Skriptsprachen

Python Grundlagen

Kai Dührkop

Lehrstuhl fuer Bioinformatik
Friedrich-Schiller-Universitaet Jena
kai.duehrkop@uni-jena.de

21. September 2015

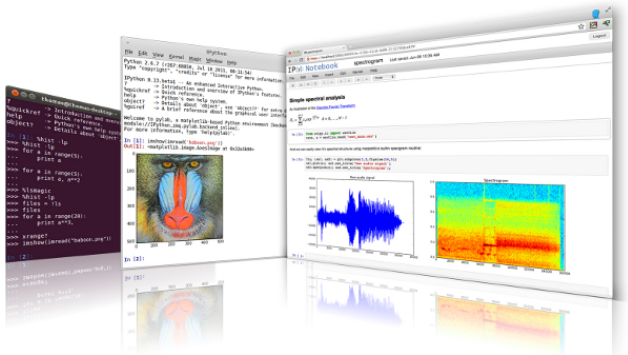
Python

Warum Python?

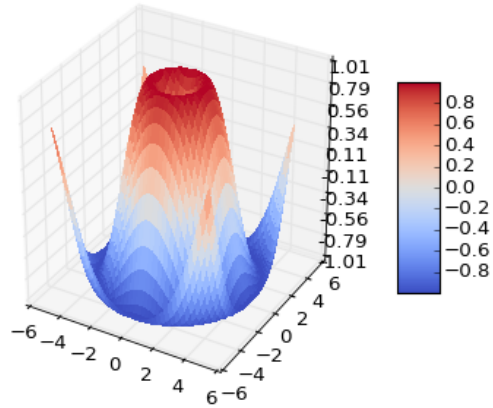
Warum Python?

- eine der am meisten genutzten Skriptsprachen
- populär bei wissenschaftlichen Anwendungen
- verbreitet in der „Linux Welt“

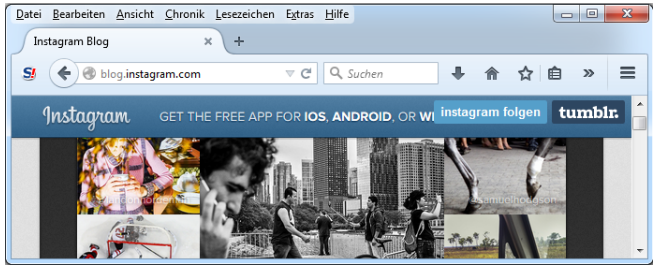
Wofür benutzt man Python?



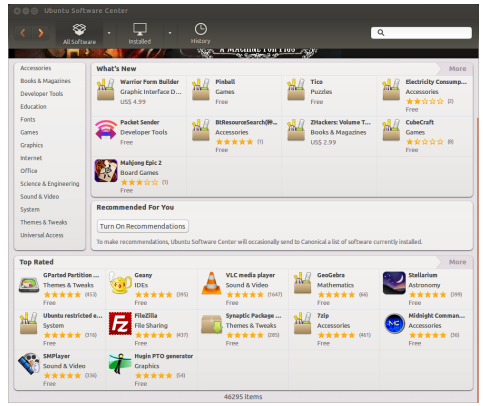
ad-hoc & shell-skripte



numerische & statistische Berechnungen, Plotten



Webapplikationen



GUIs

Grober Ablauf des Pythonkurses

- Grundlagen von Python
- Umgang mit der Python-Shell
- objektorientierte Programmierung
- **funktionale** Programmierung
- wissenschaftliches Arbeiten mit Python: numpy, scipy, scikit-learn, ...
- plotten!
- Datenformate: CSV, XML, JSON
- Webanwendungen: REST, SOAP

