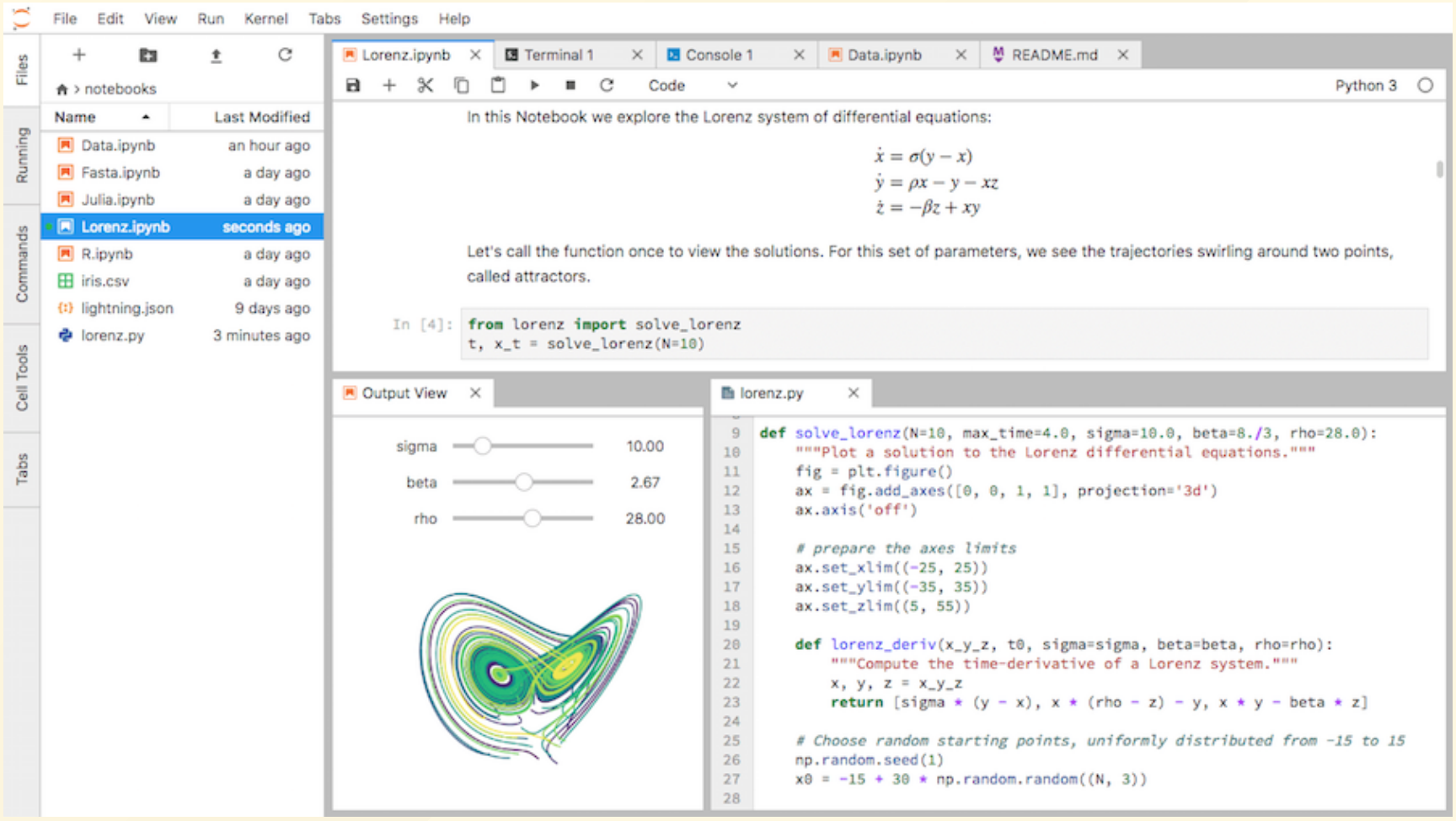
Python Schulung (2)

