

Für das konkrete Beispiel ergibt sich damit

$$B_{t+1} = 100 + 0.03 * 100$$

Mit Python berechnet:

```
In [2]: 100 + 0.03 * 100
Out[2]: 103.0
```

Um die Berechnung auch für andere Einlagen und Zinssätze einfach durchführen zu können, definieren wir folgende Variablen:

```
In [2]: B0 = 100          # Einlage zum Zeitpunkt 0
# B1 = ?            # Einlage zum Zeitpunkt 1 (d.h. nach einem Jahr)
gamma = 0.03         # Wachstumsrate (Zins durch 100)

# Berechnung
B1 = B0 + gamma * B0
# Ausgabe
print(B1)
```

103.0

Wenn wir nun die Einlagenhöhe (bei gleichbleibender Verzinsung) nicht nur nach einem Jahr, sondern für die nächsten 5 Jahre errechnen wollen, so können wir eine **Schleife** (engl. loop) verwenden, die unsere Berechnung iteriert, d.h. sie wiederholt ausführt und in jeder Schleife ein Ergebnis ausgibt. Wir benötigen dazu ein paar weitere Variablen: T für die Gesamlaufzeit und B_t und B_{next} als Zwischenspeicher, um die Werte zu errechnen und auszugeben.

```
In [3]: B0 = 100          # Einlage zum Zeitpunkt 0
Bt = 0           # Einlage zum Zeitpunkt t (Zwischenspeichervariable)
# Bnext = ?        # Einlage zum je nächsten Zeitpunkt (d.h. nach je einem Jahr)
gamma = 0.03         # Wachstumsrate

# definiere die Gesamlaufzeit (der Python-Befehl range erzeugt eine Liste der Form [0, 1, 2, 3, 4])
T = range(5)

# Übergib die Anfangseinlage für die Berechnung an die Variable Bt
Bt = B0
# definiere eine Schleife (lies: "für jedes t in T", d.h. Python weiß selbst, dass t jeweils den nächsten Wert in T meint)
for t in T:
    # Berechnung
    Bnext = Bt + gamma * Bt
    # Ausgabe
    print(Bnext)
    # Übergib den neuen Wert für die Berechnung an die Variable Bt
    Bt = Bnext
```

103.0
106.09
109.2727
112.550881
115.92740743

Wir können dieses Ergebnis "verschönern", indem wir es mit Text versehen und die Jahre mitzählen lassen.

```
In [4]: # B0 = 100      # Einlage zum Zeitpunkt 0
# Bt = 0          # Einlage zum Zeitpunkt t
# Bnext = ?       # Einlage zum je nächsten Zeitpunkt (d.h. nach je einem Jahr)
gamma = 0.03      # Wachstumsrate

# definiere die Gesamtaufzeit
T = range(5)

# Über gib die Anfangseinlage für die Berechnung an die Variable Bt
Bt = B0
# definiere eine Schleife
for t in T:
    # Berechnung
    Bnext = Bt + gamma * Bt
    # Ausgabe (%s steht als Platzhalter für den Integer-Wert, der am Ende der Zeile mit 't+1' festgelegt wird
    # beachte, dass der Computer solche Listen mit 0 beginnen lässt), und %.2f steht für einen Float-Wert,
    # der auf 2 Nachkommastellen (0.2) begrenzt ist und am Ende der Zeile mit 'Bnext' festgelegt wird
    print("Nach dem %-ten Jahr beträgt die Einlage Euro %.2f" %(t+1, Bnext))
    # Über gib den neuen Wert für die Berechnung an die Variable Bt
    Bt = Bnext

Nach dem 1-ten Jahr beträgt die Einlage Euro 103.00
Nach dem 2-ten Jahr beträgt die Einlage Euro 106.09
Nach dem 3-ten Jahr beträgt die Einlage Euro 109.27
Nach dem 4-ten Jahr beträgt die Einlage Euro 112.55
Nach dem 5-ten Jahr beträgt die Einlage Euro 115.93
```

Plotten

Wir können dieses Ergebnis auch graphisch darstellen (plotten). Dazu schreiben wir die zu berechnenden Zwischenergebnisse in eine Liste B und verwenden das Python-Paket 'matplotlib', das wir allerdings zunächst importieren müssen.

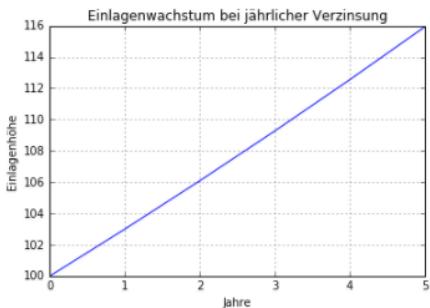
```
In [5]: # importiere das Paket (oder Modul) matplotlib unter der (beliebig definierbaren) Abkürzung plt
import matplotlib.pyplot as plt
# zeige die Plots im Notebook (sprich: auf dieser Seite)
%matplotlib inline

B0 = 100      # Einlage zum Zeitpunkt 0
B = []         # eine leere Liste, in die die berechneten Einlagen zum Zeitpunkt t geschrieben werden
gamma = 0.03   # Wachstumsrate

# definiere die Gesamtaufzeit
T = range(5) # 50

# Über gib die Anfangseinlage für die Berechnung an die Liste B
B = [B0]
# definiere eine Schleife
for t in T:
    # Berechnung, deren Ergebnis mit dem Python-Befehl 'append' gleich in die Liste B eingeschrieben wird.
    # t wird dabei als Index für die jeweils schon vorhandenen Listeneinträge verwendet
    B.append(B[t] + gamma * B[t])

plt.plot(B)
# Beschriftung der x-Achse
plt.xlabel('Jahre')
# Beschriftung der y-Achse. Unicode zur Darstellung von Umlauten
plt.ylabel(u'Einlagenhöhe')
# Titel des Plots
plt.title(u'Einlagenwachstum bei jährlicher Verzinsung')
# zeichne zusätzliche Koordinaten
plt.grid()
```



Die gezeichnete Kurve für unser Einlagenwachstum sieht auf den ersten Blick eher linear aus. Wenn wir freilich unseren Beobachtungszeitraum von 5 auf 50 Jahre ausdehnen, sehen wir deutlich, dass es sich hier um exponentielles und nicht nur lineares Wachstum handelt (Versuchen sie dies im obigen Code).

Der Übergang von der Differenzen- zur Differentialgleichung

Wie gesagt, verzinsen Banken Kontoeinlagen im Normalfall einmal pro Geschäftsjahr. Die oben durchgeführten Zinsberechnungen wurden deshalb als **Differenzengleichungen** durchgeführt, mit einer Differenz von einem Jahr.

Wie würde sich das Wachstum der Einlagenhöhe gestalten, wenn die Verzinsung öfters als nur einmal jährlich stattfinden würde?

Betrachten wir das obige Beispiel bei zwei-maliger Verzinsung pro Jahr, wobei die Einlage in jedem der beiden Halbjahre (also in der nun betrachteten Differenz) mit 1.5% verzinst wird.

```
In [6]: B0 = 100      # Einlage zum Zeitpunkt 0
Bt = 0          # Einlage zum Zeitpunkt t
# Bnext = ?     # Einlage zum je nächsten Zeitpunkt (d.h. nach je einem Jahr)
gamma = 0.03    # Wachstumsrate (Zins durch 100)

# definiere die Gesamlaufzeit
T = range(5)

# Übergib die Anfangseinlage für die Berechnung an die Variable Bt
Bt = B0
# definiere eine Schleife
for t in T:
    # erste Verzinsung: Berechnung mit halber Zinsrate
    Bnext = Bt + (gamma/2.) * Bt
    # Übergib den neuen Wert für die Berechnung an die Variable Bt
    Bt = Bnext
    # zweite Verzinsung: neuerliche Berechnung mit halber Zinsrate
    Bnext = Bt + (gamma/2.) * Bt
    # Ausgabe
    print('Nach dem %s-ten Jahr beträgt, bei zwei-maliger Verzinsung pro Jahr, die Einlage %0.2f' %(t+1, Bnext))
    # Übergib den neuen Wert für die Berechnung an die Variable Bt
    Bt = Bnext

Nach dem 1-ten Jahr beträgt, bei zwei-maliger Verzinsung pro Jahr, die Einlage 103.02
Nach dem 2-ten Jahr beträgt, bei zwei-maliger Verzinsung pro Jahr, die Einlage 106.14
Nach dem 3-ten Jahr beträgt, bei zwei-maliger Verzinsung pro Jahr, die Einlage 109.34
Nach dem 4-ten Jahr beträgt, bei zwei-maliger Verzinsung pro Jahr, die Einlage 112.65
Nach dem 5-ten Jahr beträgt, bei zwei-maliger Verzinsung pro Jahr, die Einlage 116.05
```

Wir sehen, es macht einen zwar kleinen aber doch entscheidenden Unterschied, ob eine Konto-Einlage einmal oder zweimal pro Jahr verzinst wird. Oder anders gesagt, es kann einen Unterschied machen in welchem Zeitabstand, d.h. mit welcher **Differenz** eine Entwicklung betrachtet wird.

Um uns die Bedeutung und die Konsequenz dieses Umstandes deutlich zu machen, sehen wir uns das Verzinsungsbeispiel nun mit einer Anfangseinlage von 1 Euro, einer Zinsrate von 100% und einer Laufzeit von nur einem Jahr an, wobei wir die Verzinsungsabstände, also die **Differenz** sukzessive verkleinern und die Ergebnisse plotten.

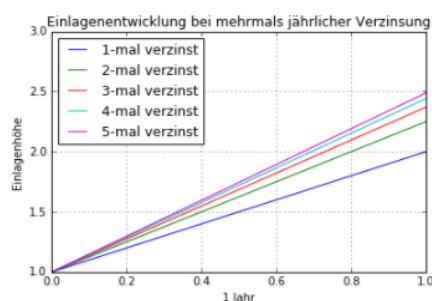
```
In [7]: B0 = 1      # Einlage zum Zeitpunkt 0
B = []        # eine leere Liste, in die die berechneten Einlagen zum Zeitpunkt t geschrieben werden
gamma = 1.     # Wachstumsrate

# definiere die Gesamlaufzeit
T = range(1)
# definiere eine Liste mit Integerwerten für die Verkleinerungsbruchzahlen
# um Division durch 0 zu vermeiden, stelle sicher, dass die Liste mit 1 beginnt
N = range(1, 6)

for n in N:
    # Übergib die Anfangseinlage für die Berechnung an die leere Liste B
    B = [B0]
    for t in T:
        # über gib den aktuellen Listeneintrag in B an die Hilfsvariable Bt
        Bt = B[t]
        # verzins so oft wie n vorgibt
        for nn in range(n):
            # Python 2.x dividiert nur mit reeller Zahl im Nenner korrekt, daher float(nn)
            Bnext = Bt + gamma/float(nn) * Bt
            Bt = Bnext
        B.append(Bnext)
    # zeichne n-mal in den selben plot
    plt.plot(B, label = '%s-mal verzinst' %n)

# Limit für die y-Achse
plt.ylim(1, 3)
# Beschriftung der x-Achse
plt.xlabel('1 Jahr')
# Beschriftung der y-Achse
plt.ylabel('Einlagenhöhe')
# Titel des Plots
plt.title('Einlagenentwicklung bei mehrmals jährlicher Verzinsung')
# zeichne zusätzliche Koordinaten
plt.grid()
# Legende
plt.legend(loc='best')
```

Out[7]: <matplotlib.legend.Legend at 0xacbd908>



Wir sehen, die Einlagenhöhe wächst mit der Zahl der jährlichen Verzinsungen. Sie scheint aber nicht gleichmäßig zu wachsen. Vielmehr fügt jede weitere Verzinsung pro Jahr einen kleiner werdenden Zinsgewinn zur Einlage hinzu. Die Zinsgewinne scheinen **assymptotisch einem Limit** zu streben.

Um dieses Limit deutlich zu sehen, führen wir die obige Berechnung noch einmal für viele und sehr klein werdende Verzinsungsabstände (d.h. sehr klein werdende Differenzen) durch und plotten die Ergebnisse nicht in der Zeit sondern in Bezug auf die Zahl der Verzinsungen.

```
In [8]: import numpy as np

B0 = 1          # Einlage zum Zeitpunkt 0
gamma = 1.       # Wachstumsrate
E = []
BB = []

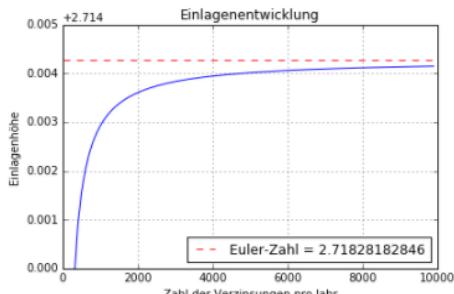
# erzeuge Liste von 1 bis 10000 in 100er-Abständen
N = np.arange(1, 10000, 100)

for n in N:
    Bt = 1
    # da nur 1 Jahr betrachtet wird, verzichten wir hier auf die 'for t in T'-Schleife
    # verzins so oft wie n vorgibt
    for nn in range(n):
        Bnext = Bt + gamma/float(nn) * Bt
        Bt = Bnext
    BB.append(Bnext)
    # schreibe die Eulerzahl wiederholt in Liste (für die rote unterbrochene Linie im Plot)
    E.append(np.e)

# plote Einlagenhöhe gegen die Zahl der Verzinsungen/Jahr
plt.plot(N, BB)
# plote die Eulerzahl, als Limit gegen das die Einlagenhöhe strebt (mit Label für Legende)
plt.plot(N, E, '---', label='Euler-Zahl = 2.71828182846')

# Limitierung der y-Achse
plt.ylim(2.714, 2.719)
# Beschriftung der x-Achse
plt.xlabel('Zahl der Verzinsungen pro Jahr')
# Beschriftung der y-Achse
plt.ylabel('Einlagenhöhe')
# Titel des Plots
plt.title('Einlagenentwicklung')
# zeichne zusätzliche Koordinaten
plt.grid()
# füge eine Legende hinzu
plt.legend(loc = 'lower right')
```

Out[8]: <matplotlib.legend.Legend at 0xb00f908>



Das Limit, dem die sukzessive Verkleinerung der Verzinsungsabstände, also der betrachteten **Differenzen** in diesem Beispiel zustrebt, ist die berühmte **Eulerzahl**. Sie steht damit als Grenzwert für eine **Differenzenrechnung**, die mit einer tendenziell gegen 0 verkleinerten Differenz rechnet.

Eine (real existierende) Differenz wird oftmals mit Δ abgekürzt. Der sogenannte Differenzenquotient $\frac{\Delta x}{\Delta t}$, der die mittlere Änderungsrate einer Entwicklung angibt, wird mit gegen 0 strebendem Δ als **Differentialquotient**, als "momentane Änderungsrate" bezeichnet.

$$\frac{\Delta x}{\Delta t} \underset{\lim \Delta \rightarrow 0}{=} \frac{dx}{dt}$$

Das sich daraus ergebende **Differential** ist ein mathematisches Abstraktum, das in der Empirie in dieser Weise nicht vorkommt, das sich aber gut zur Berechnung von Dynamiken eignet. Einfaches **exponentielles Wachstum** zum Beispiel, wie wir es bereits in Kapitel 2 kennengelernt haben, lässt sich mit der **Exponentialfunktion** (d.h. mithilfe der Euler-Zahl) sehr gut berechnen. Während die Differentialgleichung dafür lautet:

$$\frac{dN}{dt} = \gamma N$$

sieht die exakte, sprich rein mathematische Lösung dafür wie folgt aus:

$$N_t = N_0 e^{\gamma t}$$

Mituntere schreibt man dafür auch $N_t = N_0 * \exp(\gamma t)$, wobei e und exp für die Euler-Zahl stehen. Man spricht diesbezüglich von einer **analytischen Lösung**.

Der große Nachteil dieser analytischen Lösung ist allerdings, dass sie nur für eher einfache Dynamiken und nur für sehr wenige Systeme bekannt ist. Die meisten gekoppelten Differentialgleichungssysteme, wie wir sie im nächsten Kapitel kennenlernen werden, können nicht analytisch gelöst werden, sondern müssen **simuliert** werden, sprich am Computer nachgestellt werden, wenn man ihr Verhalten erkunden will.

Merke: Computer, als digitale Maschinen, können immer nur mit Differenzen, bestenfalls mit **angenäherten** Differentialen (und das heißt auch: mit angänherter Euler-Zahl) rechnen!

Zusammenfassung

Diskret / kontinuierlich

Während empirische Untersuchungen - etwa das Zählen von Tierpopulationen - zu bestimmten Zeitpunkten stattfinden und sodann zu bestimmten Zeitabständen wiederholt werden - damit also **diskrete** Informationen zur Entwicklung einer Population liefern -, werden viele physikalische Entwicklungen - etwa das Umlaufen der Planeten um die Sonne oder das Diffundieren von Gasen - als **kontinuierlich** angenommen.

Mathematisch werden **diskrete** Entwicklungen mit **Differenzengleichungen** und **kontinuierliche** Entwicklungen mit **Differentialgleichungen** erfasst.

Unterschied Differenz / Differential

Die Verkleinerung von Untersuchungsabständen (oder hier der Abstände zwischen Zinsberechnungen) bedeutet eine Annäherung an kontinuierlichen Informationsfluss, also dichtere Untersuchungsergebnisse. Die Mathematik nimmt zum Modellieren kontinuierlicher Entwicklungen eine - theoretisch - ins Unendliche verkleinerbare Differenz Δ an und bezeichnet dies als Differential d .

Differenzen- / Differentialquotient

Während der Differenzenquotient $\frac{\Delta x}{\Delta t}$ die **mittlere** Änderungsrate einer Entwicklung angibt, bezeichnet der **Differentialquotient** eine (theoretisch angenommene) **momentane** Änderungsrate.

Euler-Zahl

Die Euler-Zahl $e = 2.7182818$ ergibt sich (hier) als **Limit** einer als kontinuierlich angenommenen 100%-Verzinsung einer 1-Euro-Einlage. Sie bildet die Basis der **Exponentialfunktion** und wird in der Analysis, insbesondere in der **Differential- und Integralrechnung** zur Berechnung kontinuierlicher Entwicklungen oder Bewegungen herangezogen.

Analytische Lösung

Analytisch wird die Lösung einer Differentialgleichung genannt, wenn sie mithilfe der Euler-Zahl (oder der Exponentialfunktion) gefunden wird. Komplexere Dynamiken und die meisten Differentialgleichungssysteme kennen allerdings keine analytische Lösung. Sie müssen am Computer **simuliert** werden, um ihr Verhalten zu erkunden.

Kapitel 9 – Gekoppelte Differentialgleichungen – Räuber-Beute-Systeme

Wir haben in Kapitel 3 dieses Skriptums bereits die Entwicklung zweier Tierpopulationen betrachtet, deren Wachstum voneinander abhängig war. Solche Abhängigkeiten und Beeinflussungen unterschiedlicher Dynamiken sind gewissermaßen der Normalfall in den Systemen, die für die Systemwissenschaften interessant sind. Der Systembegriff geht ja eben davon aus, dass *interagierende*, also sich wechselseitig beeinflussende Dynamiken in ihrem Zusammenwirken etwas generieren, das ohne dieses Zusammenwirken nicht, oder zumindest nicht so, beobachtet werden kann.

In Bezug auf diese sich wechselseitig beeinflussenden Dynamiken spricht man von *gekoppelten* Dynamiken. Und die Mathematik kennt zu ihrer Analyse die Methode der **gekoppelten Differentialgleichungen**. Das systemwissenschaftliche Standardbeispiel für solche gekoppelten Differentialgleichungssysteme sind **Räuber-Beute-Systeme**, für die es in der Regel allerdings keinen analytischen (rein mathematischen) Lösungsweg gibt. Im Folgenden wollen wir deshalb ein historisches Beispiel eines solchen Räuber-Beute-Systems mit Hilfe von Python simulieren.

Pelztier-Statistiken

Zu Beginn des 20. Jahrhunderts wurde begonnen, Statistiken zu einer Vielzahl von Phänomenen anzulegen. Unter anderem wurde auch dokumentiert, wieviele Felle von kanadischen Pelztier-Jägern auf ihren Streifzügen im hohen Norden Amerikas erjagt und sodann in der Hudson's Bay Company, einer großen Handels- und Vertriebsgesellschaft, zum Verkauf abgegeben wurden. Für die Jahre 1909–1932 wurden für Hasen- und Luchsfelle die folgenden Zahlen dokumentiert.

Jahr	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932
Hasenfelle	25	50	55	75	70	55	30	20	15	15	20	35	60	80	85	60	30	20	10	5	5	10	30	80
Luchsfelle	2	4	10	14	19	14	8	9	2	1	1	2	4	4	8	7	9	7	4	3	2	3	3	5

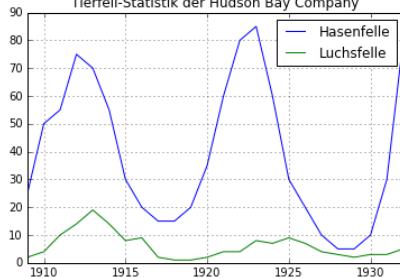
Wir können diese **Zeitreihen** wie folgt mit Python darstellen:

```
In [1]: # importiere das Python-Modul matplotlib unter der (beliebig definierbaren) Abkürzung plt
import matplotlib.pyplot as plt
# zeige Grafiken im Notebook (sprich: auf dieser Seite)
%matplotlib inline

# definiere Listen, die die Daten für die Zeitreihen enthalten
years = range(1909, 1933)
hare = [25, 50, 55, 75, 70, 55, 30, 20, 15, 15, 20, 35, 60, 80, 85, 60, 30, 20, 10, 5, 5, 10, 30, 80]
lynx = [2, 4, 10, 14, 19, 14, 8, 9, 2, 1, 1, 2, 4, 4, 8, 7, 9, 7, 4, 3, 2, 3, 3, 5]

# zeichne die Zeitreihen in einen Graph
plt.plot(years, hare, label = 'Hasenfelle')
plt.plot(years, lynx, label = 'Luchsfelle')
plt.title('Tierfell-Statistik der Hudson Bay Company')
plt.grid()
plt.xlim(1909, 1932)
plt.legend(loc='best')
```

Out[1]: <matplotlib.legend.Legend at 0xad52dd8>



Der Plot deutet an, dass die Zahlen zu den abgegebenen Fellen über die Jahre oszillieren. Mit etwas Phantasie lässt sich auch feststellen, dass die Zahlen der Hasen- und Luchsfelle irgendwie **gemeinsam** zu oszillieren scheinen, mit vielleicht kleiner zeitlicher Verzögerung der Luchsfelle.

Dies gibt Anlass zu der Annahme, dass das Vorkommen von Hasen und Luchsen im hohen Norden Amerikas miteinander in Beziehung steht. Wir haben eine solche Beziehung bereits kennengelernt: Luchse jagen Hasen als Beute. Wenn es in einem Jahr viele Hasen gibt, so finden die Luchse reichlich Beute und können sich ob der reichlichen Nahrung gut vermehren. Dadurch steigt die Zahl der Luchse in den Folgejahren, was allerdings dazu führt, dass mehr Hasen zu Beute werden und damit die Hasenpopulation sinkt. Als Folge finden die Luchse wieder weniger Beute und damit schlechtere Lebensbedingungen, was wieder den Hasen zugute kommt, deren Zahl wieder anwachsen kann, usw.

Die gekoppelten Schwingungen dieser Zeitreihen lassen sich in einem einfachen mathematischen Modell mithilfe zweier gekoppelter Differentialgleichungen darstellen:

$$\frac{dH}{dt} = a * H - b * H * L$$
$$\frac{dL}{dt} = c * L * H - d * L$$

wobei H und L für den jeweiligen Bestand an Hasen, bzw. Luchsen stehen, und a und c als Geburtsraten, sowie b und d als Sterberaten für Hasen und Luchse vorgestellt werden können. Die Sterberate b für Hasen, sowie die Geburtsrate c für Luchse hängt dabei zusätzlich vom jeweiligen Bestand der je anderen Population ab.