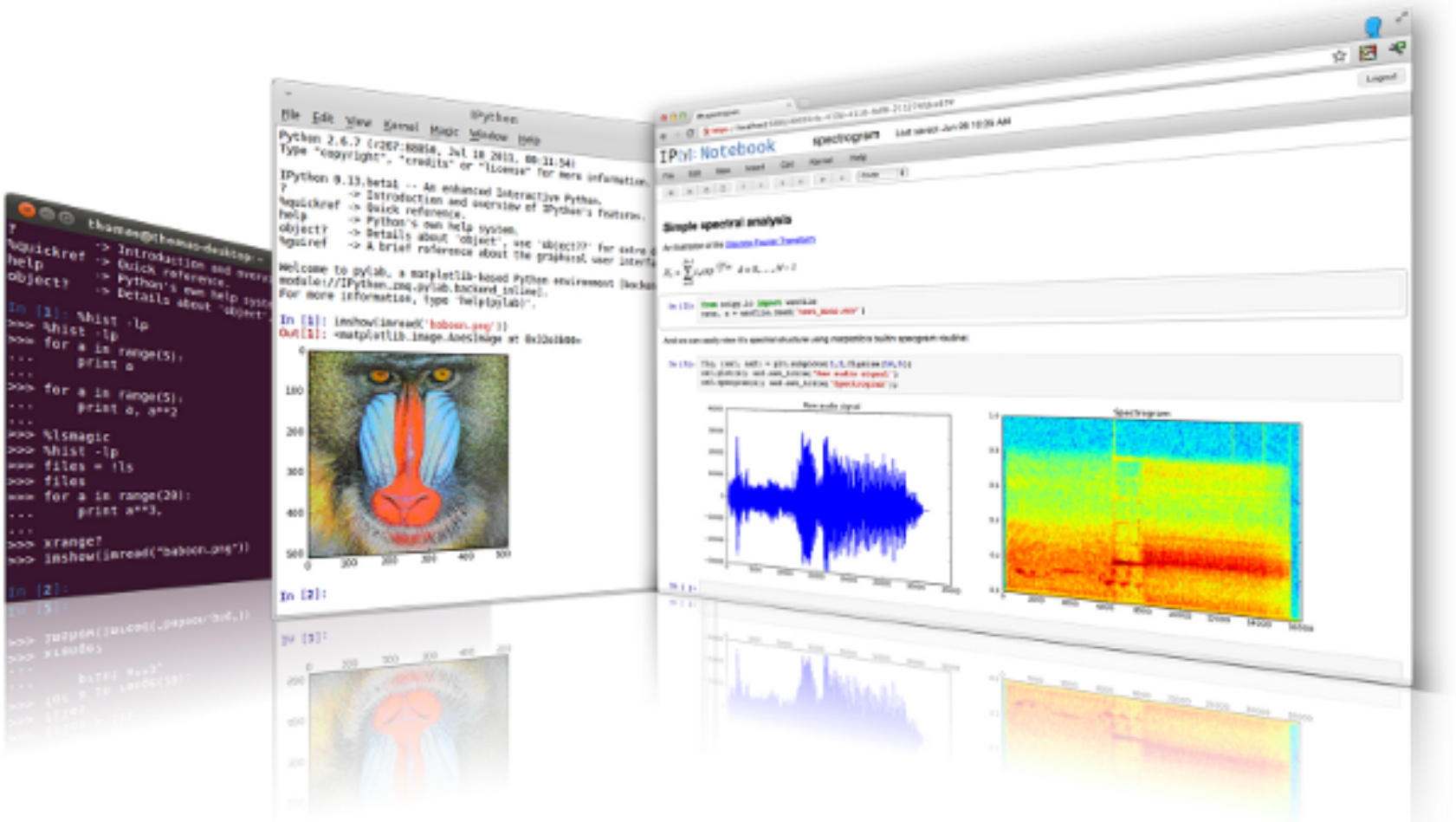
Was ist eine Skriptsprache

- REPL (read-eval-print-loop)
- dynamische Typisierung
- Interpreter statt Compiler
- schwammige Definition. Grenzen zwischen Hoch- und Skriptsprachen verschwimmen zunehmend
- Prototyping, Ad-hoc Skripte, Pipelines, GUIs, Webapplikationen