Ein Parforceritt durch die Sprache

Ein Nachtrag von letzter Woche

- Mittwoch erreichte mich die Nachricht, daß das neue **JupyterLab** nun – obwohl immer noch beta – Produktionsreife erreicht hat.
- Obwohl browserbasiert ist JupyterLab eher eine IDE (stark von RStudio beeinflußt) mit Texteditor, Filebrowser, diversen Ausgabefenstern, Interpreter etc.
- Man hat gar nicht mehr den Eindruck, im Browser zu editieren (editieren zu müssen).



Das könnte was werden!

Und nun zum heutigen Programm

- Werte und Datentypen
- Variablen und Operatoren
- Kommentare
- Funktionen mit und ohne Parameter
- Boolsche Ausdrücke und logische Operatoren
- Verzweigungen
- Schleifen

Werte und Datentypen

- Ein **Wert** ist das grundlegende Ding, mit denen ein Programm arbeitet, etwa **Buchstaben** (Zeichenketten) oder **Zahlen**.
- Diese Werte gehören **Datentypen** an, Zeichenketten sind vom Typ `string`, Zahlen können zum Beispiel vom Typ `int` (Integer, ganze Zahl) oder `float` (Fließkommazahl) sein.
- Mit dem Befehl `type` gibt der Interpreter den Typen aus.

```
type("Hallo Welt!")
```

<class 'str'>

```
type(17)
```

<class 'int'>

```
type(1.414)
```

<class 'float'>

Aber:

```
type("17")
```

<class 'str'>

Und:

```
type('Hallo Welt!')
```

<class 'str'>

`Strings` können entweder in einfachen (' ') oder doppelten (" ") Hochkommata eingeschlossen sein.

Variablen

- Eine **Variable** ist ein »Container«, der sich auf einen Wert bezieht
- Durch die **Zuweisung** wird eine neue Variable erstellt, ihr wird ein Wert zugewiesen:

```
meldung = "Und nun etwas ganz anderes"
print(meldung)
n = 17
print(n)
n = 3.14159
print(n)
```

Doch Vorsicht:

```
plz = 02492
```

gibt eine Fehlermeldung! Aber probiert mal

```
oct = 0o235
print(oct)
hex = 0xff
print(hex)
bin = 0b110011
print(bin)
```

Type- Casting

```
a = "17"
b = int(a) + 4
print(b)
```

Folgende Casting-Kommandos stehen zur Verfügung:

- `int()` macht aus einem String oder einem Float einen Integer-Wert
- `float()` macht aus einem String oder einem Integer eine Fließkommazahl
- `string()` versucht aus allem, was ihm unter die Finger kommt, eine Zeichenkette zu machen

Beispiele

```
a = 17
b = float(a) + 2
print(b)
```

Hier greift das *Duck-Typing*, aus `2` wird ein *float*.

```
a = "4711"
b = int(a)*2
print(b)
```

Aber:

```
a = "4711"
b = a*2
print(b)
```

Gültige Variablennamen

Ein **Bezeichner** in Python ist ein Name um Variablen, Module, Klassen, Funktionen oder andere Objekte eindeutig zu benennen. Ein Bezeichner kann aus folgenden Zeichen bestehen:

- Großbuchstaben A-Z
- Kleinbuchstaben a-z
- Unterstrich _
- Die Zahlen 0 bis 9 (jedoch nicht an erster Stelle)

- **Groß- und Kleinschreibung** zählt, das heißt `myNumber` und `Mynumber` sind verschiedene Namen.
- Seit Python 3 wird **Unicode** unterstützt. Somit kann der Bezeichner auch Unicode-Zeichen enthalten. Die Länge eines Bezeichners ist nicht begrenzt. Das heißt, daß dies gültige Bezeichner sind:

```
maximum_height_from_january_1920_to_december_2017 = 100
υψος = 10
μεγιστη_υψος = 100
```