Merke: Geburtsraten und Sterberaten sind **Veränderungsraten**. Die ihnen entsprechenden Dynamiken (d.h. die Veränderungen) werden mathematisch in diskreter Zeit als **Differenzengleichungen** (gemessen in Bestandseinheit pro Zeitintervall) und in kontinuierlicher Zeit als **Differentialgleichungen** dargestellt (gemessen in Bestandseinheit pro gegen Null verkleinertem Zeitintervall, also pro abstrahiertem **Zeitpunkt**).

Die mathematische Formulierung dieses Systems wurde, unabhängig voneinander, von zwei Wissenschaftern vorgeschlagen, vom Chemiker Alfred J. Lotka und vom Mathematiker Vito Volterra. Das Differentialgleichungssystem wird deshalb zumeist als **Lotka-Volterra-System** bezeichnet.

Wir können dieses Differentialgleichungssystem wie folgt als Differenzengleichungssystem annähern

$$H_{t+1} = H_t + (a * H_t - b * H_t * L_t) * dt$$

$$L_{t+1} = L_t + (c * L_t * H_t - d * L_t) * dt$$

und mit Python simulieren:

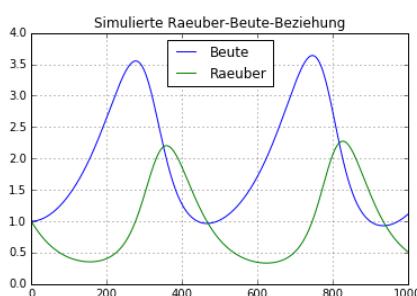
```
In [2]: import matplotlib.pyplot as plt
%matplotlib inline

a = 0.1 # Geburtsrate der Beutepopulation
b = 0.1 # Sterberate der Beutepopulation
c = 0.1 # Geburtsrate der Räuberpopulation
d = 0.2 # Sterberate der Räuberpopulation
T = range(1000) # betrachteter Zeitraum
H = [1.] # Liste mit Anfangswert für die Beutepopulation
L = [1.] # Liste mit Anfangswert für die Räuberpopulation
dt = 0.1 # Differenz pro Zeitschritt

# definiere eine Schleife
for t in T:
    # berechne H für den jeweiligen Zeitschritt und füge das Resultat jeweils am Ende der Liste hinzu
    H.append(H[t] + (a * H[t] - b * H[t] * L[t]) * dt)
    # berechne L für den jeweiligen Zeitschritt und füge das Resultat jeweils am Ende der Liste hinzu
    L.append(L[t] + (c * L[t] * H[t] - d * L[t]) * dt)

# zeichnen
plt.plot(T, H[:-1], label = 'Beute')
plt.plot(T, L[:-1], label = 'Raeuber')
plt.title('Simulierte Raeuber-Beute-Beziehung')
plt.grid()
plt.legend(loc='best')

Out[2]: <matplotlib.legend.Legend at 0xad69208>
```



Die Lotka-Volterra-Regeln

Wenn wir die obige Darstellung betrachten, erkennen wir eine der Besonderheiten, die als typisch für solche Systeme gelten und als **erste Lotka-Volterra-Regel** bezeichnet wird. Sie lautet:

- die Räuber- und die Beute-Populationen oszillieren periodisch und zueinander zeitlich versetzt. Die Räuber-Population läuft der Beute-Population zeitlich etwas hinterher.

Die **zweite Lotka-Volterra-Regel** lautet:

- die durchschnittlichen Größen der beiden Populationen bleiben über längere Zeiträume konstant, auch wenn die Maxima und Minima sehr unterschiedlich sind.

In unserer obigen Darstellung haben wir nur zwei Perioden - im Rahmen von 1000 Zeitschritten - berücksichtigt. Das lässt noch nicht viele Schlüsse über die Konstanz der Mittelwerte zu. Wir sollten dies aber leicht durch Erhöhung der Zahl der Zeitschritte testen können. Versuchen wir dies, indem wir im Folgenden die Simulationszeit auf 10000 erhöhen:

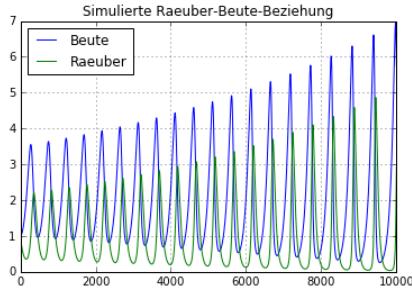
```
In [3]: import matplotlib.pyplot as plt
%matplotlib inline

a = 0.1 # Geburtsrate der Beutepopulation
b = 0.1 # Sterberate der Beutepopulation
c = 0.1 # Geburtsrate der Räuberpopulation
d = 0.2 # Sterberate der Räuberpopulation
T = range(10000) # betrachteter Zeitraum
H = [1.] # Liste mit Anfangswert für die Beutepopulation
L = [1.] # Liste mit Anfangswert für die Räuberpopulation
dt = 0.1 # Differenz pro Zeitschritt

# definiere eine Schleife
for t in T:
    # berechne H für den jeweiligen Zeitschritt und füge das Resultat jeweils am Ende der Liste hinzu
    H.append(H[t] + (a * H[t] - b * H[t] * L[t]) * dt)
    # berechne L für den jeweiligen Zeitschritt und füge das Resultat jeweils am Ende der Liste hinzu
    L.append(L[t] + (c * L[t] * H[t] - d * L[t]) * dt)

# zeichnen
plt.plot(T, H[:-1], label = 'Beute')
plt.plot(T, L[:-1], label = 'Raeuber')
plt.title('Simulierte Raeuber-Beute-Beziehung')
plt.grid()
plt.legend(loc='best')
```

Out[3]: <matplotlib.legend.Legend at 0xb1451d0>



Wir sehen, dass die Oszillation offenbar im Lauf der Zeit anwachsen, dass also diese Schwingungen nicht wirklich periodisch sind. Gleichzeitig heißt dies auch, dass die Mittelwerte der Populationsgrößen keineswegs über längere Zeiträume konstant sein können, dass also die zweite Lotka-Volterra-Regel nicht erfüllt ist.

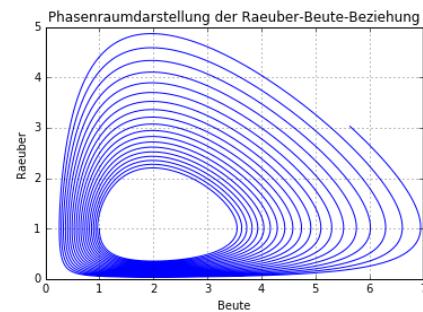
Besonders deutlich wird dies auch in der so genannten

Phasenraumdarstellung

In dieser Darstellungsweise wird die oben berechnete Entwicklung nicht in der Zeit, sondern im Verhältnis der beiden Populationen zueinander dargestellt, mit der Beute-Population auf der x-Achse und der Räuber-Population auf der y-Achse.

Wäre die Oszillation tatsächlich periodisch, und damit die Mittelwerte über längere Zeit stabil, so würde die Phasenraumdarstellung einen geschlossenen Zirkel beschreiben.

```
In [4]: plt.plot(H, L)
plt.title('Phasenraumdarstellung der Raeuber-Beute-Beziehung')
plt.xlabel('Beute')
plt.ylabel('Raeuber')
plt.grid()
```



Offenbar ist dies nicht der Fall. Die zweite Lotka-Volterra-Regel wird hier offensichtlich nicht erfüllt. Was ist schuld daran, dass unser Lotka-Volterra-Modell offenbar nicht wirklich genau periodisch oszilliert?

Die Ursache dieser Aperiodizität liegt in der gewählten **Integrationsmethode**, dem so genannten Euler-Verfahren (siehe dazu ausführlicher Kapitel 10 dieses Skriptums) und den verhältnismäßig großen Zeitschritten ($dt = 0.1$), die wir verwenden. Die Aperiodizität würde zwar geringer, wenn dt verkleinert würde (versuchen sie dies!), aber sie verschwindet mit dieser Methode nicht ganz.

Da Computer diskrete Maschinen sind, stoßen sie immer an Grenzen, wenn es darum geht, kontinuierliche Entwicklungen zu berechnen. Aus diesem Grund stellt Python auch elaboriertere Integrationsmethoden zur Verfügung, die es erlauben, solche Integrationen weitgehend "fehlerfrei" durchzuführen.

Das Prinzip einiger dieser Methoden wird in Kapitel 10 dieses Skriptums genauer erklärt.

Im Folgenden verwenden wir eine dieser Methoden, um das gekoppelte Räuber-Beute-Gleichungssystem genauer zu integrieren. Wir benötigen dazu das Python-Modul `scipy` (scientific Python) mit dem Modul `integrate`.

```
In [5]: # import Library
import numpy as np
from scipy import integrate

a = 0.1 # Geburtsrate der Beutepopulation
b = 0.1 # Sterberate der Beutepopulation
c = 0.1 # Geburtsrate der Räuberpopulation
d = 0.2 # Sterberate der Räuberpopulation
T = range(1000) # betrachteter Zeitraum

# bereite Zeichenfläche vor
fig = plt.figure(figsize=(15,5))
# definiere gleich zwei Zeichenflächen nebeneinander
# und formatiere die Zeichenflächen
fig.subplots_adjust(wspace = 0.5, hspace = 0.3)
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)

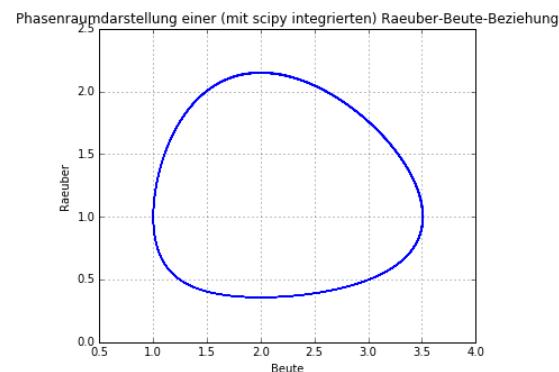
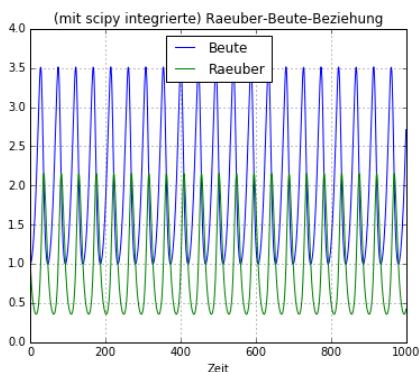
# definiere das System als Funktion
def Sys(X, t = 0):
    # X[0] = Beute und X[1] = Räuber
    return np.array([ a * X[0] - b * X[0] * X[1], c* X[1] * X[0] - d * X[1] ])

# Anfangswerte:
Sys0 = np.array([1, 1])

# integrieren
# type "help(integrate.odeint)" für mehr Information über integrate.odeint Inputs und Outputs.
X, infodict = integrate.odeint(Sys, Sys0, T, full_output=True)
# infodict['message'] # Integration erfolgreich
# transponiere den Output
x, y = X.T

# plot
ax1.plot(x, 'b-', label = 'Beute')
ax1.plot(y, 'g-', label = 'Raeuber')
ax1.set_title('(mit scipy integrierte) Raeuber-Beute-Beziehung')
ax1.set_xlabel('Zeit')
ax1.grid()
ax1.legend(loc='best')

ax2.plot(x, y)
ax2.set_title('Phasenraumdarstellung einer (mit scipy integrierten) Raeuber-Beute-Beziehung')
ax2.set_xlabel('Beute')
ax2.set_ylabel('Raeuber')
ax2.grid()
```



Wie deutlich zu erkennen ist, wächst die Entwicklung nun nicht mehr an. Das System scheint wirklich genau periodisch zu schwingen. In der Phasenraumdarstellung (rechts) bleibt der beschriebene Zirkel geschlossen. Damit ist auch die zweite Lotka-Volterra-Regel erfüllt. Die Mittelwerte bleiben über lange Zeiträume konstant, auch wenn sich die Maxima und Minima unterscheiden.

Sehen wir uns zuletzt noch kurz die **dritte Lotka-Volterra-Regel** an. Sie lautet:

- Werden Räuber- und Beute-Population gleichzeitig um den gleichen Prozentanteil dezimiert, so steigt der Mittelwert der Beutepopulation kurzfristig an, und der Mittelwert der Räuberpopulation sinkt kurzfristig ab.

Leider können wir dies nun nicht mit der soeben verwendeten Integrationsmethode `integrate` überprüfen, da wir dazu an einer bestimmten Stelle in die Berechnung intervenieren müssen. Dies ist mit `integrate` nicht (einfach) möglich. Wir können uns aber behelfen, indem wir im Folgenden das `dt`, also die Schrittweite unserer Integration auf 0.01 verkleinern und den Simulationszeitraum ausdehnen.

Wir verwenden dafür nun wieder eine Differenzengleichung zur Simulation des Systems und reduzieren beide Populationen mithilfe einer `if`-Abfrage nach dem Ende der ersten Periode (bei Zeitschritt 4660) auf 20%. Wir berechnen den Mittelwert dieser ersten Periode und vergleichen ihn mit dem Mittelwert gleich vieler Zeitschritte (also 4660) in der zweiten Periode.

```
In [6]: import matplotlib.pyplot as plt
%matplotlib inline

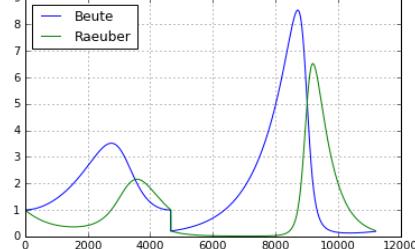
a = 0.1 # Geburtsrate der Beutepopulation
b = 0.1 # Sterberate der Beutepopulation
c = 0.1 # Geburtsrate der Räuberpopulation
d = 0.2 # Sterberate der Räuberpopulation
T = range(11210) # betrachteter Zeitraum
H = [1.] # Liste mit Anfangswert für die Beutepopulation
L = [1.] # Liste mit Anfangswert für die Räuberpopulation
dt = 0.01 # Differenz pro Zeitschritt

# definiere eine Schleife
for t in T:
    if t == 4660:
        H[t] = 0.2 * H[t]
        L[t] = 0.2 * L[t]
    # berechne H für den jeweiligen Zeitschritt und füge das Resultat jeweils am Ende der Liste hinzu
    H.append(H[t] + (a * H[t] - b * H[t] * L[t]) * dt)
    # berechne L für den jeweiligen Zeitschritt und füge das Resultat jeweils am Ende der Liste hinzu
    L.append(L[t] + (c * L[t] * H[t] - d * L[t]) * dt)

# zeichnen
plt.plot(T, H[:-1], label = 'Beute')
plt.plot(T, L[:-1], label = 'Räuber')
plt.title('Simulierte Räuber-Beute-Beziehung mit Dezimierung der Populationen')
plt.grid()
plt.legend(loc='best')
print('Der Mittelwert der Beute-Population in der ersten Periode ist %.3f' %np.mean(H[:4660]))
print('Der Mittelwert in vergleichbarer Zeit in der zweiten Periode ist %.3f' %np.mean(H[4660:9320]))
print('Der Mittelwert der Räuber-Population in der ersten Periode ist %.3f' %np.mean(L[:4660]))
print('Der Mittelwert in vergleichbarer Zeit in der zweiten Periode ist %.3f' %np.mean(L[4660:9320]))
```

Der Mittelwert der Beute-Population in der ersten Periode ist 2.000
 Der Mittelwert in vergleichbarer Zeit in der zweiten Periode ist 2.739
 Der Mittelwert der Räuber-Population in der ersten Periode ist 1.001
 Der Mittelwert in vergleichbarer Zeit in der zweiten Periode ist 0.639

Simulierte Räuber-Beute-Beziehung mit Dezimierung der Populationen



Wir sehen, auch die dritte Lotka-Volterra-Regel wird von unserem Modell erfüllt.

Zusammenfassung

Gekoppelte Differentialgleichungssysteme

Räuber-Beute-Systeme von Tierpopulationen werden mathematisch mit gekoppelten Differentialgleichungssystemen dargestellt und, weil zumeist keine analytische Lösung möglich ist, am Computer als Differenzengleichungssystem simuliert.

Phasenraumdarstellung

In dieser Darstellungsweise wird eine Systementwicklung nicht in der Zeit, sondern im Verhältnis der System-bestimmenden Dynamiken zueinander dargestellt. Bei Räuber-Beute-Systemen wird beispielsweise die Beute-Population auf der x-Achse und die Räuber-Population auf der y-Achse dargestellt.

Die Lotka-Volterra-Regeln

- Die Räuber- und die Beute-Populationen oszillieren periodisch und zueinander zeitlich versetzt. Die Räuber-Population läuft der Beute-Population zeitlich etwas hinterher.
- Die durchschnittlichen Größen der beiden Populationen bleiben über längere Zeiträume konstant, auch wenn Maxima und Minima unterschiedlich sind.
- Werden Räuber- und Beute-Population gleichzeitig um den gleichen Prozentanteil dezimiert, so steigt der Mittelwert der Beutepopulation kurzfristig an, und der Mittelwert der Räuberpopulation sinkt kurzfristig ab.

Integrationsmethode

Es hängt von der Integrationsmethode und der Größe der gewählten Zeitschritte ab, wie genau das Ergebnis einer Integration ist. Die Integration per Differenzengleichung bietet, auch mit sehr klein gewähltem `dt`, stets nur eine Annäherung an das Ergebnis einer analytischen Lösung. Genaue Integrationsmethoden lernen wir im nächsten Kapitel kennen.

Kapitel 10 – Analytisch Differenzieren und Integrieren mit dem Paket Sympy

Bisher haben wir mit Python immer numerische Berechnungen durchgeführt. Das heißt, alle Differentialen wurden vom Computer als kleine Differenzen genähert und irrationale Zahlen, wie zum Beispiel Wurzeln wurden schlichtweg gerundet. Das ist zwar die einfachste Art, wie ein Computer Mathematik betreiben kann, nicht aber die einzige. Mit Python ist es auch möglich, mathematische Probleme analytisch zu lösen, also so, wie sie ein Mensch lösen würde. So können wir Stammfunktionen berechnen, Gleichungen auflösen und komplexe mathematische Probleme lösen, für die ein Mensch Tage brauchen würde.

Fangen wir an, indem wir den Unterschied zwischen numerischen und analytischen Berechnungen genauer betrachten. Als Beispiel soll ein einfacher Prozess, das ziehen einer Quadratwurzel, dienen. Berechnen wir die Wurzel aus 9 mit dem Standard-Mathematikpaket math.

```
In [1]: import math  
math.sqrt(9)  
  
Out[1]: 3.0  
  
In [2]: import math  
math.sqrt(8)  
  
Out[2]: 2.8284271247461903
```

Hier bekommen wir zwar eine näherungsweise Lösung, korrekt ist dieses Ergebnis aber nicht mehr. Die Wurzel aus 8 ist nicht exakt 2.8284271247461903, es würden noch unendlich viele Nachkommastellen folgen. Wieso kann das ein Problem werden? Versuchen wir einmal die Wurzel aus 8 mit der Wurzel aus 8 zu multiplizieren. Wir (als Menschen) müssen die Wurzel aus 8 dazu nicht kennen, wir wissen ja, dass die Wurzel aus 8 mal der Wurzel aus 8 wieder 8 sein muss.

$$\sqrt{8} * \sqrt{8} = \sqrt{8 * 8} = \sqrt{8^2} = 8$$

Wurzel und Quadrat heben sich sozusagen auf. Wenn wir diesen Zusammenhang schon verstehen, ohne überhaupt im Kopf die Wurzel ziehen zu können, dann sollte ein Computer mit dieser Aufgabe doch auch kein Problem haben, oder?

```
In [3]: import math  
math.sqrt(8) * math.sqrt(8)  
  
Out[3]: 8.00000000000002
```

Scheinbar doch. Python behauptet felsenfest, dass die Wurzel aus 8 zum Quadrat ein bisschen größer ist als 8. Der Fehler ist klein, aber man muss ihn ernst nehmen. In einem umfangreichen Programm werden solche Berechnungen nämlich sehr oft hintereinander ausgeführt, und der Fehler schaukelt sich auf. Und solche Fehler entstehen natürlich nicht nur beim Wurzel ziehen, sondern bei jeder Operation, die in irgendeiner Form runden muss. Schon die Zahl 1/3 kann nicht mehr exakt gespeichert werden, da man für die Darstellung als 0.3333333333.... unendlich viele Stellen bräuchte um exakt zu sein.

Wie kann man dieses Problem lösen? Mit analytischen Berechnungen! Wenn Python verstehen würde, was 1/3 bedeutet, also nicht nur eine Zahl mit endlich vielen Stellen, sondern die Rechenoperation, die die Zahl 1 in 3 gleich große Teile teilt, wären wir nicht mehr auf Runden angewiesen. Noch schöner wäre es, wenn Python auch die Operation des Wurzelziehens verstehen würde, sodass das Quadrat einer Wurzel automatisch immer wieder die Zahl selbst ergeben würde. All das, und noch viel mehr ist möglich, mit dem Python-Paket sympy.

Wir sehen uns dieses Paket im Folgenden genauer an und benutzen es als erstes, um damit eine Wurzel zu ziehen:

```
In [4]: import sympy  
sympy.sqrt(9)  
  
Out[4]: 3
```

Unser erstes erfreuliches Ergebnis: Die Wurzel aus 9 ist nach wie vor 3. Was passiert aber mit der Wurzel aus 8?

```
In [5]: import sympy  
sympy.sqrt(8)  
  
Out[5]: 2*sqrt(2)
```

Dieses Ergebnis sieht zwar sonderbar aus, ist aber absolut korrekt. Die Wurzel aus 8 ist exakt doppelt so groß wie die Wurzel aus 2.

Für Mathematiker: $\sqrt{8} = \sqrt{4 * 2} = \sqrt{4} * \sqrt{2} = 2 * \sqrt{2}$

Der Vorteil, den wir hier haben ist offensichtlich: Dieses Ergebnis ist exakt, wir müssen nicht runden. Und wenn wir wollen, können wir das Ergebnis immer noch in eine Dezimalzahl umrechnen:

```
In [6]: import sympy  
float(sympy.sqrt(8))  
  
Out[6]: 2.8284271247461903
```

Nun könnten wir noch überprüfen, ob das Quadrieren einer Wurzel so besser funktioniert:

```
In [7]: import sympy  
sympy.sqrt(8) * sympy.sqrt(8)  
  
Out[7]: 8
```

Offensichtlich. Der Vorteil dieser Berechnung ist noch gering, denn die Wurzel aus 8 kann man auch im Kopf umformen (siehe die Berechnung weiter oben). Hier kommt dann aber der Vorteil eines Computers zum Tragen: Wenn das Prinzip funktioniert, sind kompliziertere Berechnungen nicht wirklich schwieriger als einfache Berechnungen, wie folgendes Beispiel zeigt:

```
In [8]: import sympy
sympy.sqrt(1264)

Out[8]: 4*sqrt(79)
```

Symbolisches Programmieren

Die Wurzel aus 1264 ist also 4 mal die Wurzel aus 79. Auch mit viel Übung kann man solche Berechnungen nicht ohne weiteres im Kopf durchführen. Das Paket sympy kann das - und noch viel mehr. Besonders spannend wird es, wenn wir selbst Variablen definieren. Dafür gibt es den Befehl `symbol`, der Python erklärt, dass es sich bei diesen Variablen um allgemeine Variablen handelt, in denen kein Wert gespeichert ist. Trotzdem kann Python damit rechnen. Wir können einen Ausdruck speichern, der x und y beinhaltet und Python kann damit arbeiten, obwohl wir x und y keine Werte zugewiesen haben, sondern sie nur als allgemeine `Symbol` verwenden:

```
In [9]: from sympy import *
x = symbols("x")
y = symbols("y")
ausdruck = 3 * x - 4 * y
ausdruck
```

```
Out[9]: 3*x - 4*y
```

```
In [10]: ausdruck - 1
```

```
Out[10]: 3*x - 4*y - 1
```

```
In [10]: ausdruck - 1
```

```
Out[10]: 3*x - 4*y - 1
```

```
In [11]: ausdruck + 3 * y
```

```
Out[11]: 3*x - y
```

```
In [12]: ausdruck * x # die Lösung wird hier noch nicht ausmultipliziert
```

```
Out[12]: x*(3*x - 4*y)
```

```
In [13]: expand(ausdruck * x) # expand führt die Multiplikation aus
```

```
Out[13]: 3*x**2 - 4*x*y
```

Man sieht sofort: Python versteht nun wirklich die Rechenoperationen, und obwohl der Variablen x kein Wert zugeordnet wurde, ist klar, dass x mal x das gleiche ist wie x -Quadrat. Das können wir zum Beispiel nutzen um sehr komplizierte Ausdrücke mit dem Befehl `factor` zu vereinfachen:

```
In [14]: from sympy import *
x = symbols("x")
y = symbols("y")
z = symbols("z")
ausdruck = -x * y * z**2 + 2 * x * y * z - x * z**2 + 2 * x * z + y * z**2 - 2 * y * z + z**2 - 2 * z
factor(ausdruck)
```

```
Out[14]: -z*(x - 1)*(y + 1)*(z - 2)
```

Der ausgesprochen komplizierte Ausdruck

$-x * y * z^{**2} + 2 * x * y * z - x * z^{**2} + 2 * x * z + y * z^{**2} - 2 * y * z + z^{**2} - 2 * z$

lässt sich also kompakt schreiben als

$-z*(x - 1)*(y + 1)*(z - 2)$

Das ist zwar hilfreich, in der Praxis wird man aber sehr selten in die Situation kommen, solche Vereinfachungen durchführen zu müssen (außer vielleicht im Rahmen von Mathematiklehrveranstaltungen). Viel öfter möchte man zum Beispiel Gleichungen lösen.