Was ist eine Skriptsprache

- **REPL (read-eval-print-loop)**
- dynamische Typisierung
- Interpreter statt Compiler
- schwammige Definition. Grenzen zwischen Hoch- und Skriptsprachen verschwimmen zunehmend
- Prototyping, Ad-hoc Skripte, Pipelines, GUIs, Webapplikationen

ipython



Was ist eine Skriptsprache

- REPL (read-eval-print-loop)
- **dynamische Typisierung**
- Interpreter statt Compiler
- schwammige Definition. Grenzen zwischen Hoch- und Skriptsprachen verschwimmen zunehmend
- Prototyping, Ad-hoc Skripte, Pipelines, GUIs, Webapplikationen

Exkurs: Dynamische, aber starke Typisierung

```
x = 5
# x ist vom Typ Integer
x = 5.0
# nun ist x vom Typ Float
x = "5"
# folgendes ist beispielsweise nicht erlaubt
x + 1 # x ist ein String!
```

Was ist eine Skriptsprache

- REPL (read-eval-print-loop)
- dynamische Typisierung
- **Interpreter statt Compiler**
- schwammige Definition. Grenzen zwischen Hoch- und Skriptsprachen verschwimmen zunehmend
- Prototyping, Ad-hoc Skripte, Pipelines, GUIs, Webapplikationen

Interpretierter Code

```
# this program won't crash
if 5 < 8:
    print "Hallo Welt"
else:
    dieseMethodeGibtEsDochGarNicht()
```

Python: Einführung

- Grundlegende Datenstrukturen
- Kontrollstrukturen
- Module
- IO
- Reguläre Ausdrücke

Section 2

Grundlegende Datenstrukturen

Grundlegende Datenstrukturen

```
5                       # Int
5.2                     # Float
True, False             # Booleans
"some String"           # String
bytearray("abc")        # bytearray
[1,2,"3"]               # Array
(1,2,"3")               # Tuple
{1,2,3}                 # Set
{"a": 1, "b": 2}        # Dictionary
```

Grundlegende Datenstrukturen: Numerische Datentypen

```
x = 5 # int (darf beliebig gross sein)
y = 5.0 # float
x += 1 # wie gehabt
# aber keine Dekrement/Inkrement Operatoren!

# Python 2:
5 / 2 # ergibt 2
# Python 3:
5 / 2 # ergibt 2.5

# immer moeglich:
5 // 2 # ergibt 2
5.0 // 2.0 # ergibt 2.0
```

Grundlegende Datenstrukturen: Numerische Datentypen

```
x = 3
y = x**2 # square of x
z = x**(-2) # squareroot of x
abs(-5) # absolute
round(2.3) # 2.0

# additional functions in math module
import math

math.exp(5) # e**5
math.log(3) # logarithm
```

Booleans

```
a = True
b = False
a == b
a != b
5 > 2
# auch folgendes moeglich
5 < x < 9
x < 5 and x < 9
x == 1 or x == 2
```

Booleans

- False, 0, 0.0, leere Sequenzen etc. zählen als Unwahr
- Logikoperatoren **and** und **or** sind **lazy**. D.h. sie evaluieren den zweiten Term nur, wenn das Ergebnis nicht schon nach dem ersten Term feststeht

```
# lazy Operatoren
x = True or dieseFunktionGibtsNicht()

# R ckgabewert muss kein Boolean sein
y = 5 and "Hallo" # y ist "Hallo"
```

Sequenztypen

```
a = [1,2,3,4,5,6,7,8]
b = "xyz"
c = ("a", "b", "c")
d = {1, 2, 3}

# pruefen ob Element enthalten ist
5 in a #=> True
2 not in c #=> True

len(a) # Laenge
max(a) # groesstes Element
min(a) # kleinestes Element
a.count(2) # Anzahl der Vorkommen
```

random-Access Sequenztypen

```
a = [1,2,3,4,5,6,7,8]
b = "xyz"
c = ("a", "b", "c")

a.index(5) # index of 5 in a

# Elementzugriff
a[0] # erstes Element
b[-1] # letztes Element
b[-2] # vorletztes Element
```