Konventionen

Im **PEP8** *(Python Enhancement Proposal 8)* gibt es einen *Style Guide* für Python-Code. Der empfiehlt (unter anderem):

- Den Unterstrich als Worttrenner: `maximum_height`
- Also kein CamelCase: `maximumHeight`
- Variablen und Funktions-/Methodennamen beginnen mit einem Kleinbuchstaben und Klassen mit einem Großbuchstaben
- Konstanten werden komplett großgeschrieben: `HEIGHT`

Operatoren

- Die Operatoren `+`, `-`, `*`, `/` und `**` stehen für Addition, Substraktion, Multiplikation, Division und Potenzen. Doch Vorsicht in Python 2, dort ergibt

```
minute = 59
minute/60
```

nicht unbedingt das, was Ihr erwartet, besser ist

```
minute = 59
minute/60.0
```

Ganzzahl-Division und der Modulo-Operator

- In Python 3 wird dieses Verhalten der Integer-Division durch diesen Operator `//` erreicht:

```
minute = 59
minute//60
```

- Als Gegenstück gibt es in Python 2 *und* Python 3 den Modulo-Operator `%`

```
minute = 59
minute%60
```

Rangfolge von Operatoren

- Klammern `( )` haben den höchsten Rang
- Danach Exponenten `**`
- Danach Multiplikation `*` und Division `/`
- Danach Addition `+` und Subtraktion `-`
- Operatoren gleichen Ranges werden von links nach rechts abgearbeitet.
- Ich mache mir darüber aber selten Gedanken, im Zweifel verwende ich »Sicherheitsklammern«.

String-Operatoren

- Python-Strings können mit `+` und `*` umgehen, *nicht* jedoch mit `-` und `/`:

```
erster = "pangalaktischer "
zweiter = "Donnergurgler"
print(erster + zweiter)


print("Spam"*3)
```

Kommentare

Kommentare beginnen mit einem Doppelkreuz `#`, alles was hinter `#` steht, wird vom Interpreter ignoriert:

```
# Das ist ein ganzzeiliger Kommentar
y = math.cos(x)  # Hier wird der Cosinus von x berechnet
```

Einige Kommentare zu Beginn einer Programmdatei haben eine besondere Bedeutung:

```
#!/usr/local/bin/python
# coding=utf-8
```

Funktionen

- Funktionen können *mit* und/oder *ohne* Parameter aufgerufen werden.
- Mehrere Parameter werden durch Kommata getrennt.

```
import turtle
import math
tess = turtle.Turtle()
tess.penup()
x = 2.5
y = math.cos(x)
print("I got it, Babe, the cosine from x is ", y)
```

Funktionen erstellen

(Selbstgeschriebene) Funktionen werden mit `def` definiert:

```
def search_spring():
    print("Veronika, der Lenz ist da.")
    print("Die Mädchen singen trallala.")

search_spring()
```

Der Doppelpunkt `:` am Ende der ersten Zeile ist genau so wichtig, wie die Einrückungen darunter!

Funktionen können andere Funktionen aufrufen:

```
def sing_twice():
    search_spring()
    search_spring()

sing_twice()
```

Parameter

Stellt Euch diese Funktion vor:

```
def print_twice(peter):
    print(peter)
    print(peter)
```

Und versucht dann folgendes:

```
print_twice("Spam")
print_twice(127)
import math
print_twice(math.pi)
```

Funktionen mit mehreren Parametern

```
def mult3(a, b, c):
    ergebnis = a*b*c
    print("Die Multiplikation ergibt: ", ergebnis)

mult3(2, 3, 4)
```

Oder:

```
def mult_word(s, a):
    print(s*a)

mult_word("SPAM", 3)
```

Funktionen mit optionalen Parametern

```
def print_vektor(x, y, z=0):
    if z == 0:
        print(x, y)
    else:
        print(x, y, z)

print_vektor(3, 5)
print_vektor(6, 6, 6)
```

Innerhalb einer Funktion sind Variablen und Paramter **lokal**:

```
my_supernumber = 333

def add_number(my_supernumber):
    print(my_supernumber)
    my_supernumber += my_supernumber
    print("The number of the beast is ", my_supernumber)

add_number(my_supernumber)
print("But my Supernumber is still ", my_supernumber)
```

Anmerkung

- `x += n` ist eine Abkürtzung für `x = x + n`.
- Das gilt auch für `x -= n`, `x *= n` und `x /= n`.