Lösen von Gleichungen

Während zur Lösung quadratischer Gleichungen noch Formeln bereitstehen, sind Gleichungen 3-ter Ordnung, wie beispielsweise

$$3x^3 + x^2 - x = 0$$

auf dem Papier ungleich schwerer zu lösen. Schneller geht es mit dem `sympy`-Befehl `solve`. Dieser Befehl benötigt nur die Gleichung und als zweites Argument die Variable, die ausgerechnet werden soll:

```
In [15]: from sympy import *
x = symbols("x")
solve(3 * x**3 + x**2 - x , x)

Out[15]: [0, -1/6 + sqrt(13)/6, -sqrt(13)/6 - 1/6]
```

Die drei Lösungen der Gleichung $3x^3 + x^2 - x = 0$ sind also $x = 0$, $x = -\frac{1}{6} + \frac{\sqrt{13}}{6}$ und $x = -\frac{1}{6} - \frac{\sqrt{13}}{6}$

Differenzieren und Integrieren

Seinen größten Vorteil aus der Perspektive der Systemwissenschaften spielt `sympy` beim Berechnen von Differentialen und Integralen aus. Das numerische Differenzieren und Integrieren haben wir bereits in früheren Kapiteln besprochen. Die Lösungen dort waren allerdings immer Zahlen oder Listen von Zahlen. Allgemeine, analytische Erkenntnisse, wie die, dass das Integral von $\cos(x)$ exakt $\sin(x)$ ist, konnten wir so nicht gewinnen. Mit Hilfe von `sympy`'s `diff` und `integrate` Befehlen können wir auch sehr komplizierte Ableitungen und Integrale analytisch durchführen und bekommen als Lösungen keine Zahlen, sondern vollständige, allgemein gültige Funktionen:

```
In [16]: from sympy import *
x = symbols("x")

In [17]: diff(sin(x))
Out[17]: cos(x)

In [18]: integrate(cos(x))
Out[18]: sin(x)

In [19]: diff(sin(3 * x) * x**2)
Out[19]: 3*x**2*cos(3*x) + 2*x*sin(3*x)

In [20]: integrate(sin(x) * cos(x))
Out[20]: sin(x)**2/2

In [21]: diff(x**3 + log(x) * sin(x) + exp(x**2))
Out[21]: 3*x**2 + 2*x*exp(x**2) + log(x)*cos(x) + sin(x)/x
```

Aber Achtung: Integrale, die analytisch nicht lösbar sind, kann auch Python nicht lösen. Differenzieren läuft immer nach sehr simplen Regeln ab, und prinzipiell kann man mit diesen Regeln jede Funktion ableiten. Integrieren ist jedoch mehr ein kreativer Prozess, und hierbei können Zusammenhänge auftauchen, die sich einfach nicht berechnen lassen. Das hat nichts damit zu tun, dass die richtigen "Rechenregeln" nur einfach noch nicht gefunden wurden. Es ist vielmehr eine Eigenschaft der Integralerechnung selbst. Unlösbarer Integrale müssen nicht schrecklich kompliziert sein. Schon das einfache Beispiel $\int \frac{\sin(x)}{\log(x)} dx$ ist nicht mehr integrierbar:

```
In [22]: integrate(sin(x) / log(x))
Out[22]: Integral(sin(x)/log(x), x)
```

Wenn die obige Zelle ausgeführt wird, zeigt sich, dass Python eine Zeitlang rechnet und offensichtlich versucht das Integral zu lösen. Wenn allerdings keine partielle Integration, keine Substitutionen oder andere Tricks mehr weiterhelfen, kapituliert Python (und auch jegliche andere Mathematik-Software und das menschliche Hirn). Als Lösung bekommt man dann die Aussage:

$$\int \frac{\sin(x)}{\log(x)} dx = \int \frac{\sin(x)}{\log(x)} dx$$

Das ist natürlich (per Definition) nicht falsch, bringt uns aber auch nicht wirklich weiter. Wenn wir dieses Integral in einem bestimmten Bereich lösen wollen, sind wir wieder auf numerische Methoden angewiesen. Diese liefern zwar keine schönen Funktionen, sondern immer nur gerundete Zahlen. Dafür funktionieren sie aber wirklich immer. Man sieht also, dass sowohl numerische, als auch analytische Methoden ihre Daseinsberechtigung haben, und man sich je nach Problem für eine der beiden Herangehensweisen entscheiden sollte. Gerade im mathematischen Bereich hat das Arbeiten mit analytischen Funktionen große Vorteile.

Die Anwendungsgebiete von `sympy` enden aber nicht mit Differenzieren und Integrieren. Man kann damit auch Grenzwerte berechnen, Differentialgleichungen lösen, Eigenwerte und Eigenvektoren ausrechnen oder (wenn man das wirklich möchte ;-)) eine Bessel-Funktion in eine sphärische Bessel-Funktion transformieren.

Rechnen mit Matrizen

Sympy erlaubt es zum Beispiel auch mit Matrizen zu rechnen. Alle wichtigen Rechenoperationen werden unterstützt. Matrizen werden in eckigen Klammern geschrieben, wobei jede Zeile ihrerseits in eckige Klammern gesetzt wird. Sympy betrachtet Matrizen also eigentlich als Listen von Listen. Die Matrix

$$M = \begin{pmatrix} 1 & 3 & 2 \\ 1 & 1 & 1 \\ 2 & 3 & 1 \end{pmatrix}$$

kann demnach geschrieben werden als:

```
In [23]: M = Matrix([[1,3,2],[1,1,1],[2,3,1]])
M
Out[23]: Matrix([
[1, 3, 2],
[1, 1, 1],
[2, 3, 1]])
```

Grundrechenarten funktionieren ganz intuitiv mit den dazugehörigen Operatoren `+`, `*` und `:`.

```
In [24]: M + M
Out[24]: Matrix([
[2, 6, 4],
[2, 2, 2],
[4, 6, 2]])
```

```
In [25]: 3 * M
```

```
Out[25]: Matrix([
 [3, 9, 6],
 [3, 3, 3],
 [6, 9, 3]])
```

```
In [26]: M - M
```

```
Out[26]: Matrix([
 [0, 0, 0],
 [0, 0, 0],
 [0, 0, 0]))
```

Auch die (per Hand oft langwierige) Matrix-Multiplikation lässt sich mit sympy einfach handhaben.

```
In [27]: M = Matrix([[1,3,2],[1,1,1],[2,3,1]])
```

```
N = Matrix([[4,1,2],[1,2,1],[1,2,4]])
```

```
M * N
```

```
Out[27]: Matrix([
 [ 9, 11, 13],
 [ 6,  5,  7],
 [12, 10, 11]])
```

```
In [28]: N * M
```

```
Out[28]: Matrix([
 [ 9, 19, 11],
 [ 5,  8,  5],
 [11, 17,  8]])
```

Als schnelle Überprüfung, ob es sich wirklich um eine echte Matrixmultiplikation handelt, und nicht einfach nur die Elemente miteinander multipliziert werden, kann man $M \cdot N$ ausrechnen lassen, und das Ergebnis mit $N \cdot M$ vergleichen. In unserem Beispiel sehen wir, dass $M \cdot N$ nicht das gleiche ist wie $N \cdot M$, ganz so wie es sich für Matrixmultiplikationen gehört.

Aber auch erweiterte Rechenoperationen sind möglich. Beispielsweise das Berechnen von Determinanten, welches unter anderem zum Lösen von Gleichungssystemen mit der Cramerschen Regel (https://de.wikipedia.org/wiki/Cramersche_Regel) benötigt wird:

```
In [29]: M.det()
```

```
Out[29]: 3
```

Auch das Berechnen von Eigenwerten ist möglich:

```
In [30]: M.eigenvals()
```

```
Out[30]: {-1: 1, -sqrt(7) + 2: 1, 2 + sqrt(7): 1}
```

Zusammenfassung

Das Paket sympy ermöglicht es mit Python analytische Berechnungen durchzuführen. Das hat den Vorteil, dass Zahlen nicht gerundet werden, sondern Brüche oder Wurzeln exakt verwendet werden können. So werden nicht nur Rundungsfehler verhindert, sondern viele Rechenoperationen möglich, die numerisch nicht, oder nur näherungsweise gelingen.

Lösen von Gleichungen

Das Lösen von Gleichungen erfordert numerisch in der Regel Tricks und Kreativität. Mit sympy lassen sich Gleichungen aber exakt lösen. Symbole, die in diesen Gleichungen vorkommen, werden zunächst mit dem Befehl `symbols` definiert und sodann mit dem Befehl `solve(gleichung, variable)` einer Lösung nach der gesuchten Variablen zugeführt.

Differenzieren

Die Ableitung einer Funktion wird mit `diff` berechnet, wobei auch hier die vorkommenden Variablen zuvor mit `symbols` zu deklarieren sind.

Integrieren

Das Integrieren, also das Suchen der Stammfunktion, geschieht mit dem Befehl `integrate`.

Merke: Nicht alle Funktionen können analytisch integriert werden. Für viele existiert keine analytische Lösung. Für solche Probleme muss auf numerische Methoden zurückgegriffen werden. Diese benötigen zwar längere Rechenzeit und liefern Lösungen nur näherungsweise und in bestimmten Intervallen. Dafür funktionieren sie in der Regel immer.

Kapitel 11 – Numerische Integrationsmethoden

Im vorhergehenden Kapitel haben wir gesehen, dass auch mächtige digitale Werkzeuge wie das Python-Paket `sympy` nicht für alle Integrationsprobleme analytische Lösungen finden. Für viele dieser Probleme existiert einfach keine Lösung, die von Menschen oder Computern in vertretbarer Zeit gefunden werden könnte. In diesen Fällen muss auf numerische Methoden zurückgegriffen werden, sprich auf die schrittweise Integration mit Differenzengleichungen, wie wir sie in Kapitel 9 kennengelernt haben.

Dort haben wir dazu festgestellt, dass Modellierungsergebnisse stark von der Genauigkeit der numerischen Integration abhängen können. Das Problem besteht dabei darin, dass am Computer ein als gegen Null gehend gedachter Zeitschritt aufgrund der diskreten Operationsweise notwendig immer größer als Null ist und dass nicht-lineare Veränderungsraten auch in kleinsten Zeitintervallen große Anfangsunterschiede für den je nächsten Berechnungsschritt generieren können. Damit kann schon nach nur wenigen Schritten nicht mehr gewährleistet sein, dass sich eine Entwicklung noch in dem Bereich befindet, in dem sie mit tatsächlich kontinuierlich (d.h. analytisch) berechneter Entwicklung sein müsste. (Merke: das Problem entsteht hier nicht so sehr aus Rundungsfehlern, sondern aus der Größe der gewählten Differenz dt , die notwendig stets > 0 ist)

Wir können dies mit Python einfach nachverfolgen, indem wir eine jener einfachen Entwicklungen betrachten, für die es eine analytische Lösung gibt, und das entsprechende Integrationsergebnis mit den Ergebnissen vergleichen, die die Methoden erbringen, die wir zuvor in Teil 9 dieses Skriptums angewandt haben.

Diese Entwicklung sei gegeben mit

$$\frac{dN}{dt} = \gamma N$$

Die exakte (analytische, sprich rein mathematische) Lösung dafür lautet $N_t = N_0 e^{\gamma t}$

Wir berechnen diese Entwicklung für N_{10} mit $N_0 = 1$, $\gamma = 0.7$ und $dt = (1, 0.1, 0.01, 0.001)$ und vergleichen die Resultate mit der analytischen Lösung.

```
In [1]: import matplotlib.pyplot as plt
%matplotlib inline
# um mit e zu rechnen
import numpy as np

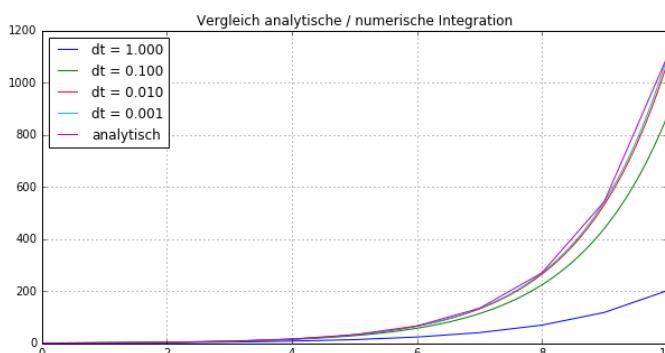
# Zeichenfläche
fig = plt.figure(figsize=(10,5))
fig.add_subplot(1,1,1)

#die Wachstumsrate
gamma = 0.7
# die betrachteten Zeitspannen, in Abhängigkeit der betrachteten Differenz dt
TT = [np.linspace(0, 10, 11), np.linspace(0, 10, 101), np.linspace(0, 10, 1001), np.linspace(0, 10, 10001)]
# die unterschiedlichen Differenzen
DT = [1, 0.1, 0.01, 0.001]
# eine Liste, in die die Ergebnisse der analytischen Lösung eingetragen werden
Ne = []
# eine Schleife, die durch die unterschiedlichen Differenzen iteriert
for n, dt in enumerate(DT):
    # eine Liste, in die die Ergebnisse der numerischen Lösungen eingetragen werden
    N = [1]
    # eine Schleife, die über die Zeit iteriert
    for t, x in enumerate(TT[n]):
        # Berechnung, deren Ergebnis in die Liste N eingetragen wird
        N.append(N[t] + (gamma * N[t]) * dt)
        # nur einmal (beim ersten Durchgang) wird das Ergebnis der analytischen Berechnung eingetragen
        if n == 0:
            Ne.append((1 * np.exp(gamma*t)))
            # das erste Ergebnis für dt = 1 wird für spätere Verwendung (s.u.) separiert
            N1 = N
    # die Ergebnisse für N werden gezeichnet
    plt.plot(TT[n], N[1:], label = ('dt = %.3f' %dt))
    # das Ergebnis für N nach 10 Zeitschritten wird ausgedruckt
    print('Mit dt = %.3f numerisch berechnet ergibt die Entwicklung nach 10 Zeitschritten %.9f' %(dt, N[-2]))

# das Ergebnis für die analytische Berechnung wird ausgedruckt
print('Analytisch (mit Euler-Zahl) berechnet, ergibt die Entwicklung nach 10 Zeitschritten %.9f' %(1*np.exp(gamma*10)))
plt.plot(TT[0], Ne, label = 'analytisch')
plt.title('Vergleich analytische / numerische Integration')
plt.xlim(0,10)
plt.grid()
plt.legend(loc = 'best')
```

Mit $dt = 1.000$ numerisch berechnet ergibt die Entwicklung nach 10 Zeitschritten 201.599390045
Mit $dt = 0.100$ numerisch berechnet ergibt die Entwicklung nach 10 Zeitschritten 867.716325566
Mit $dt = 0.010$ numerisch berechnet ergibt die Entwicklung nach 10 Zeitschritten 1070.213816958
Mit $dt = 0.001$ numerisch berechnet ergibt die Entwicklung nach 10 Zeitschritten 1093.950945868
Analytisch (mit Euler-Zahl) berechnet, ergibt die Entwicklung nach 10 Zeitschritten 1096.633158428

Out[1]: <matplotlib.legend.Legend at 0xacfc7e10>



Wir sehen, kleinere Zeitschritte erhöhen die Genauigkeit, lösen das Integrationsproblem aber nicht grundsätzlich. Die exakte analytische Berechnung würde ein genaues Ergebnis liefern, ist aber nur für einfache Entwicklungen verfügbar. Für kompliziertere Differentialgleichungen und für die meisten gekoppelten Differentialgleichungssysteme ist keine analytische Lösung verfügbar. Hier können Lösungen nur numerisch, sprich am Computer errechnet werden.

Die numerischen Methoden, die wir zuvor in Kapitel 9 und oben verwendet haben, entsprechen im Prinzip dem sogenannten

Euler-Verfahren (auch Polygonzug-, Euler-Cauchy, Euler-vorwärts-Verfahren oder Linearisierung)

(Vorsicht, auch wenn es so heißt, dieses Verfahren beruht eben gerade **nicht** auf Verwendung der Euler-Zahl)

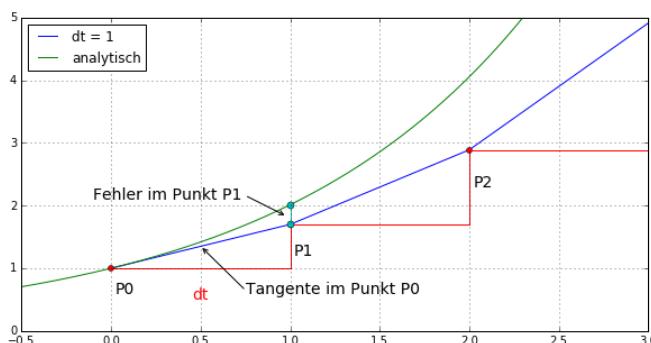
Dieses Verfahren entspricht der iterativen Berechnung von (rekursiven) Differenzengleichungen. Seine Ungenauigkeit wird besonders deutlich, wenn, wie im folgenden, eine relativ große Differenz (zB $dt = 1$) verwendet wird.

```
In [2]: import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(10,5))
fig.add_subplot(1,1,1)

# eine (kürzere) Zeitspanne
T = np.linspace(-1, 3, 401)
# das analytische Ergebnis wird für die kürzere Zeitspanne neu berechnet
Ne = []
for t in T:
    Ne.append((1 * np.exp(gamma*t)))
# das oben separierte Ergebnis für dt = 1 wird für die ersten drei Zeitschritte gezeichnet
plt.plot(N1[:4], label = 'dt = 1')
plt.plot(T, Ne, label = 'analytisch')
# eine Linie
plt.plot((0, 1), (1, 1), 'r-')
# ein Punkt
plt.plot(0, 1, 'ro', markersize = 5)
# eine Beschriftung
plt.text(0.02, 0.6, 'P0', fontsize = 14)
plt.plot((1, 1), (1, 1.7), 'r-')
plt.plot(1, 1.7, 'ro', markersize = 5)
plt.text(1.02, 1.2, 'P1', fontsize = 14)
plt.plot((1, 2), (1.7, 1.7), 'r-')
plt.plot((2, 2), (1.7, 2.889), 'r-')
plt.text(2.02, 2.3, 'P2', fontsize = 14)
plt.plot(2, 2.889, 'ro', markersize = 5)
plt.plot((2, 3), (2.889, 2.889), 'r-')
# eine Beschriftung
plt.text(.75, .6, 'Tangente im Punkt P0', fontsize = 14)
# ein Pfeil
plt.annotate('', xy=(0.5, 1.35), xytext=(0.75, 0.65), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=1,))
plt.text(.45, .5, 'dt', color = 'red', fontsize = 14)
# eine Beschriftung
plt.text(-.1, 2.1, 'Fehler im Punkt P1', fontsize = 14)
# ein Pfeil
plt.annotate('', xy=(0.98, 1.83), xytext=(0.75, 2.16), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=1,))
plt.plot((1, 1), (1.7, 2.), 'co-')
plt.xlim(-.5, 3)
plt.ylim(0, 5)
plt.grid()
plt.legend(loc = 'best')
```

Out[2]: <matplotlib.legend.Legend at 0xb0c8b00>



Der Punkt P1 ergibt sich dabei aus der Position einer **Tangente**, die fiktiv an die analytisch berechnete Kurve im Punkt P0 angelegt wird, nach einem Zeitschritt, also hier bei $x = 1$. Dieser Punkt P1, der bereits einigermaßen von der eigentlichen (analytisch gefundenen) Kurve abweicht (= Fehler), wird nun als Ausgangswert für die Berechnung des Punktes P2 herangezogen. Dadurch verstärkt sich der Fehler weiter. Mit jedem weiteren Berechnungsschritt wird damit der Abstand zur analytischen Lösung größer. Auch bei sehr kleinen Differenzen können sich so nach nur wenigen Berechnungsschritten große Abweichungen von der analytischen Lösung einstellen.

Verbessertes Euler-Verfahren

Im klassischen Euler-Verfahren wird, wie gezeigt, die Steigung der Tangente am Anfang des jeweiligen Berechnungsschrittes verwendet, um den je nächsten Punkt der Kurve zu berechnen. Am Ende des Schrittintervalls (der verwendeten Differenz) weicht die Näherungskurve deshalb je nach Schrittgröße relativ stark von der analytischen Lösung ab.

Alternativ dazu kann aber auch ein mittlerer Steigungswert des Schrittintervalls berechnet werden. In der so genannten Mittelpunktsregel wird die Steigung am Intervall-Anfang dazu herangezogen, die Richtung zum nächsten Punkt auf halbem Weg zwischen x_i und x_{i+1} zu „sondieren“. Von da aus wird die dort gefundene Steigung verwendet, um die Position des nächsten Kurvenpunktes zu berechnen. Dieses Verfahren liefert zwar deutlich genauere Ergebnisse, ist aber insbesondere bei stark nicht-linearen Entwicklungen ebenfalls nur begrenzt brauchbar.

Runge-Kutta-Verfahren

Im Anschluss an die Mittelpunktsregel wurde von den beiden Mathematikern Carl Runge und Martin W. Kutta um das Jahr 1900 herum vorgeschlagen, das Schrittintervall (die Differenz dt) noch weiter zu unterteilen und anstelle des obigen Mittelwerts einen gewichteten Mittelwert der Teilschrittweite zu verwenden, um den nächsten Kurvenpunkt zu berechnen.

Im sehr gebräuchlichen

Runge-Kutta-Verfahren vierter Ordnung (RK4)

zum Beispiel wird die Richtung viermal „sondert“: einmal links, zweimal in der Mitte und einmal am rechten Ende des jeweiligen Schrittintervalls. Für den gewichteten Mittelwert erhalten die beiden mittleren Werte sodann das jeweils doppelte Gewicht.