random-Access Sequenztypen

Slicing

```
a = [1,2,3,4,5,6,7,8]
b = "xyz"
c = ("a", "b", "c")

c[0:2] # neues Tuple mit ersten Zwei Elementen
a[1:-1] # neuer String ohne den ersten Buchstaben
a[0:-1:2] # jedes zweite Element
b[:] # alle Elemente (Kopie)
```

mutable Sequenztypen

```
a = [1,2,3,4,5,6,7,8]
b = {1,2,3}
c = bytearray() # mutable string type

# Konkatenation
a += [9,10,11]
b += {4, 5}
c += "some String"
# Parameter darf beliebige Sequenz sein
a += (1,2)
b += [5,6]

# Element loeschen
a.pop()
b.remove(2)
```

mutable random-access Sequenztypen

```
a = [1,2,3,4,5,6,7,8]
b = bytearray()

a.pop(2) # loesche drittes Element
del a[2] # Alternative

#slicing
a[0:3] = (5,6,7) # ersetze die ersten 3 Elemente
del b[0:3] # loesche die ersten 3 Elemente
```

sets

- A **set** is an unordered collection with no duplicate elements.

Set operations

```
s = {1,2,1,2,3}          #elements: [1,2,3]
s = set([1,2,1,2,3]) # the same

a | b     #union:        elem. in a or b
a & b     #intersect:    elem. a and b
a - b     #difference:   elem. in a but not in b
a ^ b     #complement:   elem. a or b but not both
```

Strings

String Literale

```
normal = "Ein String"
unicode = u"Ein Unicode String" # python 2!
raw = r"Ein String \t ohne Steuerzeichen\n"
multiline = """Ein
mehrzeiliger
String"""
```

Strings

String Konkatenation

```
x = "Dies ist eine"
x += "teure"
x += "Operation"
# nutze stattdessen join:
x = "".join([x, "effiziente", "Operation"])
```

bytearrays (weniger optimal)

```
x = bytearray("Dies ist eine")
x += "effiziente"
x += "Operation"
x = str(x)
```

Strings

join und split

```
xs = ["1","2","3"]
string = ",".join(xs)
#=> "1,2,3"
ys = string.split(",")
xs == ys #=> True
```

Format String

C-Style Format String

```
"room %03d, distance %.2f m" % (7, 5.23711)
#=> "room number 07, distance 5.24 m"

"%s strings, %d ints, %f floats" % ("a", 3, 3.7)
```

Dictionaries

```
map = {"a": 5, "b": 8}
map["a"] #=> 5
map["c"] #=> wirft eine Exception!
# gibt den Wert fuer "c" zurueck, oder Default 0
map.get("c", 0) #=> 0
map["c"] = 7
map.get("c", 0) #=> 7

"c" in map #=> True
del map["a"] # loesche Element
```

Exkurs: Mutable und Immutable

```
x = [1,2,3]
y = x
x += [4,5]
x #=> [1,2,3,4,5]
y #=> [1,2,3,4,5]

a = "Hallo"
b = a
a += "Welt"
a #=> "HalloWelt"
b #=> "Welt"
```

Exkurs: Dynamische, aber starke Typisierung

```
x = 5
# x ist vom Typ Integer
x = 5.0
# nun ist x vom Typ Float
x = "5"
# folgendes ist beispielsweise nicht erlaubt
x + 1 # x ist ein String!
```

Exkurs: Konvertierungsfunktionen

```
x = 5
x = str(5)

x = float(x)
x = int(x)

x = bytearray("Welt")
x = set([1,2,3,4])
```

Zusammenfassung

Typisierung

- starke Typisierung (z.B. Integer ungleich Float)
- mehrere Wahrheitswerte (0, [], , False sind unwahr)

Sequenztypen

- Mutable und Immutable Sequenztypen
- Random-Access (Liste, String) und Sets
- teilen sich gemeinsame Funktionen
- slicing!
- dictionaries (Hashmaps). Vorsicht bei ungeprüftem Zugriff