Funktionen mit Rückgabewert

```
def double(x):
    return(2*x)

x = 10
y = double(x)
print(y)
```

- `return()` ist die Schlüsselfunktion!
- Ähnlich wie `print` konnte in Python 2.x auch `return` ohne Klammern aufgerufen werden.

Import von Modulen

Erlaubt sind:

```
import math
import numpy as np
from mygames_framework import Sprite
```

Verboten ist:

```
from numpy import *
```

Auch wenn Ihr das häufig in der Literatur seht, das verschmutzt nur den Namensraum!!!

Ein Template für die Schildkröte

```
import turtle as t

wn = t.Screen()
wn.colormode(255)
wn.bgcolor(43, 62, 80)
wn.setup(width = 600, height = 600)
wn.title("Ein Super-Duper Turtle-Programm")

alex = t.Turtle()

# Hier kommt jetzt Euer Programm-Code hin

wn.mainloop()
```

Eine selbstgeschriebene Funktion mit der Turtle

```
def quadrat(t):
    for i in range(4):
        t.forward(100)
        t.left(90)

alex.pensize(2)
alex.pencolor("red")
quadrat(alex)
```

Es können aber auch mehrere Schildkröten diese Funktion benutzen:

```
berta = t.Turtle()
berta.pensize(2)
berta.pencolor("white")
berta.goto(-100, 0)

quadrat(berta)
```

Und wie wäre es mit einem Polygon?

```
def polygon(t, n, length):
    angle = 360.0/n
    for i in range(n):
        t.forward(length)
        t.left(angle)

polygon(alex, 7, 70)
```

Oder mit einem Kreis?

```
import math

def circle(t, r):
    circum = 2*math.pi*r
    # n = 50
    n = int(circum/3) + 1
    length = circum/n
    polygon(t, n, length)
```

Boolsche Ausdrücke

Python kennt diese boolschen Operatoren:

- `x == y`: x ist **gleich** y
- `x != y`: x ist **ungleich** y
- `x > y`: x ist **größer** y
- `x < y`: x ist **kleiner** y
- `x >= y`: x ist **größer gleich** y
- `x <= y`: x ist **kleiner gleich** y

Logische Operatoren

- Es gibt drei logische Operatoren, mit denen boolsche Ausdrücke kombiniert werden können, `and`, `or` und `not`.
- Das Ergebnis von boolschen und logischen Operatoren ist immer `True` (wahr) oder `False` (falsch).

Boolsche Funktionen

Funktionen können auch Wahrheitswerte liefern:

```
def ist_teilbar(x, y):
    if x%y == 0:
        return(True)
    else:
        return(False)
```

oder:

```
def is_even(x):
    if x%2 == 0:
        return(True)
    else:
        return(False)
```

Verzweigungen

- Mit Hilfe der logischen und boolschen Operatoren können **Verzweigungen** programmiert werden:

```
if x > 0:
   print("x ist positiv")
else:
    print("x ist negativ oder null")
```

Oder auch:

```
if x%2 == 0:
    print("x ist gerade")
else:
    print("x ist ungerade")
```

Verkettete Bedingungen

Manchmal gibt es mehr als zwei Möglichkeiten:

```
if x < y:
    print("x ist kleiner als y")
elif x > y:
    print("x ist größer als y")
elif x == y:
    print("x ist gleich y")
else:
    print("Das Ende des Universums ist nahe!")
```

`elif` steht für `else if`.