In Python stellt das `integrate`-Modul von `scipy` (www.scipy.org) dieses Verfahren unter der Bezeichnung `dopri5` neben einer Reihe weiterer Integrationsmethoden zur Verfügung. Der folgende Code vergleicht mehrere dieser Methoden in Bezug auf ihre Abweichung von der analytischen Lösung nach 10 Rechenschritten, die ersten vier davon mit dem (ungewöhnlich großen) Schrittintervall $dt = 1$. Die letzte Methode - `odeint` - haben wir bereits in Kapitel 9 dieses Skriptums verwendet. Sie sieht keine Möglichkeit vor, die Differenz dt explizit festzulegen.

```
In [3]: import numpy as np
from scipy import integrate
from scipy.integrate import ode

# eine Lambda-Funktion (siehe )
#func = Lambda t, y: gamma * y

gamma = 0.7

# eine Funktion
def f(t, y):
    return gamma * y

# eine Liste zur Bezeichnung der Methode
method = ['RK4', 'RK8', 'odeint' ]
# eine Liste zum Aufruf der Methode
backend = ['dopri5', 'dop853', 'lsoda']
# die Differenz
dt = 1
# die berechnete Gesamtzeit
T = np.linspace(0, 10, 11)
# eine Schleife, die durch die unterschiedlichen Methoden iteriert
for n, back in enumerate(backend):
    # die Bestimmung der Methode
    solver = ode(f).set_integrator(back)
    # Anfangswerte
    solver.set_initial_value(1, 0)
    # Leere Listen
    N = []
    Ni = []
    Ne = []
    # Schleife über die Zeit
    for t in T:
        # analytische Lösung
        Ne.append((1 * np.exp(gamma * t)))
        # numerische Lösung mit spezifischer Methode
        N.append(solver.y)
        solver.integrate(solver.t + dt)

    print('Methode %s: Differenz zwischen analytischer und numerischer Berechnung nach 10 Rechenschritten: %.10f'
          % (method[n], abs(N[-1] - Ne[-1])))
```

Methode RK4: Differenz zwischen analytischer und numerischer Berechnung nach 10 Rechenschritten: 0.0003926403
Methode RK8: Differenz zwischen analytischer und numerischer Berechnung nach 10 Rechenschritten: 0.0000195433
Methode odeint: Differenz zwischen analytischer und numerischer Berechnung nach 10 Rechenschritten: 0.0068688743
In diesem Vergleich liefert die Methode Runge-Kutta 8ter Ordnung (RK8) das genaueste Ergebnis. Ähnlich zu RK4 unterteilt sie das Schrittintervall in 8 Teile, aus denen wieder ein gewichteter Mittelwert hergeleitet wird. Wie leicht vorzustellen ist, benötigt diese Methode allerdings auch deutlich mehr Rechenzeit als zum Beispiel RK4. In der Regel wird deshalb oftmals RK4 als geeigneter Kompromiss zwischen Genauigkeit und Rechenzeit verwendet.

In Python bietet sich allerdings auch die leichter anwendbare `odeint`-Methode als brauchbar und oft hinreichend genau an.

Zusammenfassung

Euler-Verfahren

Die Integration mittels Euler-Verfahren (auch Polygonzug-, Euler-Cauchy-, Euler-vorwärts-Verfahren oder Linearisierung) entspricht der iterativen Berechnung von (rekursiven) Differenzengleichungen, wobei die sich (gegenüber der analytischen Lösung) ergebende Ungenauigkeit von der Größe der verwendeten Differenz (dt) abhängt.

Runge-Kutta-Verfahren

Die Integration mittels Runge-Kutta-Verfahren unterteilt das Schrittintervall (die Differenz dt) in mehrere Teilschritte und verwendet einen gewichteten Mittelwert der Teilschrittweite, um den je nächsten Kurvenpunkt zu berechnen. Das sehr gebräuchliche Runge-Kutta-Verfahren vierter Ordnung (RK4) zum Beispiel unterteilt das Schrittintervall viermal und teilt dabei den beiden mittleren Werte das jeweils doppelte Gewicht zu.

Kapitel 12 – Visualisierungen

Egal, was programmiert werden soll, ob ein Modell erstellt wird, eine Simulation durchgeführt oder statistische Daten ausgewertet werden: am Ende sollen die Ergebnisse meist visualisiert werden. Die Darstellung von Ergebnissen - das "Plotten" - ist oft ebenso wichtig, wie die Ergebnisse selbst und es gibt unzählige Arten des Visualisierens.

Einige wichtige Regeln zum Erstellen von Grafiken sollten, unabhängig von der Art der Darstellung, immer beachtet werden:

- alle Achsen sollten beschriftet sein
- wenn möglich sollte im Plot ersichtlich sein, welche Einheiten verwendet werden
- alle relevanten Größen sollten erkennbar sein
- Farben sollten so gewählt werden, dass sie auch in Graustufen gut unterscheidbar sind. Alternativ können Markierungen verwendet werden.

Im Folgenden geben wir eine Übersicht über die gängigsten Arten, Ergebnisse zu visualisieren. Einige davon haben wir bereits kennengelernt, anderen sind wir noch nicht begegnet.

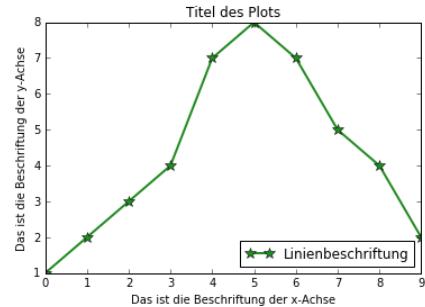
Der Linienplot

```
In [1]: import matplotlib.pyplot as plt
%matplotlib inline

testdaten = [1, 2, 3, 4, 7, 8, 7, 5, 4, 2]
plt.xlabel("Das ist die Beschriftung der x-Achse")
plt.ylabel("Das ist die Beschriftung der y-Achse")
plt.title("Titel des Plots")

plt.plot(testdaten, color = "forestgreen", lw = 2, marker = "*", markersize = 10, label = "Linienbeschriftung")
plt.legend(loc = "best")
```

Out[1]: <matplotlib.legend.Legend at 0xa988d30>



Ein Linienplot ist eine der einfachsten Arten, Daten zu visualisieren. Immer wenn man eine Größe (z.B. Anzahl der Frösche) hat, die man gegen eine andere Größe auftragen möchte (z.B. Zeit) bietet sich ein Linienplot an.

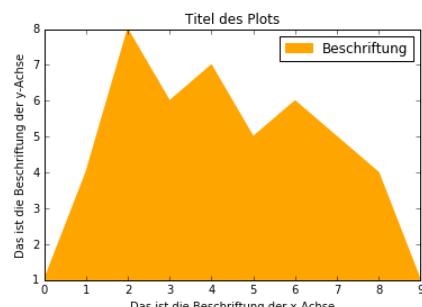
Der Fill-Plot

```
In [2]: import matplotlib.pyplot as plt
%matplotlib inline

testdaten = [1, 4, 8, 6, 7, 5, 6, 5, 4, 1]
plt.xlabel("Das ist die Beschriftung der x-Achse")
plt.ylabel("Das ist die Beschriftung der y-Achse")
plt.title("Titel des Plots")

plt.fill(testdaten, color = "orange", label = "Beschriftung")
plt.legend(loc = "best")
```

Out[2]: <matplotlib.legend.Legend at 0xa988cf8>



Der Fill-Plot ist konzeptionell ähnlich wie der Linienplot, nur möchte man hier eher auf die Fläche unter der Kurve hinweisen, als auf die Kurve selbst.

Der Stackplot

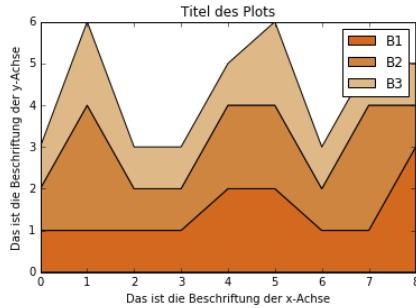
```
In [3]: import matplotlib.pyplot as plt
%matplotlib inline

testdaten = [1, 1, 1, 1, 2, 2, 1, 1, 3]
testdaten2 = [1, 3, 1, 1, 2, 2, 1, 3, 1]
testdaten3 = [1, 2, 1, 1, 1, 2, 1, 1, 1]

plt.xlabel("Das ist die Beschriftung der x-Achse")
plt.ylabel("Das ist die Beschriftung der y-Achse")
plt.title("Titel des Plots")

plt.stackplot(range(len(testdaten)), testdaten, testdaten2, testdaten3, labels = ("B1","B2","B3"),
              colors=["chocolate","peru","burlywood"])
plt.legend(loc = "best")
```

Out[3]: <matplotlib.legend.Legend at 0xace67b8>



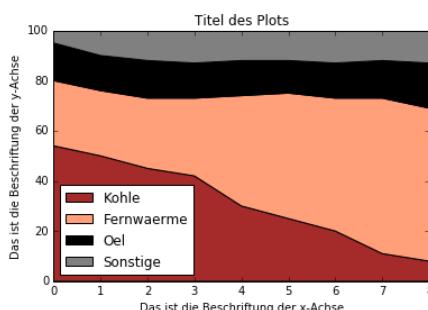
Beim Stackplot können wir mehrere Fill-Plots übereinanderstapeln. Der Stackplot ist relativ schwer zu lesen, da man ihn leicht mit einem Linienplot verwechselt. In unserem Beispiel, würde man also fälschlicherweise glauben, dass die Größe B3 am Anfang den Wert 3 hat. In Wirklichkeit ist der Plot aber so zu lesen: B3 geht von 2 bis 3, hat also den Wert 1. Auch Steigungen kann man in einem Stackplot nicht mehr korrekt ablesen. Stackplots sind also mit Vorsicht zu genießen. Wo sie aber doch Sinn machen, ist wenn man Größen hat, die sich immer auf eine gemeinsame Summe ergänzen. Zum Beispiel den prozentuellen Anteil einer gewissen Heiz-Technologie. Alle Technologien sollten sich immer auf 100% ergänzen:

```
In [4]: import matplotlib.pyplot as plt
%matplotlib inline

tec1 = [54,50,45,42,30,25,20,11,8]
tec2 = [26,26,28,31,44,50,53,62,61]
tec3 = [15,14,15,14,14,13,14,15,18]
tec4 = [ 5,10,12,13,12,12,13,12,13]
plt.xlabel("Das ist die Beschriftung der x-Achse")
plt.ylabel("Das ist die Beschriftung der y-Achse")
plt.title("Titel des Plots")

plt.stackplot(range(len(tec1)), tec1, tec2, tec3, tec4, labels=("Kohle","Fernwaerme","Oel","Sonstige"),
              colors = ["brown","lightsalmon","black","gray"])
plt.legend(loc = "lower left")
```

Out[4]: <matplotlib.legend.Legend at 0xaea35f8>



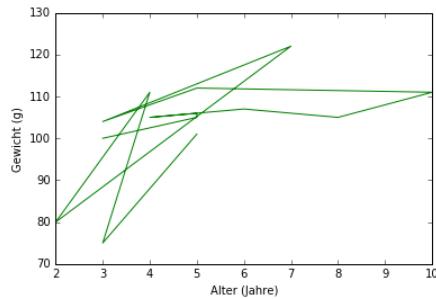
Der Scatterplot

Alle bisherigen Plots hatten mehr oder weniger das gleiche Anwendungsgebiet: Wir möchten eine Größe y (Anzahl der Frösche) einer anderen Größe x (Zeit) eindeutig zuordnen. Was aber, wenn diese Zuordnung nicht eindeutig ist. Was ist, wenn wir die Frösche wiegen und ihr Alter bestimmen wollen. In diesem Fall ist es nicht so, dass wir jedem Alter ein exaktes Gewicht zuordnen können: Frösche mit gleichem Alter können unterschiedliches Gewicht haben, und Frösche mit unterschiedlichem Alter könnten durchaus das gleiche Gewicht haben. Was passiert also, wenn wir alle Frösche im Teich wiegen, und sowohl ihr Alter als auch ihr Gewicht grafisch darstellen wollen? Versuchen wir zuerst einen Linienplot.

```
In [5]: import matplotlib.pyplot as plt
%matplotlib inline

# Die Daten sind wie folgt aufgebaut:
# froschgewicht = (gewicht von frosch1, gewicht von frosch2, gewicht von frosch3, ...)
# froschalter = (alter von frosch1, alter von frosch2, alter von frosch3, ...)
froschgewicht = (100,105,106,105,107,105,111,112,104,122,80,111,75,101)
froschalter = (3, 5, 4, 6, 8, 10, 5, 3, 7, 2, 4, 3, 5)
plt.plot(froschalter, froschgewicht, color = "green")
plt.xlabel("Alter (Jahre)")
plt.ylabel("Gewicht (g)")
```

Out[5]: <matplotlib.text.Text at 0xae0c4e0>

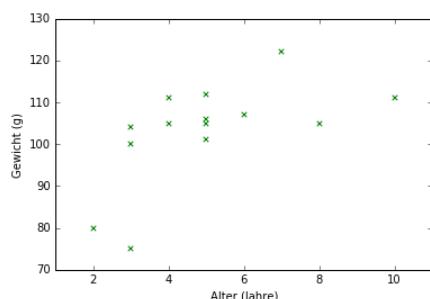


Dieser Plot ist prinzipiell nicht falsch, er ist nur sehr verwirrend. Die Verbindungslien suggestieren, dass die Frösche, die zufällig nacheinander gewogen wurden, irgend etwas miteinander zu tun hätten. Die Reihenfolge der Frösche hat in diesem Plot aber keine Bedeutung, die Frösche hätten auch in ganz anderer Reihenfolge gewogen werden können. Um diesen Plot zu reparieren, sollten wir also die Verbundungslien entfernen und einfach nur für jeden Frosch eine Markierung vornehmen. Damit entsteht ein so genannter **Scatter-Plot**.

```
In [6]: import matplotlib.pyplot as plt
%matplotlib inline

# Die Daten sind wie folgt aufgebaut:
# froschgewicht = (gewicht von frosch1, gewicht von frosch2, gewicht von frosch3, ...)
# froschalter = (alter von frosch1, alter von frosch2, alter von frosch3, ...)
froschgewicht = (100,105,106,105,107,105,111,112,104,122,80,111,75,101)
froschalter= (3, 5, 4, 6, 8, 10, 5, 3, 7, 2, 4, 3, 5)
plt.scatter(froschalter, froschgewicht, color = "green", marker = "x")
plt.xlabel("Alter (Jahre)")
plt.ylabel("Gewicht (g)")
```

Out[6]: <matplotlib.text.Text at 0xb420cc0>



Das Kuchendiagramm

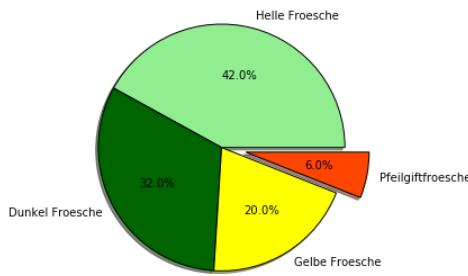
Ein Klassiker unter den Visualisierungen, der sehr gut geeignet ist, um prozentuelle Anteile darzustellen, ist das Kuchen- oder auch Tortendiagramm. Als wissenschaftliches Diagramm ist es nur bedingt geeignet, da man konkrete Größen nur schwer ablesen kann. Für Präsentationen, oder um sich einen groben Überblick über Daten zu verschaffen, eignet es sich aber recht gut:

```
In [7]: import matplotlib.pyplot as plt
%matplotlib inline

labels = ('Helle Froesche', 'Dunkel Froesche', 'Gelbe Froesche', 'Pfeilgiftfroesche')
farben = ("lightgreen","darkgreen","yellow", "orangered")
anzahl = [42, 32, 20, 6]
explode = (0, 0, 0, 0.2) # Hervorgehoben werden nur Pfeilgiftfrösche

plt.pie(anzahl, explode = explode, labels = labels, autopct = '%1.1f%%',
        shadow = True, colors = farben)
plt.axis('equal') # macht den Plot quadratisch
```

```
Out[7]: (-1.0000000157694127,
 1.1964574459176487,
 -1.0112847428214335,
 1.0237638451210083)
```



Subplots

Oft möchte man mehrere Grafiken nebeneinander darstellen. Dafür eignen sich so genannte subplots. Die Syntax dafür ist ein wenig gewöhnungsbedürftig. Subplots werden immer mit drei Zahlen definiert, wobei zum Beispiel die Zahlenfolge (2, 3, 4) - oder alternativ auch (234) - bedeutet, dass sämtliche Plots, die dargestellt werden sollen, in 2 Zeilen und 3 Spalten angeordnet werden und aktuell (mit der letzten Zahl 4) der 4te von 6 Plots (2 mal 3) angesprochen wird.

Beim Definieren von Subplots wird oftmals mit `fig = plt.figure(figsize=(10, 3))` zunächst eine allgemeine Zeichenfläche festgelegt. `figsize` definiert dabei die Größe der gesamten Zeichenfläche und der Befehl `subplots_adjust(hspace = 0.5)` erlaubt es, die Abstände zwischen den Subplots zu definieren, hier zum Beispiel die Höhenabstände (`hspace`). Mit Befehlen wie `ax1 = fig.add_subplot(2, 3, 1)` wird sodann der jeweilige Subplot in diese allgemeine Zeichenfläche eingefügt - hier also der Subplot `ax1` an die erste Stelle der Plots.

Merk: wenn statt der `matplotlib.pyplot`-Abkürzung `plt` ein eigener Plotname definiert wird, wie hier etwa `ax1`, so müssen sämtliche Plot-Beschriftungen mit `set_` vorgenommen werden, also zum Beispiel `ax1.set_title('Plot 1')`.

```
In [8]: import matplotlib.pyplot as plt
%matplotlib inline

testdaten=[2,3,4,4,2]

# definiere eine allgemeine Zeichenfläche
fig = plt.figure(figsize=(10, 4))
# legt Höhenabstände zwischen den Subplots fest
plt.subplots_adjust(hspace = 0.5)

# füge ersten Subplot zur allgemeinen Zeichenfläche hinzu
ax1 = fig.add_subplot(2, 3, 1)
ax1.plot(testdaten, color = "lime")
ax1.set_title('Plot 1')

# zweiter Subplot
ax2 = fig.add_subplot(2, 3, 2)
ax2.fill(testdaten, color = "green")
ax2.set_title('Plot 2')

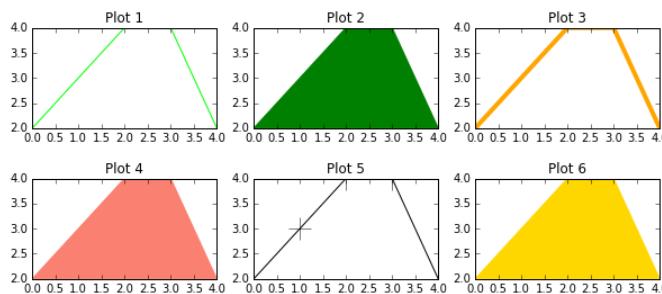
# dritter Subplot
ax3 = fig.add_subplot(2, 3, 3)
ax3.plot(testdaten, color = "orange", lw=4)
ax3.set_title('Plot 3')

# vierter Subplot
ax4 = fig.add_subplot(2, 3, 4)
ax4.fill(testdaten, color = "salmon")
ax4.set_title('Plot 4')

# fünfter Subplot
ax5 = fig.add_subplot(2, 3, 5)
ax5.plot(testdaten, color = "black", marker="+", ms=20)
ax5.set_title('Plot 5')

# sechster Subplot
ax6 = fig.add_subplot(2, 3, 6)
ax6.fill(testdaten, color = "gold")
ax6.set_title('Plot 6')
```

```
Out[8]: <matplotlib.text.Text at 0xc54c8d0>
```



Das Polardiagramm

Das Polardiagramm sieht zwar ähnlich aus, wie ein Kuchendiagramm, hat aber eine grundlegend andere Bedeutung. Wichtig ist hier der Winkel: jedem Winkel (0 - 360 Grad) wird eine bestimmte Größe zugeordnet. In der Physik wird dieser Plot zum Beispiel verwendet, um darzustellen, in welche Richtung ein Objekt wie viel Strahlung emittiert. Als einfacheres Beispiel könnte man damit auch das Sichtfeld verschiedener Tiere vergleichen:

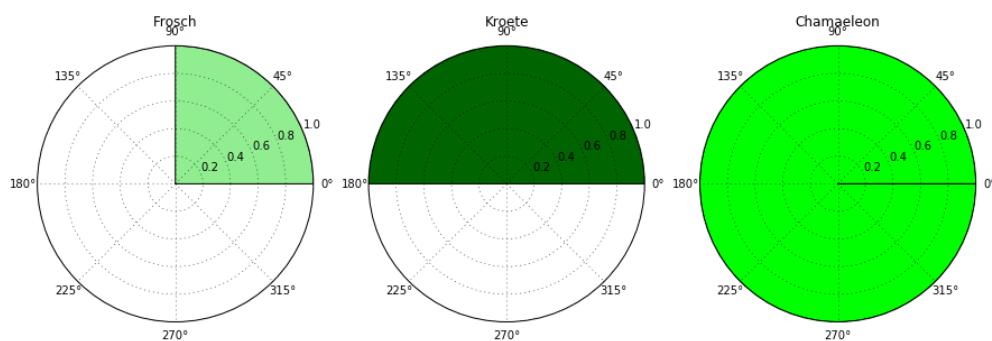
```
In [9]: import matplotlib.pyplot as plt
%matplotlib inline

# definiere eine allgemeine Zeichenfläche
fig = plt.figure(figsize=(15,5))
# füge einen Sub-Plot in die allgemeine Zeichenfläche ein, als ersten in einer ein-reihigen Zeile mit drei Plätzen = (1, 3, 1)
ax1 = fig.add_subplot(1, 3, 1, projection='polar')
# der Plot wird als Polarplot initialisiert
ax1.bar(0, 1, 3.1415/2, bottom = 0.0, color = "lightgreen")
# Für einen Polarplot lautet die Syntax:
# plt.bar(Um wieviel Grad ist die Fläche zentriert, Radius der Fläche, welchen Winkel hat die Fläche)
# Beachte: Die Winkel werden hier nicht in Grad, sondern als Bogenmaß angegeben,
# also  $360^\circ = 2 * \pi \approx 2 * 3.1415$ 
ax1.set_title("Frosch") # Beachte: mit subplot muss die Beschriftung mit set_ erfolgen, also zB set_title

# füge einen weiteren Sub-Plot hinzu, als zweiten in einer ein-reihigen Zeile mit drei Plätzen = (1, 3, 2)
ax2 = fig.add_subplot(1, 3, 2, projection='polar')
ax2.bar(0, 1, 3.1415, bottom = 0.0, color = "darkgreen")
ax2.set_title("Kroete")

# füge noch einen weiteren Sub-Plot hinzu, als dritten in einer ein-reihigen Zeile mit drei Plätzen = (1, 3, 3)
ax3 = fig.add_subplot(1, 3, 3, projection='polar')
ax3.bar(0, 1, 2 * 3.1415, bottom = 0.0, color = "lime")
ax3.set_title("Chamaeleon")
```

Out[9]: <matplotlib.text.Text at 0xc92a0b8>



Histogramme

Histogramme eignen sich, um darzustellen, wie oft ein gewisser Wert in einer Liste vorkommt. Nehmen wir an, wir wissen, dass es eine Bevölkerungsexplosion in unserem Froschteich gab, wir wissen aber nicht genau, in welchem Jahr sie stattgefunden hat. Um diese Frage zu beantworten, fangen wir alle Frösche im Teich und bestimmen ihr Alter. Um unsere Forschungsfrage zu beantworten, stellen wir die resultierende Liste sodann als Histogramm dar:

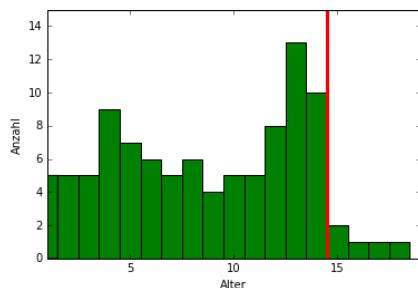
```
In [10]: import matplotlib.pyplot as plt
%matplotlib inline

froschalter=(15,16,15,17,14,13,11,14,14,12,13,1,1,2,3,4,4,5,6,2,1,1,1,3,4,5,14,14,10,10,14,14,11,11,14,
           14,13,12,14,13,12,11,11,10,7,6,8,9,13,13,13,5,4,3,2,4,5,6,7,8,12,12,9,4,12,2,3,4,5,6,7,8,9,10,10,
           6,7,8,4,5,9,8,2,3,4,5,6,7,8,13,13,12,12,13,13,13,19,18)

# Syntax: plt.hist(Liste, Anzahl der Balken, kleinster und größter Wert, Farbe)
plt.hist(froschalter, 18, range = (0.5, 18.5), color = "green")
plt.xlim(1,19)
plt.ylim(0,15)

# zeichne eine rote Vertikale, um die Bevölkerungsexplosion vor ca 15 Jahren zu markieren
plt.plot((14.5, 14.5),(0, 16), color = "red", lw = 3)
plt.ylabel("Anzahl")
plt.xlabel("Alter")
```

Out[10]: <matplotlib.text.Text at 0xaf253c8>



Wir erkennen aus dem Histogramm, dass viele Frösche im Teich aktuell 14 Jahre alt sind, aber nur sehr wenige 15 Jahre. Vor 14 Jahren wurden also wesentlich mehr Frösche geboren, als vor 15 Jahren. Die Bevölkerungsexplosion fand also offensichtlich vor ca 15 Jahren statt.