Section 3

Kontrollstrukturen

Kontrollstrukturen

- Bedingungen
- Schleifen
- Funktionen

Bedingungen

If-Bedinging

```
if 3 < x < 8:
  print x
elif x < 0:
  print " is negative"
else:
  print "x not in range"
```

- python ist whitespace-sensitiv
- keine Klammern oder Ends. Stattdessen: Einrückung
- Einrückungen müssen innerhalb eines Files konsistent sein!
- Empfohlen: 4 Leerzeichen, keine Tabs

Bedingungen

Ternärer Operator

```
x = 5 if y < 0 else -5
```

Schleifen

While-Schleife

```
while x < 5:
    x += 1
    print x
```

For-Schleife

```
xs = [1,2,3,4,5]
for x in xs:
    print x
```

Schleifen

Iteriere Index-Wert Paare:

```
for index, x in enumerate([1,2,3,4]):
  print "%d at %d" % (index, x)
```

Iteriere in Intervallen:

```
for i in xrange(1,10):
  print i # iteriere von 1 bis 9
```

Schleifen

Iteriere über Schlüssel von Dictionaries:

```
ys = {"a": 1, "b": 2}
for key in ys:
    print key
```

Iteriere Schlüssel-Wert Paare:

```
for key, value in ys.iteritems():
    print "%s is key, %d is value" % (key, value)
```

Schleifen

break

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print "%d = %d / %d" % (n/x, n, x)
            break
    else:
        print "%d is a prime number" % n
```

- else-Fall in Schleifen wird ausgeführt, wenn die Schleife ausläuft, nicht aber, wenn sie per break unterbrochen wird

Funktionen

Funktionen definieren

```
def fibonacci(x):
    if x <= 1:
        return x
    return fibonacci(x-1) + fibonacci(x-2)
```

Funktionen

lokale Funktionen definieren

```
def power(a, b):
    def pow(a, b, acc):
        if b==0:
            return acc
        else:
            return pow(a,b-1,acc*a)
    return pow(a,b,1)
```

Tuple-Unpacking

Mehrere Rückgabewerte

```
def divmod(a,b):
    return (a / b, a % b)

quotient, rest = divmod(a, b)
```

Parallelzuweisung

```
def fibonacci(n):
    a = 1
    b = 1
    for i in xrange(1,n):
        a,b = b, a+b
    return a
```

Zusammenfassung

Schleifen

- for-Schleife am wichtigsten
- xrange für Intervalle, enumerate, iteritems
- konsistente Einrückungen beachten

Funktionen

- Mehrfache Rückgabewerte über Tuple
- Parallelzuweisung/Tuple-unpacking

Exkurs: Arbeiten mit ipython

- mit ? und ?? am Ende eines Namens ruft Hilfe auf
- Alternativ auch help(“begriff“)
- Autovervollständigung mit TAB
- x.[TAB] listet alle Methoden von x auf

Section 4

Module

Package Structure

```
package/                        #top level package
    __init__.py
    subpackage/                 #subpackage
        __init__.py
        module.py
        module2.py
```

- a module is a python file
- the interpreter interprets directories on the **sys.path** with an **__init__.py** as package of modules
- **__init__.py** is loaded first, may contain initialization code

Importing and using modules

```
import sys                      #import sys module
from os import chdir            #import chdir function
                                #from os module


chdir(sys.argv[1])
```

- module import: call functions/variables by module & name
- function/variable import: call functions/variables by name

CAUTION!

```
from package import *
```

- **does not import all functions of a module!**
- imports functions defined in:

__init__.py

```
__all__ = ['function', 'function2']
```

Zusammenfassung

Module

- Importieren von Modulen und Paketen
- from ... import * um mehrere Funktionen zu importieren
- Viele Module in Standardbibliothek vorhanden!

Section 5

IO

- **raw_input()** reads strings
- **input()** evaluates and casts input appropriately

We type '42':

```
a = raw_input() # a is the string '42'
b = input()     # b is 42
```

- **str()** casts to 'human-readable' string
- **repr()** casts to 'computer-readable' string
- **print** implicitly uses **str()**

String representation