Rekursion

Eine Funktion darf nicht nur eine andere Funktion, sondern auch sich selber aufrufen. Dies nennt man eine **Rekursion**:

```
def countdown(n):
    if n <= 0:
        print("Whammm …      !!")
    else:
        print(n)
        countdown(n-1)
```

Schleifen

Die `while`-Schleife ist eigentlich die einzige Schleife, die man braucht. Beispiel:

```
def countdown(n):
    while n > 0:
        print(n)
        n -= 1
    print(""Whammm …      !!")

countdown(10)
```

Die Endlos-Schleife

```
while True:
    print("From here to internity")
    # Hektische Suche nach dem Kill-Befehl
```

Die Endlos-Schleife ist aber nicht immer ein Programmfehler, sie wird zum Beispiel benutzt

- bei einem Server (der soll schließlich »immer« laufen)
- bei GUIs und/oder Spielen (warte auf Benutzer-Eingaben)

Callback (Ruf uns nicht an, wir rufen zurück)

```
def exit_prog():
    global keepGoing
    keepGoing = False

t.listen()
t.onkey(exit_prog, "Escape") # Escape beendet Programm

keepGoing = True
while keepGoing:
    pass  # Mache irgendetwas
```

(Beispielprogramm: `particle1.py` in `turtlepy`.)

Die `for`-Schleife

Python kennt noch die `for`-Schleife:

```
for i in range(10):
    print(i)
```

Der Endwert ist »exklusiv« (mathematisch gesprochen wird das halboffene Intervall `[0 … 10[` abgearbeitet, das heißt die Schleife zählt von Null bis Neun.

Die `for`-Schleife kann natürlich auch rückwärts zählen:

```
for i in range(10, 0, -1):
    print(i)
```

- Beachtet dabei, daß auch hier der Endwert exklusiv ist, die Schleife also von 10 bis 1 rückwärts zählt.
- Es gibt auch noch eine Abwandlung der `for`-Schleife für Strings und Listen. Dazu später mehr.

Die `range()`-Funktion

```
for identifier in range([start, ] stop [, step])
```

Dabei gilt:

- Alle Parameter sind Integer
- Alle Parameter können positiv oder negativ sein
- Wie alles in Python beginnt der Index mit `0`, daher ist der Stop-Wert »exklusiv«:

```
for i in range(5, 10):
    print i
```

Die `range()`-Funktion (2)

In Python 2.x gab es noch eine `xrange()`-Funktion. Der Unterschied war:

- `range()` gab als Ergebnis eine Liste zurück
- `xrange()` gab als Ergebnis einen `iterator` zurück

In Python 3 wurde `xrange()` zu `range()` und das originale `range()` als veraltet erklärt *(deprecated)*, das heißt `range()` liefert nun immmer einen `iterator` zurück.

Das war es mit den Grundlagen

Ihr kennt nun die grundlegenden Abläufe eines Programms:

- Sequenzen (werden von oben nach unten abgearbeitet)
- Verzweigung (entweder dies oder das)
- Schleifen (Mehrfachdurchläufe)

Das war es mit den Grundlagen (2)

Und die wichtigsten einfachen Datentypen:

- Zahlen (Integer und Float)
- Zeichenketten in ihrer einfachsten Form (sie werden noch einmal bei den Datenstrukturen behandelt)
- Boolsche Werte und Ausdrücke

Somit seid Ihr in der Lage, einfache Python-Programme selber zu schreiben.

Ausblick auf Modul 3

Hier geht es um Datenstrukturen:

- Strings (noch einmal, denn Zeichenketten sind sowohl Datentypen wie auch Datenstrukturen)
- Dateien (Lesen und Schreiben, Text, JSON, XML, CSV)
- Listen (die wichtigsten Datentypen in Python überhaupt)
- Dictionairies (Hash-Tables)
- Tupel

Fragen?