Box- und Violinplots

Box- und Violinplots eignen sich, ähnlich wie Histogramme, dazu Verteilungen darzustellen. Boxplots erlauben es überdies, mehrere Verteilungen miteinander zu vergleichen. Sie zeigen den Mittelwert einer Verteilung (rote Linie), den Bereich, in dem der Großteil der Werte liegt (schwarze Box), aber auch die Extremwerte und Ausreißer.

Sehr ähnlich sind Violinplots, mit dem Unterschied, dass sie detailliertere Informationen zur Verteilung liefern. Ähnlich dem Histogramm zeigen sie (stark geglättet), in welchen Bereichen viele, und in welchen wenige Werte vorliegen.

Im folgenden vergleichen wir das Gewicht der Frösche in 3 verschiedenen Teichen, einmal mit Box- und einmal mit Violinplots:

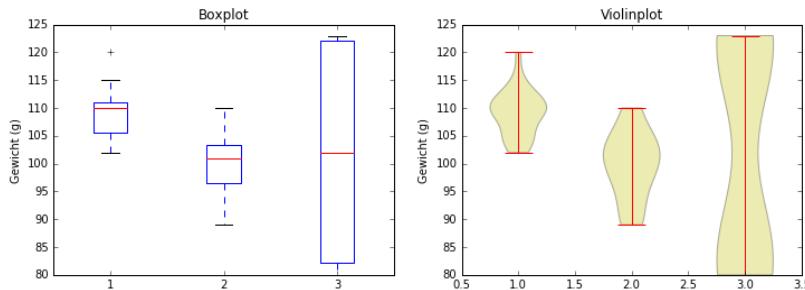
```
In [11]: import matplotlib.pyplot as plt
%matplotlib inline

teich1=(110,105,106,104,110,115,113,112,113,103,104,110,109,108,105,102,110,110,111,109,111,120)
teich2=(98,99,100,102,103,103,104,102,110,99,95,98,97,93,98,110,105,106,102,89)
teich3=(120,122,123,122,120,122,123,122,119,88,85,80,81,82,83,82,84,85)

fig = plt.figure(figsize=(12,4))
# erster Plot
ax1 = fig.add_subplot(1, 2, 1)
ax1.boxplot((teich1, teich2, teich3))
ax1.set_title("Boxplot")
ax1.set_ylabel("Gewicht (g)")

# zweiter Plot
ax2 = fig.add_subplot(1, 2, 2)
ax2.violinplot((teich1, teich2, teich3))
ax2.set_title("Violinplot")
ax2.set_ylabel("Gewicht (g)")
```

Out[11]: <matplotlib.text.Text at 0xd6aa898>



Heatmaps

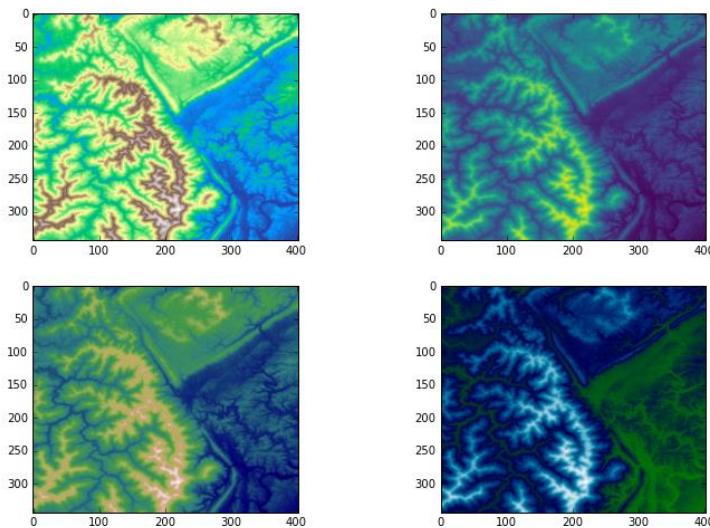
Beim plotten dreidimensionaler Daten möchte man in der Regel jedem x/y-Daten-Paar einen weiteren z-Wert zuweisen, so wie etwa den Geo-Koordinaten auf topologischen Karten eine Seehöhe zugewiesen wird. Um diese Plots eindrucksvoll darzustellen, importieren wir im Folgenden vorgefertigte Beispieldaten aus dem `matplotlib.cbook` mit dem Befehl `get_sample_data`.

```
In [12]: import matplotlib.pyplot as plt
%matplotlib inline
# importiere Beispieldaten
from matplotlib.cbook import get_sample_data
import numpy as np

data = np.load(get_sample_data('jacksboro_fault_dem.npz')) # liest einen Beispieldatensatz ein
z = data['elevation'] # speichert die Höheninformation des Datensatzes unter z

fig = plt.figure(figsize=(12, 8))
ax1 = fig.add_subplot(2, 2, 1)
ax1.imshow(z, cmap = plt.cm.terrain) # colormap terrain: z.B. für Karten
ax2 = fig.add_subplot(2, 2, 2)
ax2.imshow(z, cmap = plt.cm.viridis) # colormap viridis: für abstrakte Daten
ax3 = fig.add_subplot(2, 2, 3)
ax3.imshow(z, cmap = plt.cm.gist_earth) # colormap earth: für realistische Karten
ax4 = fig.add_subplot(2, 2, 4)
ax4.imshow(z, cmap = plt.cm.ocean) # colormap ocean: Blautöne für Meeresdarstellungen
```

```
Out[12]: <matplotlib.image.AxesImage at 0xdcdcf278>
```



Weitere Visualisierungen

Neben den hier vorgestellten Möglichkeiten gibt es noch viele mehr, um Daten zu visualisieren. Einen guten Überblick liefert die Plot-Gallerie von `matplotlib` unter <https://matplotlib.org/gallery.html>, die zu jedem Plot auch den Code bereitstellt, mit dem der Plot erzeugt wird.

Darüber hinaus gibt es auch andere spezialisierte Python-Pakete, die sich der Datenvisualisierung widmen. Beispiele sind `seaborn`, zum Darstellen von statistischen Daten, oder `bokeh`, das interaktive Visualisierungen im Browser erlaubt.

Zusammenfassung

Linienplots

Linienplots sind einfache Plots, die immer dann verwendet werden, wenn eine Größe eindeutig einer anderen Größe zugeordnet werden soll.

Scatterplots

Mit Scatterplots werden Datenpaare (z.B. Alter und Gewicht) dargestellt, die ohne eindeutige Zuordnung zusammengehören. Beispiel: ein drei Jahre alter Frosch muss nicht immer exakt 111 Gramm haben.

Histogramme, Box- und Violinplots

Mit Histogrammen, sowie Box- oder Violinplots werden Verteilungen dargestellt. Während ein Histogramm eine exakte Verteilung darstellt, lassen sich mit Box- und Violinplots mehrere Verteilungen miteinander vergleichen. Boxplots zeigen wichtige Charakteristika deutliche (Mittelwert, Standardabweichung, Ausreißer), Violinplots zeigen mehr Details der Verteilung.

Subplots

Subplots erlauben es, mehrere Grafiken nebeneinander darzustellen. Subplots werden immer mit drei Zahlen bezeichnet: `subplot(a, b, c)`, wobei die Grafiken in einer Matrix mit a Zeilen und b Spalten organisiert sind und der jeweils gerade angesprochene Plot den Index c hat.

Kapitel 13 – Netzwerke

Eine besonders fruchtbare Form, die Komponenten eines Systems in ihren Wechselwirkungen zu untersuchen, bietet die Netzwerkforschung. Ähnlich wie die Systemwissenschaft findet die Netzwerkforschung ihren Gegenstand in unterschiedlichsten Bereichen. Computer, die miteinander kommunizieren, bilden Computernetzwerke. Menschen, die miteinander in Beziehung stehen, bilden soziale Netzwerke. Tiere und Pflanzen, die sich einen Lebensraum teilen, bilden ökologische Netzwerke. All diese Netzwerke, obwohl sie eigentlich grundverschieden scheinen, haben gemeinsame Eigenschaften und folgen ähnlichen Regeln.

In den letzten Jahren gewinnt der Netzwerkbegriff in den Wissenschaften enorm an Bedeutung. Es ist deshalb kaum verwunderlich, dass auch Python mittlerweile spezialisierte Pakete bereithält, die sich der Darstellung und Analyse von Netzwerken widmen. Wir werden uns im Folgenden eines dieser Paket etwas genauer ansehen.

Was sind Netzwerke?

Von Netzwerken spricht man immer dann, wenn Objekte als **Knoten** (oder auf Englisch auch **nodes** oder **vertices**) vorliegen, die mit anderen solchen Objekten in irgendeiner Weise verbunden sind. Die **Verbindungen** werden im Englischen **links** oder auch **edges** genannt.

Charakterisiert werden Netzwerke durch ihre Verbindungsstruktur, also durch die Information darüber, welche Knoten mit welchen anderen Knoten auf welche Weise verbunden sind. Aus dieser Verbindungsstruktur ergibt sich eine Vielzahl unterschiedlicher Netzwerktypen, von denen wir einige weiter unten detaillierter besprechen.

NetworkX

Zunächst sehen wir uns nun aber einige Grundlagen des Python-Pakets NetworkX an, das auf die Darstellung und Analyse von Netzwerken spezialisiert ist und bereits im installierten Anaconda-Paket enthalten ist.

Für den Anfang betrachten wir ein sehr einfaches Netzwerk: Das Netzwerk soll aus drei Personen bestehen: Alice, Bob und Carol. Diese drei Personen sind die Knoten (**nodes**) unseres Netzwerks. Alle sind miteinander befreundet, was wir in unserem "Freundschaftsnetzwerk" durch Verbindungen (**edges**) darstellen.

Wir beginnen damit, das Paket NetworkX zu importieren und zunächst ein leeres Netzwerk anzulegen. Dieses füllen wir sodann mit **nodes** und **edges**:

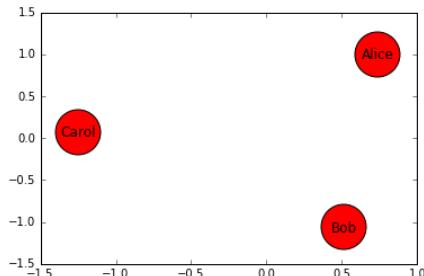
```
In [1]: import networkx as nx  
G = nx.Graph() # erstelle ein neues Netzwerk mit dem Namen G
```

Im nächsten Schritt fügen wir drei Knoten hinzu: Alice, Bob und Carol.

```
In [2]: G.add_node("Alice") # füge dem Netzwerk G den Knoten "Alice" hinzu  
G.add_node("Bob")  
G.add_node("Carol")
```

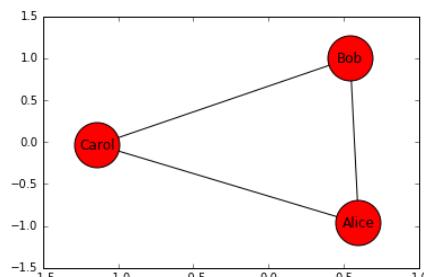
Um das Netzwerk grafisch darzustellen, benötigen wir abermals das Paket **matplotlib**, das sehr gut mit networkx harmoniert.

```
In [3]: import matplotlib.pyplot as plt  
%matplotlib inline  
  
nx.draw_networkx(G, node_size = 1600) # zeichne das Netzwerk G. Die Knoten sollen die Größe 1600 haben.
```



Wir sehen: die Knoten werden korrekt dargestellt. Was noch fehlt, sind die Verbindungslien zwischen den Knoten. Diese werden mit dem Befehl **G.add_edge(von, zu)** hinzugefügt:

```
In [4]: G.add_edge("Alice","Bob") # füge dem Netzwerk G eine neue Verbindung zwischen Alice und Bob hinzu.  
G.add_edge("Bob","Carol")  
G.add_edge("Carol","Alice")  
  
nx.draw_networkx(G, node_size = 1600)
```

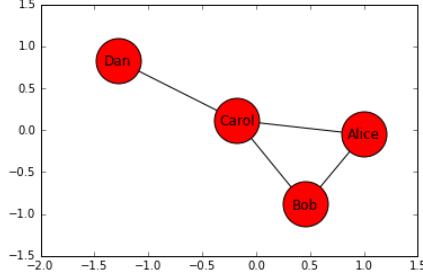


Somit ist unser (sehr einfaches) Netzwerk korrekt dargestellt. Wenn wir die obige Code-Zelle mehrmals ausführen, fällt auf, dass die Positionen der einzelnen Knoten jedesmal neu vergeben werden. Sie werden zufällig generiert. In dieser Darstellung haben sie keine Bedeutung, wichtig ist nur, wer mit wem verbunden ist, nicht an welchem Ort sie sich befinden.

Fügen wir unserem Netzwerk eine weitere Person hinzu: Dan. Dan ist nur mit Carol befreundet, kennt sonst aber niemanden aus dem Netzwerk. Wir machen also nur eine Verbindung zwischen Carol und Dan:

```
In [5]: G.add_node("Dan")
G.add_edge("Carol", "Dan")

nx.draw_networkx(G, node_size = 1600)
```

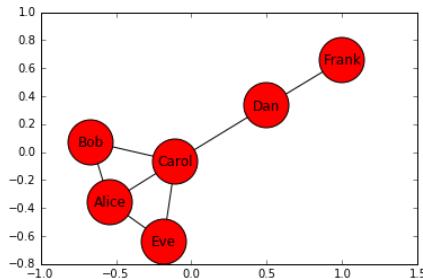


Um unser Netzwerk noch etwas zu erweitern, fügen wir noch zwei weitere Personen hinzu: Eve ist eine Freundin von Alice und Carol. Frank ist nur mit Dan befreundet:

```
In [6]: G.add_node("Eve")
G.add_edge("Eve", "Alice")
G.add_edge("Eve", "Carol")

G.add_node("Frank")
G.add_edge("Frank", "Dan")

nx.draw_networkx(G, node_size = 1600)
```



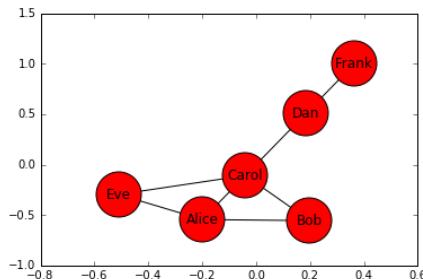
Dieses soziale Netzwerk ist nun schon hinreichend komplex, um daran unterschiedliche Darstellungsarten auszuprobieren. Bisher waren die Positionen der Knoten in der Grafik dem Zufall überlassen. Man kann sie aber auch mit so genannten "Layouts" arrangieren. Die Knoten werden dann nach bestimmten Regeln angeordnet, was spezielle Netzwerkeigenschaften verdeutlichen kann, oder einfach nur die Netzwerke generell übersichtlicher gestaltet.

Das Spring-Layout

Eines der übersichtlichsten Layouts (und deswegen auch die Standardeinstellung in NetworkX) ist das Spring-Layout. Es heißt so, weil die Verbindungslien als Federn (*springs*) interpretiert werden, was zur Folge hat, dass zwei verbundene Knoten, die sich zu nahe sind, vom Algorithmus, der das Netzwerk generiert, auseinander gezogen werden, und wenn sie zu weit voneinander entfernt sind, zusammengezogen werden. Zumeist liefert dies ein recht übersichtliches Bild.

Layouts werden innerhalb des `draw`-Befehls mit dem Argument `pos` definiert (Merke: wir haben das Netzwerk `G` oben bereits in all seinen Bestandteilen definiert und brauchen es deshalb in Folge nur mehr in seinen unterschiedlichen Layouts aufzurufen):

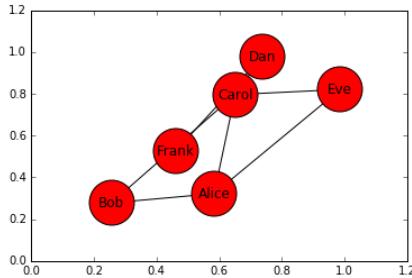
```
In [7]: # nx.spring_layout(G) errechnet das Springlayout für das Netzwerk
nx.draw_networkx(G, node_size = 1600, pos = nx.spring_layout(G))
```



Das Random-Layout

Etwas weniger übersichtlich ist das Layout "random". Hier werden die Positionen der Knoten absolut zufällig gewählt, was dazu führt, dass die Verbindungslien kreuz und quer laufen können. Der Vorteil dieses Layouts ist aber, dass es kaum Rechenzeit benötigt und somit gern bei großen Netzwerken benutzt wird, bei denen man den einzelnen Verbindungslien ohnehin nur schwer folgen kann.

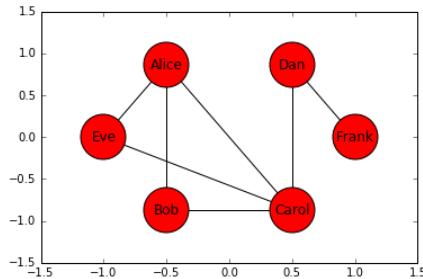
```
In [8]: nx.draw_networkx(G, node_size = 1600, pos = nx.random_layout(G))
```



Das Circular-Layout

Beim Circular-Layout werden alle Knoten in einem Kreis angeordnet. Hier liegt der Vorteil darin, schnell erkennen zu können, wer mit wem verbunden ist.

```
In [9]: nx.draw_networkx(G, node_size = 1600, pos = nx.circular_layout(G))
```



Netzwerkanalyse

Was lässt sich nun mit einem solchen Netzwerk machen? Zunächst könnten wir versuchen, bestimmte Netzwerkeigenschaften festzustellen. Für unser soziales Netzwerk könnten wir zum Beispiel herausfinden, wie die Freundschaften im Netzwerk verteilt sind. Wir wollen also wissen, wie viele Verbindungslien die einzelnen Knoten haben. Im Englischen spricht man hierbei vom **degree** eines Knoten. Wir erhalten diese Information mit dem Befehl `nx.degree(G)`:

```
In [10]: nx.degree(G)
Out[10]: {'Alice': 3, 'Bob': 2, 'Carol': 4, 'Dan': 2, 'Eve': 2, 'Frank': 1}
```

Die Antwort ist so zu lesen: Der Knoten mit dem Namen "Alice" hat degree 3 (also drei Freunde), "Bob" hat degree 2 (zwei Freunde), und so weiter.

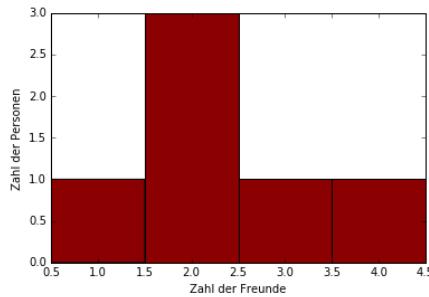
Oft ist man aber nur an den Zahlen selbst interessiert, nicht unbedingt daran, wie die Knoten heißen. Dazu lässt sich der Befehl `.values()` wie folgt hinzufügen:

```
In [11]: nx.degree(G).values()
Out[11]: [1, 2, 3, 2, 2, 4]
```

Als Antwort ergibt sich eine Liste, die sich recht einfach in ein Histogramm verwandeln lässt:

```
In [12]: plt.hist(nx.degree(G).values(), 4, range = (0.5, 4.5), color = "darkred")
plt.xlabel('Zahl der Freunde')
plt.ylabel('Zahl der Personen')

Out[12]: <matplotlib.text.Text at 0x1b890860>
```



Das Histogramm zeigt, dass es drei Personen gibt, die genau zwei Freunde haben und jeweils eine Person, die einen, drei und vier Freunde hat. Richtig interessant wird eine solche Analyse erst mit großen Netzwerken mit einigen hunderten oder tausenden Knoten. Solche Netzwerke werden in der Regel nicht mehr manuell eingegeben. Es wird also nicht mehr jeder Knoten und jede Verbindung einzeln definiert, sondern ein so genannter Netzwerkgenerator verwendet.

Netzwerkgeneratoren

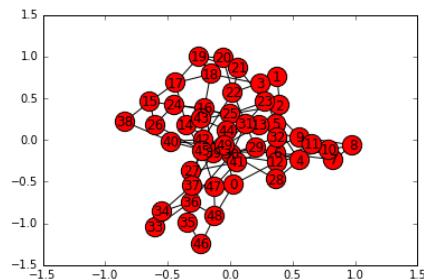
Netzwerkgeneratoren eignen sich insbesondere dazu, bestimmte Archetypen von Netzwerken zu erstellen, die in der Natur, aber auch in der Gesellschaft oder in technischen Zusammenhängen immer wieder auftreten. Diese Archetypen zeichnen sich durch ganz bestimmte Netzwerktopologien aus, die in verschiedenen Kontexten in ähnlicher Weise auftreten.

Das Small-World Netzwerk

Einer der verbreitetsten Netzwerk-Archetypen ist das so genannte Small-World-Netzwerk, das seinen Namen einem berühmt gewordenen Experiment von Stanley Milgram verdankt (siehe: http://systems-sciences.uni-graz.at/etextbook/networks/networks_2.html). Es zeichnet sich durch relativ geringe durchschnittliche Vernetzungsdichte aus, die allerdings von Regionen hoher Vernetzungsdichte unterbrochen ist, wobei diese Regionen ihrerseits über kurze Pfade mit anderen solchen dichten Regionen verbunden sind. Die meisten Knoten in solchen Netzwerken haben nur recht wenige Verbindungen. Einige wenige haben dagegen viele und ganz wenige Knoten haben sehr viele Verbindungen. Diese sehr wenigen, überaus gut vernetzten Knoten werden als **Hubs** bezeichnet. Viele soziale Netzwerke, aber auch das Internet oder viele Gen-Netzwerke haben diese Small-World-Struktur.

Der Python-Befehl zum Erzeugen solcher Netzwerke ist ein wenig sperrig. Die Namen der Entwickler des Algorithmus zur Generierung dieses Netzwerktyps findet in ihm Platz: `connected_watts_strogatz_graph`. Als Argumente braucht dieser Generator die Anzahl der Knoten im Netzwerk, die mittlere Anzahl der Verbindungen pro Knoten, und eine Zahl, die festlegt wie groß die Hubs werden dürfen, die entstehen.

```
In [13]: # erzeugt ein Small-world-Netzwerk mit 50 Knoten, und durchschnittlich 5 Verbindungen pro Knoten
# die dritte Zahl gibt die Chance an, dass ein Knoten mit einem seiner Verbindungen an einen Hub ansetzt.
G = nx.connected_watts_strogatz_graph(50, 5, 0.40)
nx.draw_networkx(G)
```



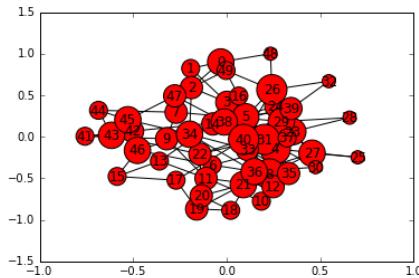
Durch die Größe des Netzwerks ist die Darstellung nun ein wenig unübersichtlich geworden, und wir können nicht mehr ganz klar erkennen, wer mit wem verbunden ist, bzw. welcher Knoten viele Verbindungen hat und welcher wenige. Übersichtlicher könnte dies werden, wenn wir die Größe der Knoten in der Darstellung davon abhängig machen, welchen `degree` sie haben:

```
In [14]: import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

G = nx.connected_watts_strogatz_graph(50, 5, 0.40)
deg = nx.degree(G).values()
size = [] # Leere Liste wird angelegt

# für jeden Wert in der Degreeliste wird ein Eintrag in die Sizeliste angefügt
for wert in deg:
    size.append(50 * wert**1.5)

nx.draw_networkx(G, node_size = size)
```



Nun sieht man anhand der Größe der Knoten, dass einige Knoten deutlich mehr Verbindungen haben als andere. Diese Hubs des Netzwerks können großen Einfluss auf das Gesamtsystem haben.