```
print 1.0/7.0          #'0.142857142857'
print str(1.0/7.0)     #'0.142857142857'
print repr(1.0/7.0)    #'0.14285714285714285'
```

Files

- Filehandles sind Betriebssystemressourcen, die angefordert und freigegeben werden müssen
- Problem: Funktion darf zwischen Öffnen und Schließen einer Datei nicht abbrechen
- Lösung: Das Öffnen wird in einem with-Block durchgeführt. Am Ende des with-Blocks wird das Filehandle automatisch wieder geschlossen

Reading a text file

```
lines = []
with open("someFile.txt") as fhandle:
    for line in fhandle:
        lines.append(line)
```

Files

- zwei Arten von Files: binary (b) und textfiles
- modes: appending (a), writing (w), reading (r)
- Beispiel: wb zum Schreiben eines binary files
- + Öffnet eine Datei zum Lesen UND Schreiben. r+ startet am Anfang, a+ am Ende

Reading binary file

```
bytes = ""
with open("someFile.dat", "rb") as fhandle:
    bytes = fhandle.read()
```

Files

Writing files

```
numbers = [1, 2, 3, 4]
with open("someFile.txt", "w") as fhandle:
    for num in numbers:
    fhandle.write(num)
fhandle.write("\n")
```

Appending to files

```
with open("someFile.log", "a") as fhandle:
    fhandle.write("Some new error message\n")
```

Verzeichnisse

Ordnerinhalte listen

```
from glob import glob

files = glob("/home/kaidu/.local/*")
```

Ordner anlegen

```
from os import path
from os import mkdir

if not path.exists("someDir"):
  os.mkdir("someDir")
```

Zusammenfassung

IO

- open in with Statement zum Öffnen von Dateien
- Dateien lassen sich leicht zeilenweise einlesen
- os.path für Funktionen um mit Dateipfaden zu arbeiten
- os und darin enthaltene Module um auf Dateisystem zu operieren

Section 6

Reguläre Audrücke

Reguläre Audrücke

Kompilieren eines reg. Ausdrucks

```
import re
reg = re.compile(r"my number is (\d+)")
```

Prüfen ob ein Ausdruck matcht

```
reg = re.compile(r"my number is (\d+)")
# matcht der Ausdruck von Beginn an?
if reg.match("my number is 123")
  print "Treffer"

# matche ersten Substring
match = reg.search("Hi, my number is 123")
if match:
  number = int(match.group(1))
```

Reguläre Audrücke

Substitution

```
r = re.compile(r"[aouei]", re.IGNORECASE)
str = "Ein Test"
r.sub("*", str) #=> "**n T*st"
```

Splitten

```
r = re.compile(r"\s*,\s*")
str = "a, b,c,  d,  e"
r.split(str) #=> ["a", "b", "c", "d", "e"]
```

Iterieren über mehrere Matches

```
r = re.compile(r"(\d+)\.\d+")
str = "5.4 und 2.6, aber auch 3.1"
for match in r.finditer(str):
    num = match.group(1)
print num
```

Zusammenfassung

Reguläre Ausdrücke

- sind im Modul re definiert
- selbe Syntax wie in Perl
- am besten Raw-Strings benutzen um nicht alles zu escapen

Section 7

Kommandozeilen Programme

Kommandozeilenprogramme

Main-Block

```
if __name__ == '__main__':
    # ....
```

- wird nur aufgerufen, wenn Skript direkt über den Python Interpreter gestartet wird
- wird NICHT aufgerufen, wenn Modul innerhalb einer anderen Library geladen wird

Kommandozeilenprogramme

Kommandozeilenargumente

```
import os
if __name__ == '__main__':
    if len(os.argv) < 3:
    print "expect 2 filenames as parameter"
    else:
    someMainFunction(os.argv[1], os.argv[2])
```

- os.argv enthält Kommandozeilenargumente
- erstes Element ist immer der Programmname

Kommandozeilenprogramme

Kommandozeilenargumente

```
python FILENAME          # fuehrt Programm aus
python -c "command"      # fuehrt Python-Codezeile aus
```