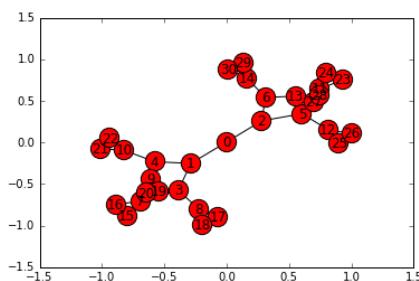
Das hierarchische Netzwerk

In hierarchischen Netzwerken hat zumeist nur ein Knoten eine besondere Stellung. Dieser zentralste Knoten steht in Verbindung zu weniger wichtigen Knoten, die ihrerseits in Verbindung zu weiteren, noch weniger zentralen Knoten stehen.

Solche hierarchischen Strukturen finden sich vielfach in Organisationen, aber auch zum Beispiel in Nahrungsketten im Tierreich. Der NetworkX-Befehl zum Erstellen von hierarchischen Netzwerken lautet `balanced_tree`. Als Argumente werden die Anzahl der Knoten-Verbindungen in die jeweils untere Ebene der Hierarchie und die Zahl der Ebenen benötigt.

```
In [15]: import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

# ein hierarchisches Netzwerk mit 2 Verbindungen "nach unten" und 4 Ebenen
G = nx.balanced_tree(2, 4)
nx.draw_networkx(G)
```

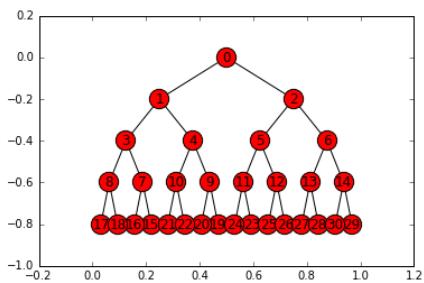


Besonders übersichtlich wird die Darstellung eines hierarchischen Netzwerks mit speziellem Layout. Da der genaue Aufbau der Funktion für dieses Layout den Rahmen dieses Kapitels sprengt, behandeln wir ihn im Folgenden als "Blackbox". Das heißt, wir zeigen die Funktion und ihren Output im Folgenden, erklären aber nicht wie sie genau funktioniert.

```
In [16]: import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

#BLACKBOXTFUNKTION, die man nicht verstehen muss
def hierarchy_pos(G, root, width=1., vert_gap = 0.2, vert_loc = 0, xcenter = 0.5,
                  pos = None, parent = None):
    if pos == None:
        pos = {root:(xcenter,vert_loc)}
    else:
        pos[root] = (xcenter, vert_loc)
    neighbors = G.neighbors(root)
    if parent != None:
        neighbors.remove(parent)
    if len(neighbors) != 0:
        dx = width/len(neighbors)
        nextx = xcenter - width/2 - dx/2
        for neighbor in neighbors:
            nextx += dx
            pos = hierarchy_pos(G,neighbor, width = dx, vert_gap = vert_gap,
                                 vert_loc = vert_loc-vert_gap, xcenter = nextx, pos = pos,
                                 parent = root)
    return pos
# ENDE DER BLACKBOXTFUNKTION

G=nx.balanced_tree(2, 4)
nx.draw_networkx(G, pos = hierarchy_pos(G, 0))
```



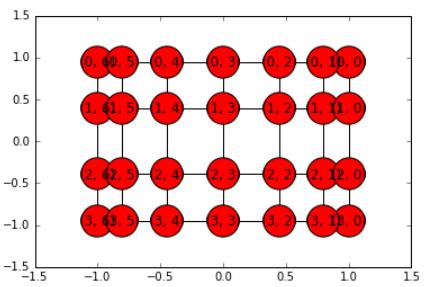
Das Grid-Netzwerk

Auch eine einfache Gitterstruktur (Grid) lässt sich als Netzwerk betrachten. Solche Strukturen treten zum Beispiel in Kristallen auf, aber auch überall sonst, wo Objekte bevorzugt mit ihren nächsten Nachbarn interagieren.

Der Befehl zum Erstellen von Grid-Netzwerken lautet `grid_2d_graph` und braucht als Argumente die Lnge und die Breite des Gitters.

```
In [17]: import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

G = nx.grid_2d_graph(4, 7) # erzeugt ein 4 mal 7 Grid
# Spectral-Layout für passende Grid-Form
nx.draw_networkx(G, pos = nx.spectral_layout(G), node_size = 800)
```



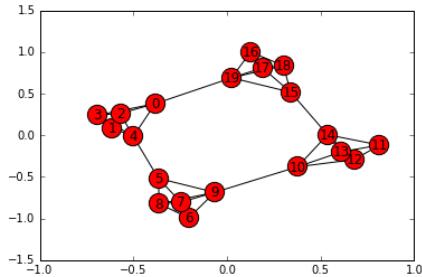
Das Caveman-Netzwerk

Ein so genanntes Caveman-Netzwerk zeichnet sich durch mehrere dichter vernetzte Knoten-Gruppen aus, so genannte **Cluster**, die untereinander nur wenig miteinander verbunden sind.

Der Befehl zum erzeugen solcher Netzwerke lautet `connected_caveman_graph` und benötigt als Argumente die Zahl der Cluster und die Größe der Cluster.

```
In [18]: import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

G=nx.connected_caveman_graph(4, 5) # Netzwerk aus 4 Clustern mit je 5 Knoten
nx.draw_networkx(G)
```



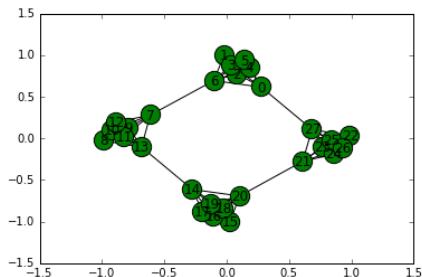
Beispiel: Die Ausbreitung eines Computer-Virus

Versuchen wir nun, unser bisheriges Wissen über Netzwerke in einem einfachen Anwendungsbeispiel zu nutzen. Wir erstellen ein fiktives Computer-Netzwerk und untersuchen, wie sich ein Software-Virus in diesem Netzwerk ausbreiten würde.

Als Netzwerktypus wählen wir ein Caveman-Netz, in dem vier über das Internet verbundene Unternehmen jeweils sieben Computer in einem firmeneigenen Intranet betreiben.

```
In [19]: import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

G = nx.connected_caveman_graph(4, 7)
# speichere die Knoten-Positionen, um in allen folgenden Darstellungen das selbe Netzwerk zu sehen
position = nx.spring_layout(G)
nx.draw_networkx(G, pos = position, node_color = "green")
```



Mit dem folgenden Code fügen wir für jeden Knoten ein Attribut hinzu, in dem später gespeichert wird, ob der jeweilige Computer vom Software-Virus infiziert ist oder nicht. Wir iterieren dazu mit einer Schleife über die Zahl der Knoten des Gesamtnetzwerks (`G.number_of_nodes()`), definieren dabei für jeden Knoten das neue Attribut "infiziert" und setzen es auf "nein". Dies geschieht mit `G.node[it]["infiziert"] = "nein"`.

```
In [20]: # iteriere über alle Knoten und gib ihnen ein neues Attribut namens "infiziert"
# setze dieses neue Attribut auf "nein"
for it in range(G.number_of_nodes()):
    G.node[it]["infiziert"] = "nein"
```

Wir überprüfen diese Operation, indem wir einen beliebigen Knoten - den Knoten mit dem Index 0 - auf seine Attribute befragen:

```
In [21]: G.node[0] # frage ab welche Daten im Knoten mit Index 0 gespeichert sind
Out[21]: {'infiziert': 'nein'}
```

In ähnlicher Weise lassen sich beliebig viele Daten in einem Knoten speichern. Hier begnügen wir uns aber mit nur einem Attribut.

Im nächsten Schritt infizieren wir einen Knoten (einen Computer) mit dem Software-Virus, zum Beispiel den Knoten mit dem Index 8:

```
In [22]: # setze "infiziert" bei Knoten 8 auf "ja":
G.node[8]["infiziert"] = "ja"
# frage ab welche Daten im Knoten 8 gespeichert sind
G.node[8]
Out[22]: {'infiziert': 'ja'}
```

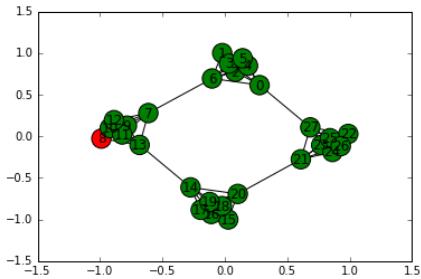
Wenn wir nun das Gesamtnetzwerk auf Virusbefall überprüfen sollen, würden wir wohl alle Knoten mithilfe einer For-Schleife befragen:

```
In [23]: for it in range(G.number_of_nodes()):
    print(G.node[it]), # das Komma am Ende des Print-Befehls schreibt Output fortlaufend in die selbe Zeile
```

Übersichtlich ist das so natürlich nicht. Schöner wäre es, diese Information mit der Farbe der Knoten widerzugeben. Wir legen dazu eine neue Liste an, in der die Farben der Knoten gespeichert werden: grün für "nicht infiziert" und rot für "infiziert".

```
In [24]: farben = []
for it in range(G.number_of_nodes()):
    # frage den Wert des Attributes "infiziert" ab:
    if G.node[it]["infiziert"] == "ja":
        farben.append("red")
    if G.node[it]["infiziert"] == "nein":
        farben.append("green")

# anstelle der einen Farbe "Grün", plotten wir nun mit der Liste "farben".
# für jeden Knoten gibt es genau einen Eintrag
nx.draw_networkx(G, pos = position, node_color = farben)
```



In dieser Darstellung ist der infizierte Knoten klar erkennbar.

Wie würde sich ein solcher Software-Virus ausbreiten? Die Epidemiologie kennt viele Modelle zur Infektionsausbreitung. Wir verwenden hier zunächst das einfachste: In jedem Zeitschritt werden alle Knoten infiziert, die mit einem infizierten Knoten verbunden sind.

Dafür nutzen wir einen neuen Befehl: `G.neigbors(n)` liefert uns alle Nachbarn (also verbundene Knoten) eines Knoten n .

Wenn wir nun fünf Zeitschritte dieses einfachen Infektionsmodells simulieren wollen, ergeben sich einige komplexe Verschachtelungen von If-Abfragen und For-Schleifen. Wir sehen uns diese im folgenden Code-Segment genauer an:

```
In [25]: # definiere eine allgemeine Zeichenfläche
fig = plt.figure(figsize=(15, 10))

n = 1 # Zähler zum Mitzählen der Subplots
# erster Subplot
ax = fig.add_subplot(2, 3, n)
nx.draw_networkx(G, pos = position, node_color = farben)

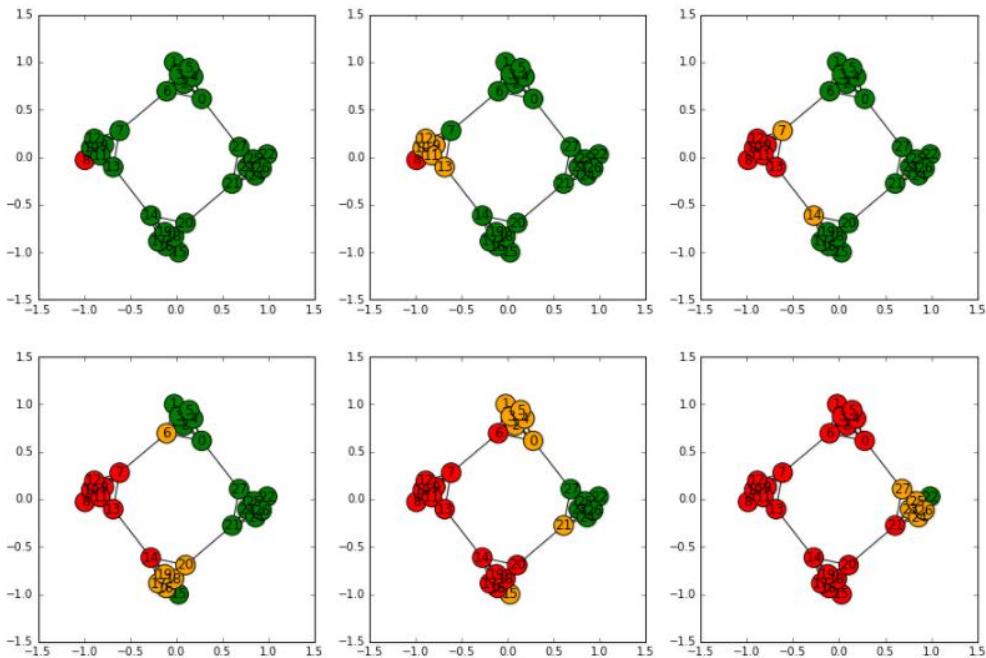
# simuliere 5 Zeitschritte
for zeit in range(5):
    n += 1 # setze Zähler zum Zählen der Subplots um 1 hinauf
    # for-Schleife über alle Knoten
    # die Laufvariablen nennen wir inf, da wir hier nur die infizierten Knoten betrachten
    for inf in range(G.number_of_nodes()):
        # wenn der Knoten infizierte ist,
        if G.node[inf]["infiziert"] == "ja":
            # gehen wir in einer weiteren Schleife über alle Nachbarn des infizierten Knoten:
            for opfer in G.neighbors(inf):
                # und wenn dieser Knoten noch nicht infiziert ist
                if G.node[opfer]["infiziert"] == "nein":
                    # wird sie in diesem Zeitschritt neu infiziert:
                    G.node[opfer]["infiziert"] = "neu"
                    # Ende der if-Abfrage
            # Ende der for-Schleife über die Opfer
        # Ende der if-Abfrage, die feststellt, ob ein Knoten infiziert war
    # Ende der for-Schleife über alle Knoten
    # hier sind wir wieder in der for-Schleife über 5 Zeitschritte
```

```

farben = []
for it in range(G.number_of_nodes()):
    # die neu infizierten Knoten bekommen die Farbe Orange, damit wir die Ausbreitung besser verfolgen können
    if G.node[it]["infiziert"] == "ja":
        farben.append("red")
    if G.node[it]["infiziert"] == "nein":
        farben.append("green")
    if G.node[it]["infiziert"] == "neu":
        farben.append("orange")
# neuer Subplot
ax = fig.add_subplot(2, 3, n)
nx.draw_networkx(G, pos = position, node_color = farben)

# hier ändern wir nun alle "neu" infizierten Knoten auf infizierte ("ja"),
# damit sie im nächsten Schritt rot statt orange erscheinen und ihrerseits weitere Knoten infizieren
for it in range(G.number_of_nodes()):
    if G.node[it]["infiziert"] == "neu":
        G.node[it]["infiziert"] = "ja"

```



Mit diesem sehr einfachen Modell können wir also ansatzweise bereits Aspekte der Ausbreitung eines Netzwerk-Virus analysieren. Versuchen sie selbst, dieses Modell zu erweitern, zum Beispiel größere oder andere Netzwerke zu generieren, oder die Knoten mit zusätzlichen Attributen - Antivirenprogramme, Firewalls etc. - auszustatten.

Zusammenfassung

Netzwerke

In der Netzwerkforschung werden Objekte als Knoten (*nodes*) aufgefasst, die mit anderen Knoten verbunden sind. Entscheidend sind hierbei nicht so sehr die speziellen Eigenschaften der Knoten, als vielmehr die Verbindungsstruktur (die Topologie eines Netzwerks), die sich aus der Zahl der Verbindungen (*links*, *edges*) der Knoten - dem so genannten *degree* - und aller Nachbarknoten ergibt. Bestimmte Netzwerktypen, wie etwa Small-World- oder hierarchische Netzwerke, liegen Phänomene in unterschiedlichsten Kontexten zugrunde.

Netzwerke in Python

In Python steht das Paket Networkx zur Darstellung und Analyse von Netzwerken zur Verfügung.

Manuelles Generieren von Netzwerken

Mit dem Befehl `G = nx.Graph()` wird ein neues, leeres Netzwerk mit dem Namen G generiert. Neue Knoten können mit dem Befehl `G.add_node("Name_des_neuen_Knoten")` hinzugefügt werden und mit `G.add_edge("Knoten_1", "Knoten_2")` mit anderen Knoten verbunden werden.

Netzwerkgeneratoren

Um spezielle Netzwerktypen zu erstellen, stehen so genannte Netzwerkgeneratoren bereit, die teils zufällige, teils deterministische Netzwerke generieren. Häufig verwendete Generatoren sind die für Small-World-Netzwerke (`connected_watts_strogatz_graph`) und für hierarchische Netzwerke (`balanced_tree`).

Programmieren mit Netzwerken

Um über alle Knoten in einem Netzwerk zu iterieren, steht der Schleifen-Befehl `for it in range(G.number_of_nodes()):` bereit. Um einem Knoten ein Attribut zu zuweisen, das zusätzliche Informationen enthält, steht der Befehl `G.node["Name_des_Knoten"]["Name_des_Attributs"] = "Wert_des_Attributs"` bereit.

Kapitel 14 – Die Stabilität von Systemen

Wie einleitend betont, steht der Begriff "System" für die analytische (d.h. wissenschaftliche) Betrachtung eines Phänomens, von dem bekannt ist, dass es in der Zusammenwirkung mehrerer Einzelkomponenten entsteht ("emergiert"), wobei diese Einzelkomponenten (auf Microebene) nicht notwendig das selbe oder ein ähnliches Verhalten zeigen, wie das System (auf Macroebene) als solches. Systemwissenschaftlich spricht man diesbezüglich davon, dass das System ein **Eigenverhalten** zeigt.

Dieses Eigen- (oder eben System-)verhalten kann stabil sein (d.h. es kann gleich bleiben), auch wenn sich die Komponenten, die dafür verantwortlich sind, verändern. Wasser zum Beispiel ist zwischen ca. 0 und ca. 100 Grad Celsius gleichbleibend flüssig, obwohl sich die Geschwindigkeit der Komponenten, also der Wasser- und Sauerstoffatome, in diesem Bereich mit der Temperatur verändert.

Umso überraschender kann dann in diesem Zusammenhang ein so genannter **Phasenübergang** sein, also eine Veränderung des Eigenverhaltens, beim Wasser etwa der Übergang von flüssig zu gas- oder fest-förmig. Die Komponenten verändern dabei ihr Verhalten nicht wesentlich, (d.h. eigentlich verändert sich ihr "Veränderungsverhalten" nicht wesentlich) - die Teilchen werden weiterhin schneller, bzw. langsamer - , das System als Ganzes zeigt aber ab einem bestimmten Punkt - einem so genannten Kippunkt (engl. "tipping point") - eine deutlich andere Qualität.

Weil diese "tipping points" oft überraschend auftreten und das Systemverhalten wesentlich bestimmen, besteht ein zentraler Teil der systemwissenschaftlichen Analyse im Versuch, die **Stabilität von Systemen** zu erkunden, also zu verstehen zu versuchen, unter welchen Bedingungen ein System seinen Zustand bewahrt und wann es in einen anderen Zustand kippt. **Stabilitätsanalyse** ist ein wesentlicher und komplexer Bestandteil der modernen Systemwissenschaften.

Dieser Teil des Skripts führt mit Hilfe von Python in erste einfache Grundlagen der Stabilitätsanalyse ein. Das erste dafür gewählte Beispiel ist der so genannte

Beispiel 1: Allee-Effekt

(siehe auch: <http://systems-sciences.uni-graz.at/etextbook/sw1/attractors.html>)

Bei vielen Lebewesen hängt der Erfolg der Fortpflanzung nicht nur von der Üppigkeit des Nahrungsangebots ab, sondern zum Beispiel auch von der Dichte ihrer Population. Wenn Bären zum Beispiel in einer Region nicht hinreichend oft auf Fortpflanzungspartner stoßen, einfach weil die Bären in dieser Region zu weit gestreut herumziehen, so wird die Population schrumpfen und eventuell aussterben. Wenn umgekehrt zuviele Bären ein Gebiet bevölkern, könnte die Fortpflanzung auch nicht optimal klappen, weil sie sich zuviel Konkurrenz machen.

Dieser Effekt lässt sich mit Hilfe eines kleinen Zusatzterms zum in Kapitel 2 dieses Skripts besprochenen **Logistischen Wachstum** modellieren. Dieser Term ist $\left(\frac{N-a}{K}\right)$ und wird einfach mit dem Ausdruck für Logistisches Wachstum multipliziert. Die entsprechende Veränderungsrate (bzw. Wachstumsrate) sieht dann wie folgt aus:

$$\frac{dN}{dt} = \gamma * N * \left(1 - \frac{N}{K}\right) * \left(\frac{N-a}{K}\right)$$

Wir erinnern uns: der Parameter γ bezeichnet die eigentliche Wachstumsrate der Population, der Term $N * \left(1 - \frac{N}{K}\right)$ bezeichnet eine von der Umweltkapazitätsgrenze K abhängige, zunächst exponentielle und dann gebremste Entwicklung, und der hier neu hinzugekommene Parameter a gibt einen so genannten "Aussterbeschwellenwert" ("extinction threshold") an - einen "tipping point" -, unter dem die Bevölkerungsgröße sinkt, und über dem sie bis K anwächst. Mit Python als *Differenzengleichung* dargestellt wird dies anschaulich.

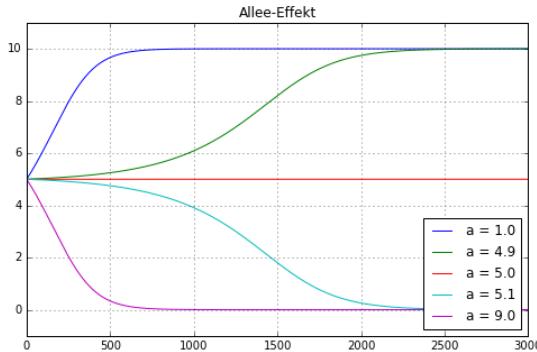
```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(8,5))
fig.add_subplot(1,1,1)

# definiere die Wachstumsrate
gamma = 0.1
# definiere ein Intervall mit verschiedenen a-Werten
A = [1, 4.9, 5, 5.1, 9]
T = range(3000)
# der Zeitschritt (= die Differenz)
dt = 0.1
# die Wachstumsgrenze (Beachte: da durch K dividiert wird, muss in Python-2 ein reeller Wert, also 10.0 und nicht 10
# definiert werden, daher 10. und nicht 10)
K = 10.
# Anfangswert
aw = 5.
# iteriere über die a-Werte
for a in A:
    N = [aw] # bilde eine Liste, die bereits den Anfangswert enthält
    # iteriere über die Zeit
    for t in T:
        # Integration: berechne die Differenzengleichung und füge die Ergebnisse zur Liste N hinz
        N.append(N[t] + (gamma * N[t] * (1 - N[t]/K) * (N[t] - a)/K) * dt)
    # plotte das Ergebnis
    plt.plot(N, label = 'a = %0.1f' %a)

# Ausschmücken des Plots
plt.title('Allee-Effekt')
plt.ylim(-1, 11)
plt.grid()
plt.legend(loc = 'lower right')
```

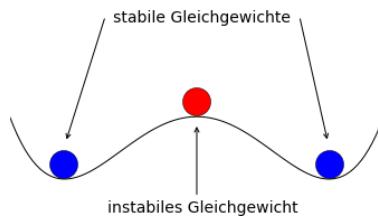
Out[2]: <matplotlib.legend.Legend at 0xad0fb38>



Wird der Aussterbeschwellenwert a gleich groß wie der Anfangswert aw gewählt - in diesem Fall $a = aw = 5$ -, so verändert sich die Größe der Population nicht mehr. Das System ist **stabil**. Liegt der a -Wert allerdings nur ein klein wenig unter oder über dem Anfangswert - hier zum Beispiel bei 4.9 oder 5.1 - so strebt die Populationsgröße (das System) von diesem Schwellenwert weg und erreicht früher oder später entweder den Nullwert - die Population stirbt aus - oder, im anderen Fall, die Umweltkapazitätsgrenze ("carrying capacity") $K = 10$. Auch diese beiden Zustände sind sodann **stabil**. Das System verändert sich nicht mehr.

Im Gegensatz zum Aussterbeschwellenwert a , bei dem die Entwicklung nur stabil bleibt, wenn der Wert ganz genau getroffen ist (was in der Natur natürlich so gut wie nie der Fall ist), werden die beiden anderen stabilen Zustände, die Umweltkapazitätsgrenze wie auch der Nullzustand (also das Ausgestorben-Sein), von vielen unterschiedlichen Anfangswerten aus angestrebt. Das heißt, diese beiden Zustände wirken als **Attraktoren**, während der Systemzustand am Aussterbeschwellenwert als "Abstoßer" ("Repellent") wirkt. In systemwissenschaftlicher Diction markieren die beiden Attraktoren **stabile Gleichgewichte** und der "Abstoßer" ein **instabiles Gleichgewicht**.

Als **Potentialtopf** (engl.: "potential well"), bzw. Doppel-Potentialtopf so wie in der folgenden Grafik dargestellt, wird deutlich, wie das instabile Gleichgewicht im Vergleich zu den beiden stabilen Gleichgewichten vorgestellt werden kann. Nur wenn das rote Kugelchen wirklich ganz genau in der Mitte des Maximums zwischen den beiden Minima zu liegen kommt, würde es tatsächlich liegen bleiben. Die leiseste Erschütterung dagegen, würde es in Richtung einer der beiden Minima rollen lassen. Diese wirken diesbezüglich als **Attraktoren**.



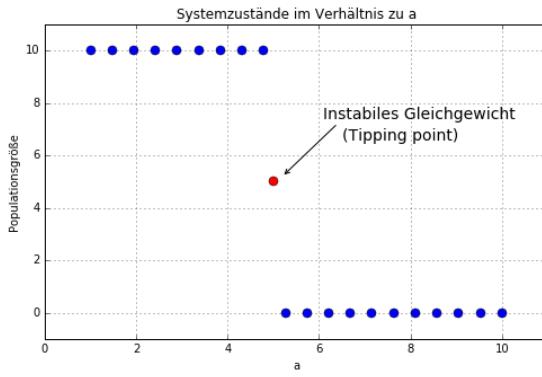
Dieser Umstand wird vielleicht noch deutlicher, wenn die Systemzustände - und zwar am Ende der hier betrachteten zeitlichen Entwicklung, also bei $t = 2999$ - nicht in der Zeit, sondern in Bezug auf ein dichteres Intervall von a -Werten dargestellt werden.

```
In [4]: fig = plt.figure(figsize=(8,5))
fig.add_subplot(1,1,1)

A = np.linspace(1, 10, 20)
T = range(3000)
gamma = 0.1
K = 10.

NN = []
for a in A:
    N = [5.] # bilde eine Liste, die bereits den Anfangswert enthält
    # iteriere über die Zeit
    for t in T:
        # integriere und füge die Ergebnisse zur Liste N hinzu
        N.append(N[t] + (gamma * N[t] * (1 - N[t]/K) * (N[t] - a)/K) * dt)
    # füge das Ergebnis in eine weitere Liste NN ein
    NN.append(N[-1])

# plotte NN in Bezug auf A
plt.plot(A, NN, 'bo', markersize = 8)
plt.plot(5, 5, 'ro', markersize = 8)
plt.xlim(0, 11)
plt.ylim(-1, 11)
# verwende unicode um den Umlaut richtig darzustellen
plt.title(u'Systemzustände im Verhältnis zu a')
plt.xlabel('a')
plt.ylabel('Populationsgröße')
plt.text(6.1, 7.4, 'Instabiles Gleichgewicht', fontsize = 14)
plt.text(6.5, 6.6, '(Tipping point)', fontsize = 14)
plt.annotate('', xy=(5.2, 5.2), xytext=(6.4, 7.2), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=1),)
plt.grid()
```



Wir sehen, dass a -Werte < 5 das System (die Population) geschlossen zum Wert $K = 10$ anwachsen lassen, a -Werte > 5 dagegen das System durchwegs auf Null reduzieren. In diesem Fall wirkt der Schwellenwert 5 wie ein Schalter, der zwischen Aussterben und Überleben auf Umweltkapazitätsgrenze hin und her schaltet.

Gleichgewichte und Fixpunkte

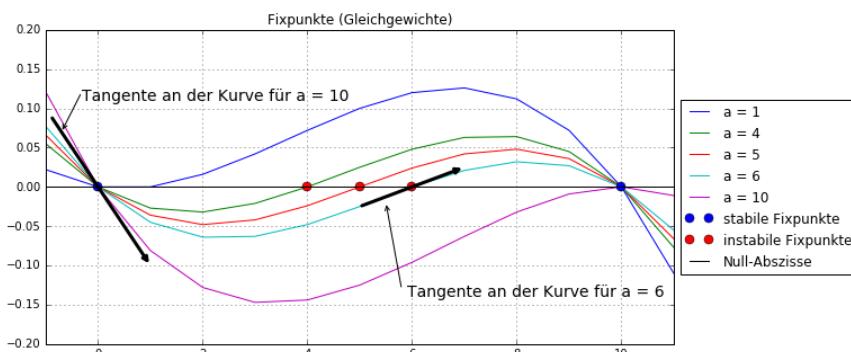
Grundsätzlich markieren Gleichgewichte Systemzustände, bei denen ein System sein Verhalten nicht (mehr) ändert. Wenn - wie im vorliegenden Fall - solche Systemzustände mit *Differenzierenden- oder Differentialgleichungen* modelliert werden, so lässt sich, weil mit diesen Gleichungen Veränderungsraten wiedergegeben werden, ein Gleichgewicht finden, indem die entsprechende Gleichung, also die Veränderung, gleich Null gesetzt wird (d.h. es findet keine Veränderung statt).

Im vorliegenden Fall können wir dies dadurch erreichen, dass die obige Differenzengleichung wie eine Funktion $f(x) = \gamma * x * (1 - \frac{x}{K}) * \frac{x-a}{K}$ berechnet und gezeichnet wird - und zwar auch in diesem Fall für verschiedene a -Werte. Die Kurven, die sich aus dieser Berechnung ergeben, schneiden an verschiedenen Punkten die Null-Abszisse. Dies sind somit jene Punkte, an denen unsere Gleichung gleich Null ist. Diese Punkte markieren damit die Gleichgewichte, bzw. die **Fixpunkte** unserer Systementwicklung.

Ob diese Fixpunkte stabile oder instabile Gleichgewichte darstellen, kann in diesem Fall am leichtesten festgestellt werden, indem in den Punkten **Tangenten** an die durch sie gehenden Kurven angelegt werden. Wenn diese Tangenten eine negative Steigung haben (sprich von links oben nach rechts unten weisen), so liegt an dem Punkt ein stabiles Gleichgewicht vor. Wenn die Tangenten dagegen eine positive Steigung haben (von links unten nach rechts oben weisen), so liegt ein instabiles Gleichgewicht vor.

```
In [5]: fig = plt.figure(figsize=(10,5))
fig.add_subplot(1,1,1)

A = [1, 4, 5, 6, 10]
X = np.arange(-1, 12)
for a in A:
    F = []
    for x in X:
        # berechne die Funktion und füge die Ergebnisse zur Liste F hinzu
        F.append(gamma * x * (1 - x/K) * (x - a)/K)
    plt.plot(X, F, label = 'a = %s' %a)
# Tangenten
plt.text(-0.3, 0.11, u'Tangente an der Kurve für a = 10', fontsize = 14)
plt.annotate('', xy=(1, -0.1), xytext=(-0.9, 0.09), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=3),)
plt.annotate('', xy=(-0.7, 0.07), xytext=(-0.32, 0.12), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=1),)
plt.text(5.9, -0.14, u'Tangente an der Kurve für a = 6', fontsize = 14)
plt.annotate('', xy=(7, -0.025), xytext=(5, -0.025), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=3),)
plt.annotate('', xy=(5.5, -0.02), xytext=(5.8, -0.13), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=1),)
plt.plot(0, 0, 'bo', markersize = 8, label = 'stabile Fixpunkte')
plt.plot(4, 0, 'ro', markersize = 8, label = 'instabile Fixpunkte')
plt.plot(5, 0, 'ro', markersize = 8)
plt.plot(6, 0, 'bo', markersize = 8)
plt.plot(10, 0, 'bo', markersize = 8)
plt.plot(X, [0]*len(X), 'k-', label = 'Null-Abszisse')
plt.title('Fixpunkte (Gleichgewichte)')
plt.xlim(-1, 11)
plt.ylim(-0.2, 0.2)
# platziere die Legende außerhalb der Zeichenfläche
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.grid()
```



Wenn wir nun diese Funktion erneut für sehr viele a -Werte in dieser Weise berechnen, die Schnittpunkte der Kurven mit der Null-Abszisse herauslösen und für sich in Bezug auf die a -Werte zeichnen, so erhalten wir eine Gleichgewichtsübersicht für den betrachteten Systemzustandsbereich - auch **Phasenraum** (engl. "phase space") genannt.

Allerdings wäre hierbei zu beachten, dass wir, um die Schnittpunkte herauslösen zu können, erstens die Funktion mit großer Auflösung berechnen müssten, und zum zweiten auch eine gewisse Toleranz um Null herum berücksichtigen müssten, weil die Schnittpunkte sonst durch die Lücken der Rechenschritte fallen und nicht gefunden werden.

Um dies anhand eines interessanteren Modells zu sehen, betrachten wir als nächstes:

Beispiel 2: Nährstoffeintrag in einem Teich

In diesem Fall wird das Modell durch die folgende Differentialgleichung dargestellt

$$\frac{dX}{dt} = a - b * X + r * \frac{X^p}{X^p + h^p}$$

wobei die Variable X ein Nährstoff ist, gelöst im Phytoplankton eines Teiches, der Parameter a den Nährstoffeintrag in diesen Teich angibt, der Parameter b den Nährstoffabbau und r eine natürliche Recycling-Rate des Nährstoffs bestimmt. h gibt die Halbwelt der Sättigung des Recycling an, und der Exponent p bestimmt die Stärke der Rückkoppelung, mit der das Recycling wirkt (siehe ausführlicher dazu <http://systems-sciences.uni-graz.at/etextbook/sw2/crittrans.html> und <http://systems-sciences.uni-graz.at/etextbook/sw2/catatrophe.html>).

Wir betrachten das Modell neuerlich in Form einer Differenzengleichung und wandeln es daher wie folgt um:

$$X_{t+1} = X_t + \Delta X_t$$

mit

$$\Delta X_t = a - b * X_t + \frac{X_t^p}{X_t^p + h^p}$$

Dargestellt mit Python sieht dies wie folgt aus:

```
In [6]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

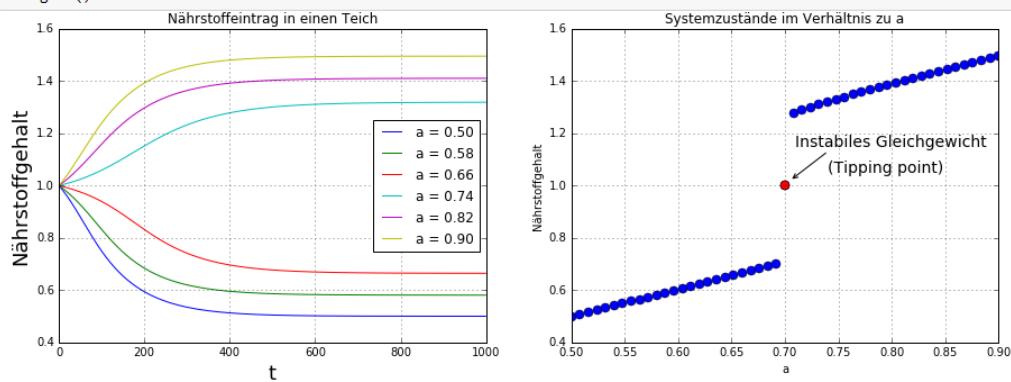
fig = plt.figure(figsize=(15,5))
ax1 = fig.add_subplot(1,2,1)

# Parameter und leere Listen
h = 1.; b = 1.; p = 12; r = 0.6; dt = 0.01; A=[]; XA=[]; XX = []

T = range(1000)
for n, a in enumerate(A):
    X = [1]
    for t in T:
        # die Differenzengleichung
        X.append(X[t] + (a - b * X[t] + r * (X[t]**p / (X[t]**p + h**p))) * dt)
    # zeichne nur jede 10te Entwicklung
    if n % 10 == 0:
        plt.plot(T, X[:1], label = ('a = %0.2f' %a))
    # aber füge die letzten Werte aller Entwicklungen in die Liste XX hinzu
    XX.append(X[-1])

# zeichnen
ax1.set_title(u'Nährstoffeintrag in einen Teich')
ax1.set_xlabel('t', fontsize=18)
ax1.set_ylabel(u'Nährstoffgehalt', fontsize=18)
ax1.grid()
plt.legend(loc='best')

ax2 = fig.add_subplot(1,2,2)
ax2.plot(A, XX, 'bo', markersize = 8)
ax2.plot(0.71, 1.15, 'ro', markersize = 8)
ax2.set_xlim(0.5, 0.9)
ax2.set_title(u'Systemzustände im Verhältnis zu a')
ax2.set_xlabel('a')
ax2.set_ylabel(u'Nährstoffgehalt')
ax2.text(0.71, 1.15, 'Instabiles Gleichgewicht', fontsize = 14)
ax2.text(0.74, 1.05, '(Tipping point)', fontsize = 14)
ax2.annotate('', xy=(0.705, 1.02), xytext=(0.74, 1.13), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=1),)
ax2.grid()
```



Wir sehen in diesem Beispiel wieder einen Schwellenwert - er liegt hier bei $a = 0.7$ - unter dem der Nährstoffgehalt im See gering bleibt, über dem er aber schnell anwächst.

In der rechten Darstellung sehen wir Gleichgewichtswerte, die über weite Bereiche zusammen mit a leicht ansteigen, bei $a = 0.7$ aber einen **tipping point** aufweisen und das System, d.h. in dem Fall den See von einem Zustand mit relativ niedrigem Nährstoffgehalt nahezu schlagartig in einen Zustand mit hohem Nährstoffanteil versetzen. Der Nährstoffgehalt explodiert förmlich. Der See veragt und wandelt sich in kurzer Zeit von einem Klarwasser- in einen Trüb Wassersee.

Wir wenden wieder die obige Methode an und betrachten die Differenzengleichung als Funktion, deren Schnittpunkte mit der Null-Abszisse die Gleichgewichts-(oder Fix-)Punkte der Entwicklung angeben. Wir können auch hier wieder die Steigungen der Tangenten in den Schnittpunkten betrachten, um die Qualität der Fixpunkte zu bestimmen. Dies ist allerdings gleich bedeutend mit dem **Differenzieren der Funktion** (die Tangente im Schnittpunkt ergibt sich aus der Ableitung der Kurvenfunktion). Erneut gilt, dass, wenn die Ableitung negativ ist, d.h. die Steigung der Tangente nach unten weist, ein stabiler Gleichgewichtspunkt vorliegt, und wenn die Ableitung positiv ist, d.h. die Steigung der Tangente nach oben weist, ein instabiler Gleichgewichtspunkt gegeben ist.

```
In [7]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(15,15))
ax1 = plt.subplot(3, 1, 1)

# Leere Listen um die Gleichgewichtswerte zu sammeln
EX1 = []; EX2 = []; EX3 = []

T = np.linspace(0.5, 1.5, 1000)
A = np.linspace(0.5, 0.9, 501)

for n, a in enumerate(A):
    F = []
    for x in T:
        f = a - b * x + (r * x**p)/(x**p + h**p)
        F.append(f)
    if -10**-4 < f < 10**-4:
        # bilde die Ableitung von f:
        Df = -b + (r * p * x**p - 1) * h**p / ((x**p + h**p)**2)
        # schreibe die x-Koordinate eines Gleichgewichts zusammen mit dem entsprechenden a-Wert in eine Liste
        if Df > 0: EX2.append([x, a])      # Liste für instabile Gleichgewichte
        if Df < 0:
            if x < 1: EX1.append([x, a])  # Liste für stabile Gleichgewichte < 1 (untere Gleichgewichte)
            if x > 1: EX3.append([x, a])  # Liste für stabile Gleichgewichte > 1 (obere Gleichgewichte)
    # zeichne eine Auswahl der Kurven für unterschiedliche a-Werte
    if n%100 == 0:
        ax1.plot(T, F, label = 'a = ' + str(a))

# zeichne Tangenten
ax1.text(0.65, 0.4, 'Tangente an stabilem Fixpunkt', fontsize = 14)
ax1.annotate('', xy=(0.75, -0.074), xytext=(0.6, 0.063), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=3),)
ax1.annotate('', xy=(0.62, 0.07), xytext=(0.645, 0.42), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=1),)
ax1.text(1.1, -0.4, 'Tangente an instabilem Fixpunkt', fontsize = 14)
ax1.annotate('', xy=(1.14, 0.053), xytext=(0.98, -0.054), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=3),)
ax1.annotate('', xy=(1.04, -0.05), xytext=(1.095, -0.37), arrowprops=dict(arrowstyle="->", connectionstyle="arc3", lw=1),)

ax1.set_title('Fixpunkte für ausgewählte a-Werte')
ax1.set_xlabel('x', fontsize=18)
ax1.set_ylabel('f', fontsize=18)
# zeichne Punkte
ax1.plot(0.5, 0, 'ko', markersize = 8, label = 'stabile Fixpunkte')
ax1.plot(0.58, 0, 'ko', markersize = 8)
ax1.plot(0.67, 0, 'ko', markersize = 8)
ax1.plot(0.76, 0, 'ko', markersize = 8)
ax1.plot(0.95, 0, 'wo', markersize = 8, label = 'instabile Fixpunkte')
ax1.plot(1.06, 0, 'wo', markersize = 8)
ax1.plot(1.2, 0, 'ko', markersize = 8)
ax1.plot(1.32, 0, 'ko', markersize = 8)
ax1.plot(1.41, 0, 'ko', markersize = 8)
ax1.plot(1.5, 0, 'ko', markersize = 8)
ax1.plot(T, [0]*len(T), 'k-', label = 'Null-Abszisse')
ax1.set_xlim(0.5, 1.5)
ax1.grid()
ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5))

#sortiere die Listen mit den Gleichgewichten bez. zunehmendem a-Parameter
EX1 = zip(*sorted(EX1, key=lambda x: x[0])); EX1x = EX1[1]; EX1y = EX1[0]
EX2 = zip(*sorted(EX2, key=lambda x: x[0])); EX2x = EX2[1]; EX2y = EX2[0]
EX3 = zip(*sorted(EX3, key=lambda x: x[0])); EX3x = EX3[1]; EX3y = EX3[0]

# 2ter Plot
ax2 = plt.subplot(3, 1, 2)
# der untere Teil der Kurve (stabile Fixpunkte)
ax2.plot(EX1x, EX1y, 'ko', label = 'stabile Fixpunkte', linestyle = 'dashdot');
# mittlerer Teil der Kurve (instabile Fixpunkte)
ax2.plot(EX2x, EX2y, 'wo', label = 'instabile Fixpunkte', linestyle = 'dashdot');
# der obere Teil der Kurve (stabile Fixpunkte)
ax2.plot(EX3x, EX3y, 'ko', linestyle = 'dashdot');

# zusätzliche Punkte
ax2.plot(0.7, 0.95,color = 'm',marker='o')
ax2.plot(0.7, 1, color = 'c', marker='x',ms=8,mew=3)

ax2.plot(0.7,0.95,color = 'r',marker='o')
ax2.annotate('', xy=(0.7, 0.73), xytext=(0.7, 0.92),arrowprops=dict(arrowstyle="->",connectionstyle="arc3",lw=1),)
ax2.plot(0.7,0.72,color = 'm',marker='x',ms=8,mew=3)

ax2.plot(0.775,0.95,color = 'r',marker='o')
ax2.annotate('', xy=(0.775, 1.34), xytext=(0.775, 0.97),arrowprops=dict(arrowstyle="->",connectionstyle="arc3",lw=1),)
ax2.plot(0.775,1.36,color = 'r',marker='x',ms=8,mew=3)
```

```

ax2.plot(0.775,0.5,color = 'b',marker='o')
ax2.annotate(' ', xy=(0.775, 0.81), xytext=(0.775, 0.53),arrowprops=dict(arrowstyle="->",connectionstyle="arc3",lw=1,))
ax2.plot(0.775,0.85,color = 'b',marker='x',ms=8,mew=3)

ax2.plot(0.785,0.5,color = 'y',marker='o')
ax2.annotate(' ', xy=(0.785, 1.33), xytext=(0.785, 0.53),arrowprops=dict(arrowstyle="->",connectionstyle="arc3",lw=1,))
ax2.plot(0.785,1.37,color = 'y',marker='x',ms=8,mew=3)

# Plot ausschmücken
ax2.set_title('Fixpunkte')
ax2.grid();
ax2.set_xlabel('$a$', fontsize=18)
ax2.set_ylabel('$x$', fontsize=18)
ax2.legend(loc='center left', bbox_to_anchor=(1, 0.5))

# 3ter Plot
ax3 = plt.subplot(3, 1, 3);

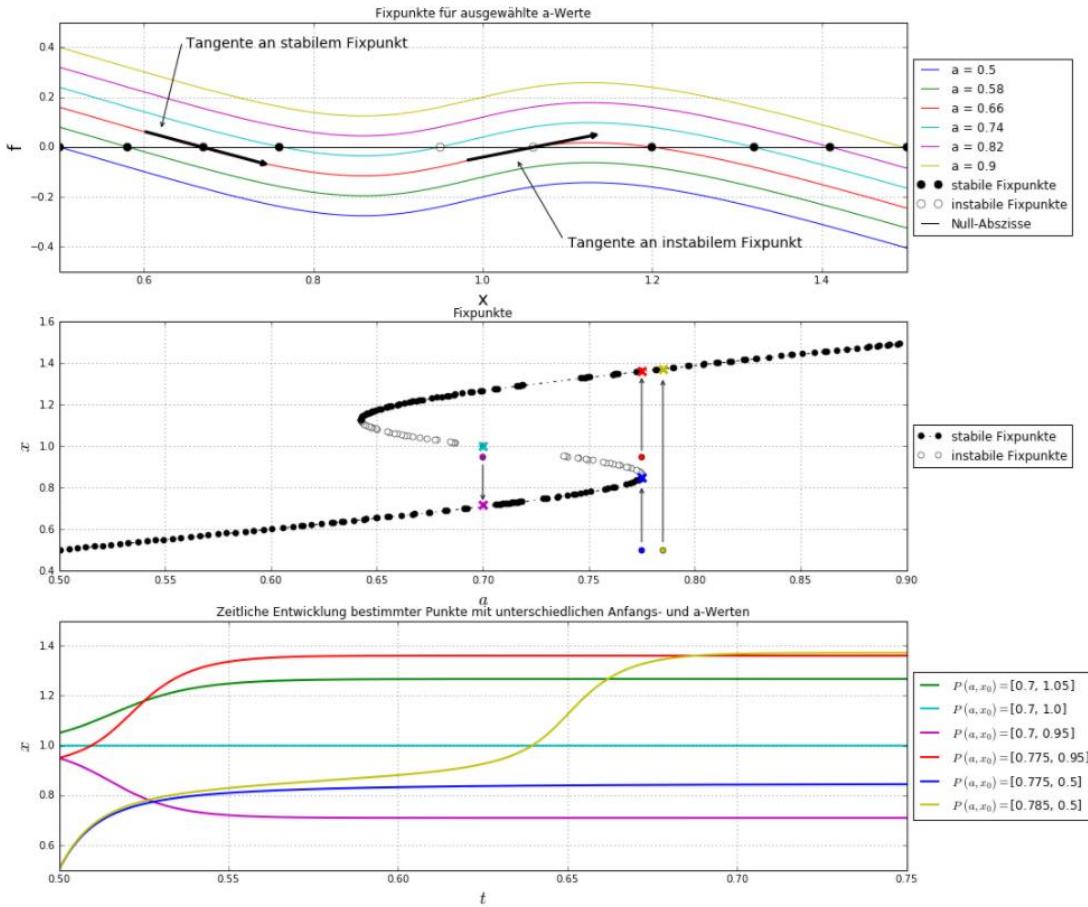
# Parameter und Listenvariablen:
h = 1.; b = 1.; p = 12; r = 0.6; A=[]; XA=[]
start_xy_list = [[0.7,1.05,'g'],[0.7,1.0,'c'],[0.7,0.95,'m'],[0.775,0.95,'r'],[0.775,0.5,'b'],[0.785,0.5,'y']]

# variiere Parameter a im Intervall (0.4,0.9):
for start_xy in start_xy_list:
    a = start_xy[0]
    x = start_xy[1]
    color = start_xy[2]
    X = []; T = []
    for t in np.linspace(0.5,0.75,2500):
        # Differenzgleichung
        delta_x = a - b * x + r * (x**p / (x**p + h**p))
        x += delta_x/100
        X.append(x); T.append(t)
    ax3.plot(T,X,label = '$\$\$ \$\$(a,x_0) = \$\'+ str(start_xy[:2]),color = color, linewidth=2)
ax3.legend(loc='center left', bbox_to_anchor=(1, 0.5))

# Plot ausschmücken
ax3.set_title('Zeitliche Entwicklung bestimmter Punkte mit unterschiedlichen Anfangs- und a-Werten')
ax3.grid();
ax3.set_xlim(0.5,0.75)
ax3.set_ylim(0.5,1.5)
ax3.set_xlabel('$t$', fontsize=18)
ax3.set_ylabel('$x$', fontsize=18)
ax3.legend(loc='center left', bbox_to_anchor=(1, 0.5))

```

Out[7]: <matplotlib.legend.Legend at 0xb9c6908>



Im mittleren der drei obigen Plots sind einige zusätzliche (bunte) Punkte eingefügt und mithilfe von Pfeilen und Kreuzen ist markiert, welche Richtung die Entwicklung der entsprechenden Teich-Zustände nehmen würde, wenn sie von den angegebenen Anfangs- und α -Parameterwerten aus gestartet würden. Im untersten Plot werden diese Entwicklungen sodann in der Zeit dargestellt.

Wir sehen, dass kleinste Anfangsunterschiede und auch sehr kleine α -Wert-Unterschiede recht große Unterschiede des schlußendlichen Teichzustandes hervorrufen können. Besonders markant scheint etwa der Unterschied zwischen gelber und blauer Entwicklung: obwohl die Anfangszustände die selben sind und die α -Werte nur um 0.01 differieren, unterscheiden sich die schlussendlich *stabilen* Teichzustände, die das System ausbildet, markant.

Zusammenfassung

Systemstabilitäten (Eigenverhalten)

Stabile Systemzustände können aus dem Zusammenwirken von Einzelkomponenten entstehen ("emergieren"), die (auf Microebene) nicht notwendig das selbe oder ein ähnliches Verhalten zeigen, wie das System (auf Macroebene) selbst. Systemwissenschaftlich spricht man von Eigenverhalten.

Zwischen diesen stabilen Systemzuständen kommt es zu Phasenübergängen (also zur Veränderung des Eigenverhaltens), beim Wasser etwa zum Phasenübergang von flüssiger zu gas- oder fest-förmiger Form. Diese Phasenübergänge können abrupt und damit überraschend an so genannten instabilen Fixpunkten ("tipping points") auftreten.

Systemanalyse

Um festzustellen, wo ein (mathematisch definiertes) System seine Fixpunkt- (oder auch Gleichgewichts-) Zustände hat, werden die Differential- oder Differenzengleichungen, die dieses System bestimmen, gleich Null gesetzt und gelöst. Grafisch kann dies geschehen, indem die Gleichungen als Funktionen interpretiert und gezeichnet werden. Die Schnittpunkte dieser Funktion mit der Nulllinie markieren sodann die Fixpunkte.

Die Qualität der Fixpunkte (also ob es sich um einen *stabilen* oder *instabilen* Fixpunkt handelt), kann dabei grafisch anhand der Tangenten-Steigung in diesem Fixpunkt bestimmt werden. Weist die Tangente eine negative Steigung auf (sie weist von links-oben nach rechts-unten), so handelt es sich um einen stabilen Fixpunkt. Weist die Tangente dagegen eine positive Steigung auf (sie weist von links-unten nach rechts-oben), so handelt es sich um einen instabilen Fixpunkt.

Die Stabilitäten komplexer Systeme können sehr empfindlich sein. Kleinste Anfangs- oder Parameterunterschiede können große Unterschiede der Stabilitätszustände zur Folge haben.

Kapitel 15 – Spaß mit Attraktoren

Dieser Teil des Skriptums soll abschließend ein paar Grundlagen des systemwissenschaftlichen Arbeitens mit Python zu erinnern helfen. Als Beispiel, anhand dessen das Definieren von Variablen, das Rechnen in Schleifen und auch das Zeichnen von Plots noch einmal wiederholt wird, dienen so genannte "seltsame Attraktoren" (engl.: *strange*, oder auch: *itinerant attractors*), die neben ihrem interessanten systemwissenschaftlichen Aspekt auch einfach ästhetisch ansprechend sein können.

Wir beginnen mit dem wohl bekanntesten "seltsamen Attraktor", dem so genannten

Lorenz-Attraktor

Wie [hier](#) nachzulesen ist, wurde der Lorenz-Attraktor im Versuch gefunden, ein einfaches Wettermodell zu erstellen. Dieses Modell besteht aus dem folgenden System gekoppelter Differentialgleichungen:

$$\begin{aligned}\frac{dx}{dt} &= -a * x_n + a * y_n \\ \frac{dy}{dt} &= b * x_n - y_n - x_n * z_n \\ \frac{dz}{dt} &= -c * z_n + x_n * y_n\end{aligned}$$

Wir betrachten das System im folgenden allerdings erneut in Form von Differenzengleichungen

$$\begin{aligned}x_{n+1} &= x_n + (-a * x_n + a * y_n) * dt \\ y_{n+1} &= y_n + (b * x_n - y_n - x_n * z_n) * dt \\ z_{n+1} &= z_n + (-c * z_n + x_n * y_n) * dt\end{aligned}$$

und zeichnen seinen Attraktor in Phasenraumdarstellung (im Englischen auch *map* genannt). Wir wollen dabei den Umstand besonders hervorheben, dass Attraktoren - wie ihr Name sagt - von einer Vielzahl von Anfangszuständen aus angestrebt werden, d.h. dass für die spezifische Form eines seltsamen Attraktors nicht die Anfangswerte, von denen aus zu rechnen begonnen wird, verantwortlich sind, sondern vielmehr die "interne" Wechselwirkung der durch die Differenzengleichungen zum Ausdruck gebrachten Dynamiken. Mit anderen Worten, wir wollen zeigen, dass das Lorenz-System (d.h. das Lorenzsche Wettermodell) ein *Eigenverhalten* hat, das unabhängig von Anfangsgegebenheiten immer die selbe Form - eine *Eigenform* - ausbildet, eben den Lorenz-Attraktor.

Zu diesem Zweck integrieren wir das Gleichungssystem im folgenden dreimal von auffallend verschiedenen Ausgangspunkten (rot, grün, blau) und zeichnen seine Trajektorien zunächst im zwei-dimensionalen Raum (d.h. wir ignorieren in der Darstellung des Attraktors das individuelle Verhalten einer der drei Gleichungen, das der y-Gleichung).

```
In [1]: # importiere die benötigten Module
import numpy as np
import random
import matplotlib.pyplot as plt
%matplotlib inline

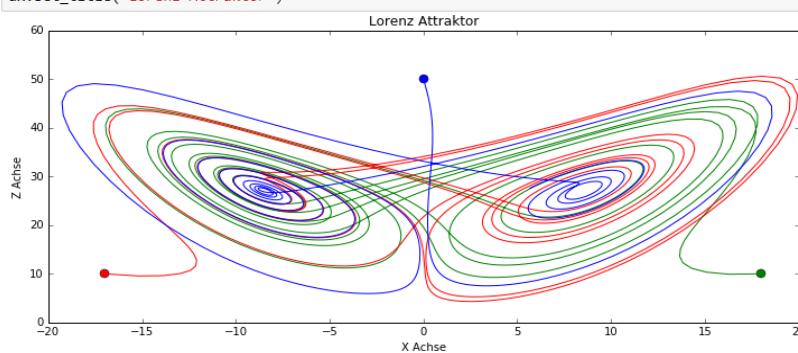
# definiere die Zeichenfläche
fig = plt.figure(figsize=(12, 5))
ax = fig.add_subplot(111)

# Parameter nach Lorenz
a, b, c = 10., 2.66, 28.
# Integrationsschrittgröße
dt = 0.01
# Farben zum Unterscheiden der drei Trajektorien
colors = ['g-', 'r-', 'b-']
c_point = ['go', 'ro', 'bo']
# Drei Punkte als Anfangswerte, in einer Liste
IV = [[18, 1, 10], [-17, 1, 10], [0, 1, 50]]
# Zeitachse
T = np.linspace(0, 10, 1000)

# Schleife über die drei unterschiedlichen Anfangspunkte
for col in range(len(colors)):
    # Auslesen der Anfangswerte
    u = [IV[col][0]]; v = [IV[col][1]]; w = [IV[col][2]]
    # Zeichnen der Anfangspunkte, hier in 2D als u, w, bzw. auf der x- und z-Achse
    ax.plot(u, w, c_point[col], markersize = 8)

    # Schleife zum Integrieren
    for n,t in enumerate(T):
        u.append(u[n] + (-a * (u[n] - v[n])) * dt)
        v.append(v[n] + (c * u[n] - v[n] - u[n] * w[n]) * dt)
        w.append(w[n] + (-b * w[n] + u[n] * v[n]) * dt)
    # Zeichnen
    ax.plot(u, w, colors[col])

# Plot beschriften
ax.set_xlabel("X Achse")
ax.set_ylabel("Z Achse")
ax.set_title("Lorenz Attraktor")
```



Das Lorenz-System besteht aus drei gekoppelten, gewöhnlichen nicht-linearen Differential-, bzw. hier Differenzengleichungen. Seine Phasenraumdarstellung sollte daher **drei Dimensionen** umfassen, und nicht, wie in obiger Darstellung, nur zwei. Das Python-Modul `matplotlib` ermöglicht aber selbstverständlich auch drei-dimensionale Darstellungen, wie der folgende Code vorführt.

Zur nochmaligen Verdeutlichung des Systemeigenverhaltens wurden im Folgenden gleich fünf unterschiedliche Anfangspunkte gewählt. Überdies wurde das System als (wiederaufrufbare) Funktion definiert und mit der `scipy`-Funktion `odeint` integriert.

```
In [2]: import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(15, 10))
# definiere Matplotlib 3D Projektion
ax = fig.gca(projection='3d')

# Parameter nach Lorenz
a, b, c = 10., 2.66, 28.
# Anfangswerte
IV = [[30, -30, 0], [-40, -30, 0], [-20, 40, 60], [10, -30, -20], [30, 50, 20]]
# Farben
colors = ['g-', 'r-', 'b-', 'y-', 'c-']
c_point = ['go', 'ro', 'bo', 'yo', 'co']

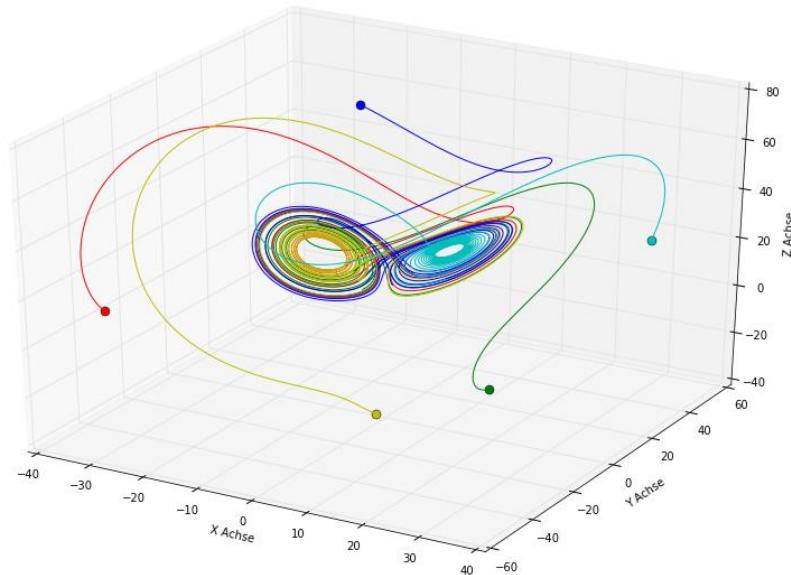
# Funktionsdefinition
def lorenz(L, t, a, b, c):
    # Lorenzgleichungen
    u, v, w = L
    up = -a * (u - v)
    vp = c * u - v - u * w
    wp = -b * w + u * v
    return up, vp, wp

T = np.linspace(0, 10, 10000)

for col in range(len(colors)):
    u0 = IV[col][0]; v0 = IV[col][1]; w0 = IV[col][2]
    ax.plot([u0], [v0], [w0], c_point[col], markersize = 8)
    # Integration mit odeint
    f = odeint(lorenz, (u0, v0, w0), T, args=(a, b, c))
    u, v, w = f.T
    ax.plot(u, v, w, colors[col])

# Beschriftung
ax.set_xlabel("X Achse")
ax.set_ylabel("Y Achse")
ax.set_zlabel("Z Achse")

# falls gewünscht, zur Darstellung ohne Achsen
#ax.set_axis_off()
```



Der Lorenz-Attraktor beschreibt einen spezifischen Phasenraum, der von den Trajektorien (d.h. den Verlaufskurven) der Dynamiken, die das System bestimmen, eingenommen wird, und zwar unabhängig von seinen Anfangswerten. Dieser Raum, der aussieht wie eine etwas verbogene Acht, wird dabei erst nach und nach ausgefüllt, wobei sich die Trajektorien aber niemals wiederholen. Sie wandern vielmehr innerhalb dieses Raums bis ins Unendliche herum. Diese Art von Attraktor wird deshalb im Englischen auch gerne *itinerant attractor* genannt (itinerant für herumwandern).

Aus diesem "Herumwandern" beziehen solche seltsamen Attraktoren ihr ästhetisches Potential. Ein weiteres Beispiel dafür gibt der so genannte

Peter de Jong-Attraktor

Dieser Attraktor ergibt sich aus dem folgenden Gleichungssystem:

$$\begin{aligned}x_{n+1} &= \sin(a * y_n) - \cos(b * x_n) \\y_{n+1} &= \sin(c * x_n) - \cos(d * y_n)\end{aligned}$$

und erzeugt in Abhangigkeit seiner Parameterwerte eine Vielzahl interessanter Formen. Wir verwenden ihn im Folgenden, um anhand seines Beispiels einerseits die Iteration uber verschiedene Parameterwerte und das entsprechende Zeichnen der Resultate zu wiederholen und dabei gleichzeitig die Moglichkeit zu demonstrieren, verschiedene Farben als esthetischen Effekt zu verwenden.

Eine weitere nette Online-Demonstration für fortgeschrittenere Visualisierungsmöglichkeiten findet sich [hier](#).

```
In [3]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

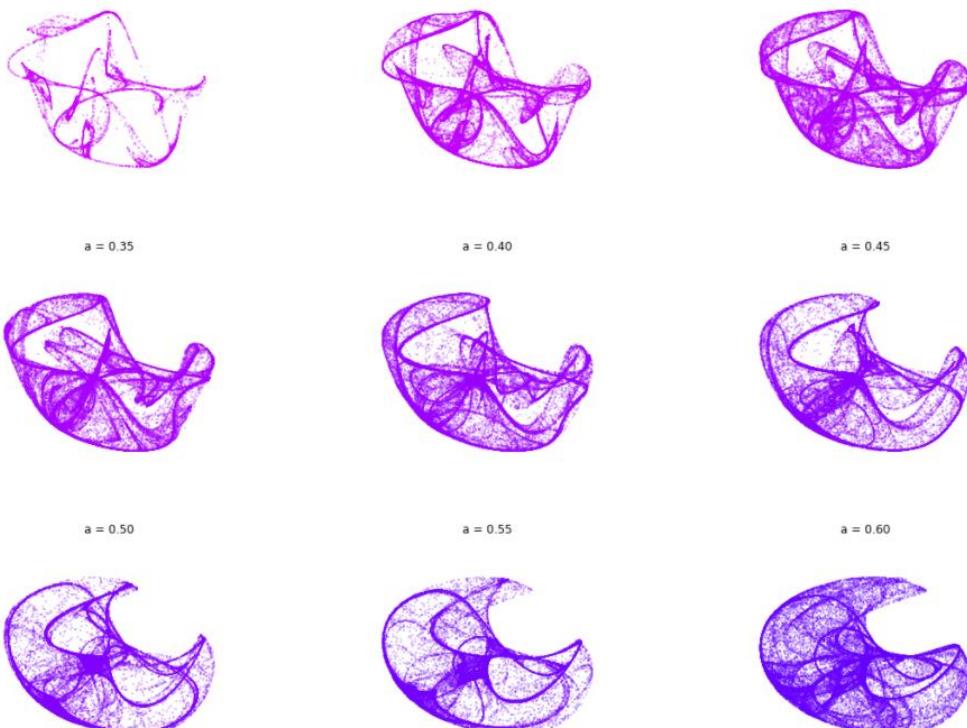
# definiere die allgemeine Zeichenfläche
fig = plt.figure(figsize=(20,15))

# Parameter
# definiere ein Liste von a-Werten
A = np.linspace(0.2, 0.6, 9)
b = 0.66
c = 0.49
d = 0.68

# ein Zähler für die Zeichenflächen
m = 1

# Schleife über die a-Werte
for n, a in enumerate(A):
    # definiere iterativ die speziellen Zeichenflächen
    ax = fig.add_subplot(3, 3, n + 1)
    # springe stets nach drei Plots eine Zeile weiter
    if n%3 + 1 == 3:
        m += 1
    # Anfangswerte
    X = [0.001]
    Y = [0.001]
    # Schleife über die Zeit
    for i in range(50000):
        X.append(np.sin(a * Y[i] * np.pi) - np.cos(b * X[i] * np.pi))
        Y.append(np.sin(c * X[i] * np.pi) - np.cos(d * Y[i] * np.pi))

    # wir zeichnen hier keine Trajektorien, sondern die konkreten Datenpunkte, undd verwenden dazu die Funktion scatter
    # für die Farben wird die RGB-Codierung (https://de.wikipedia.org/wiki/RGB-Farbraum) verwendet
    # und mit dem a-Wert abgleichen
    ax.scatter(X, Y, color = (1 - a, 0, 1), s = 0.1)
    # als Titel der jeweilige a-Wert
    ax.set_title('a = %0.2f' %a)
    # ohne Achsen
```



Zum Abschluss sehen wir uns noch kurz einen Attraktor an, der mit bestimmten Parameterwerten Gebilde erzeugt, die an Pflanzen oder Unterwasser-Lebewesen erinnern, der so genannte

Gumowski-Mira-Attraktor

Dieser Attraktor ergibt sich aus dem folgenden Gleichungssystem:

$$\begin{aligned}x_{n+1} &= y_n + b * (1 - 0.05 * y_n^2) + f(x_n) \\y_{n+1} &= f(x_{n+1}) - x_n\end{aligned}$$

mit

$$f(x) = a * x + 2 * (1 - a) * \frac{x^2}{1 + x^2}$$

Das System wurde von zwei Physikern, I. Gumowski und C. Mira, 1980 im Rahmen ihrer Tätigkeit am CERN in Genf gefunden, wo sie die Trajektorien subatomarer Teilchen berechneten.

Zur Übung schreiben wir hier den gesamten Code zur Kalkulation und zum Zeichnen in eine Funktion, die die Integration ihrerseits auf zwei Funktionen verteilt. Die Farbgebung wird zufällig aus einer Liste von Farben bestimmt.

```
In [4]: import numpy as np
import random
import matplotlib.pyplot as plt
%matplotlib inline

# definiere die allgemeine Zeichenfläche
fig = plt.figure(figsize=(20,6))

# definiere die Funktion
def Gum(a, b):
    # definiere f(x)
    def f(x):
        return a * x + 2. * (1. - a) * x * x / (1. + x * x)

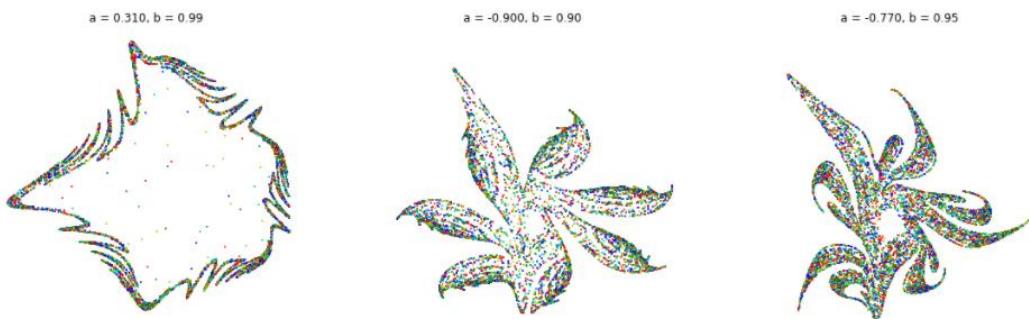
    # definiere die Funktion für die beiden Differenzengleichungen
    def gm(x, y):
        xnew = y + b * (1. - 0.05*y*y) + f(x)
        ynew = f(xnew) - x
        x = xnew
        y = ynew
        return (x, y)

    # Anfangswerte
    x0 = 0.1
    y0 = 0.1
    #Farben
    col = ['g.', 'r.', 'b.', 'y.', 'c.']

    # Schleife zum integrieren
    for i in range(10000):
        (x, y) = gm(x0, y0)
        ax.plot(x, y, random.choice(col), markersize = 3.)
        x0 = x
        y0 = y

    # drei unterschiedliche Parametersettings, über die iteriert wird
    paras = [[0.31, 0.99],
              [-0.9, 0.9],
              [-0.77, 0.95]]

    # Schleife über die unterschiedlichen Parameter-Settings
    for n, p in enumerate(paras):
        ax = fig.add_subplot(1, 3, n+1)
        ax.set_title('a = %0.3f, b = %0.2f' %(p[0], p[1]))
        # Aufrufen der Funktion
        Gum(p[0], p[1])
        # ohne Achsen
        ax.set_axis_off()
```



Dies beendet dieses Skriptum.

Glossar

	Seiten-Nr.		Seiten-Nr.
Allee-Effekt	93	Listen	11, 14
Anaconda	6	Logistisches Wachstum	17, 93
And-Verknüpfung	27	Lotka-Volterra-Regeln	65
Attraktoren	100f	Lorenz-Attraktor	100
Datentypen	50	Matrizen	37ff, 71f
Diskret/kontinuierlich	58	Netzwerke	83f
Differenz/Differential	61f	Netzwerkanalyse	85
Differenzen-/Differentialgleichung	65	Normal-(Gauss)Verteilung	29
Differenzieren	71, 97	Numerisches Integrieren	52, 56
Dynamik	2	Numerisches Differenzieren	55
Eigenverhalten, Eigenform	2, 44, 93	Phasenraumdarstellung	66, 96
Einlesen externer Daten	49	Plotten, Grafiken zeichnen	11, 60, 76f
Euler-Zahl	62	Python - Version 2.x oder 3.x	4
Euler-Verfahren	74	Räuber-Beute-System	64f
Exponentielles Wachstum	16	Rekursion	43
Feedback	22	Runga-Kutta-Verfahren	75
Fibonacci-Folge	14	Small-World-Network	86
Fixpunkte, Gleichgewichte	95	Stabiles/instabiles Gleichgewicht	94
For-Schleifen	10, 14	Stabilität	93f
Funktionen	44	Subplots	79
Gleich-(uniforme)Verteilung	30	Symbolisches Programmieren	70
Histogramm	80	Tipping point	93
If-Abfrage	20	Variablen	9
If-else-Abfrage	23	Vektoren	37ff
Integral, Integrieren, Integration	69, 71	Zellulare Automaten	43, 45
Integrationsmethode	67, 73	Zinsrechnung	59
Jupyter-Notebook	7, 9	Zufallszahlen	29
Komplexität	3	Zufallsereignis	33
Lineares Wachstum	14		