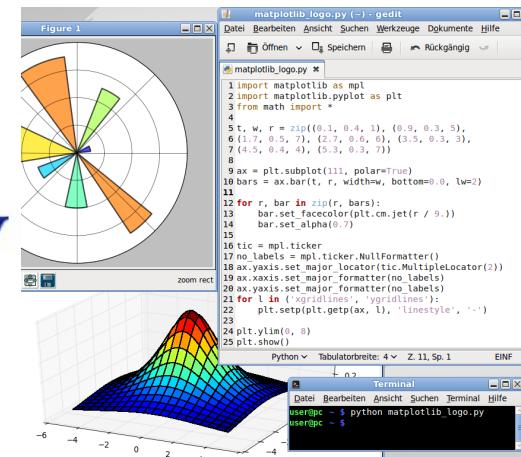


Python und Numpy



Bachelor Medieninformatik
Wintersemester 2019/20

Prof. Dr.-Ing. Kristian Hildebrand
khildebrand@beuth-hochschule.de

Lernziele

- Überblick in die Programmiersprache Python
 - Datentypen
 - Container
 - Funktionen
 - Klassen
- Einführung in Numpy
 - Datentypen
 - Zugriffe
 - Mathe

Programmiersprache
Python



3.6.

Python

Wer hat schon damit rumgespielt?
Gibt es schon Fragen?

- umfangreiches Ökosystem
 - Console, IDE, Anaconda
 - pip, conda / site-packages
 - dient häufig als Skript-Sprache für c++ Bibliotheken (wrapper)
- Allgemeine Infos
 - universelle, interpretierte höhrere Programmiersprache / Skriptsprache
 - unterstützt mehrere Programmierparadigmen
 - objektorientiert (vollständig unterstützt) und auch funktional (in einzelnen Aspekten)
 - dynamische Typisierung

Typüberprüfungen finden zur Laufzeit statt

Dynamische Typisierung in Python

```
>>> x = 1      # a enthält durch Zuweisung eine ganze Zahl
>>> x += 0.1   # addiert die Gleitkommazahl 0.1 und
               # legt neuen Wert (mit anderem Typ) in x ab

>>> x.lower() # Failed: x ist keine Zeichenkette
               Fehlermeldung:
               Traceback (most recent call last): File "<stdin>", line 1,
               in <module> AttributeError: 'float' object has no attribute 'lower'

>>> x          # gibt den Wert von a aus 1.1

>>> x = "FOOBAR"

>>> x += 1     # Failed: Inhalt von x ist jetzt ein String
               # Fehlermeldung: Traceback (most recent call last): File
               # "<stdin>", line 1, in <module> TypeError: cannot
               # concatenate 'str' and 'int' objects

>>> x.lower()
'foobar'
```

Datentypen

Datentypen in Python

- Duck-Typing Prinzip (“*When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*”)
- Integer, Float, Boolean, Complex
- String und Stringoperationen
- In Python ist alles ein Objekt
- Vergleichsoperator

`==` überprüft zwei Objekte auf Wertgleichheit `is` überprüft zwei Objekte auf Identitätsgleichheit

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> a is b
False
>>> a == b
True
```

Datentypen in Python / Zahlen und Booleans

- Zahlen arbeiten wie erwartet

```
x = 3
print type(x) # Ausgabe: "<type 'int'>"
print x # Ausgabe: "3"
print x + 1 # Addition; prints "4"
print x - 1 # Subtraktion; prints "2"
print x * 2 # Multiplikation; prints "6"
print x ** 2 # Exponentiation; prints "9"
x += 1
print x # Ausgabe: "4" x *= 2
print x # Ausgabe: "8" y = 2.5
print type(y) # Ausgabe: "<type 'float'>"
print y, y + 1, y * 2, y ** 2 # Ausgabe: "2.5 3.5 5.0 6.25"
```

- Booleans – Logische Operationen – Jetzt bitte selbst ausprobieren
(and, or, not)

Datentypen in Python / Strings

■ String Basisoperationen

```
hello = 'hello' # String literals can use single quotes

world = "world" # or double quotes; it does not matter.

print(hello) # Prints "hello"

print(len(hello)) # String length; prints "5"

hw = hello + ' ' + world # String concatenation

print(hw) # prints "hello world"

hw12 = '%s %s %d' % (hello, world, 12) # sprintf style string
                                             # formatting

print(hw12) # prints "hello world 12"
```

Datentypen in Python / Strings

- Strings – weitere nützliche Methoden

```
s = "hello" print s.capitalize() # Capitalize a string; Ausgabe:  
# "Hello"  
  
print(s.upper()) # Convert a string to uppercase; Ausgabe: "HELLO"  
  
print(s.rjust(7)) # Right-justify a string, padding with spaces;  
# Ausgabe: " hello"  
print(s.center(7)) # Center a string, padding with spaces;  
Ausgabe: " # hello "  
  
print(s.replace('l', '(ell)')) # Replace all instances of one  
# substring with another;  
# Ausgabe: "he(ell)(ell)o"  
  
print(' world '.strip()) # Strip leading and trailing whitespace;  
# Ausgabe: "world"  
  
print(s[0:3]) # Ausgabe: 'hel'
```

Listen

Container in Python / Listen

- es gibt einige built-in Containertypen
 - Listen, Tuple, Sets und Dictionaries
- Listen:

```
xs = [3, 1, 2] # Liste erzeugen
print xs, xs[2] # Ausgabe:[3, 1, 2] 2
print xs[-1] # Negative indices count from the end of the list;
prints "2"
xs[2] = 'foo' # Lists can contain elements of different types
print xs # Prints "[3, 1, 'foo']"
xs.append('bar') # Add a new element to the end of the list
print xs # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop() # Remove and return the last element of the list
print x, xs # Prints "bar [3, 1, 'foo']"
```

Container in Python / Slicing

sehr wichtig, weil sehr sehr nützlich
häufig auch bei Numpy eingesetzt

- Listenzugriffe, Zugriff auf Unterlisten aka **Slicing (start:stop:step)**

```
nums = range(5) # range is a built-in function that
                 # creates a list of integers
print nums # Prints "[0, 1, 2, 3, 4]"
print nums[2:4] # Get a slice from index 2 to 4 (exclusive);
                # Ausgabe: "[2, 3]"
print nums[2:] # Get a slice from index 2 to the end;
                # Ausgabe: "[2, 3, 4]"
print nums[:2] # Get a slice from the start to index 2 (exclusive);
                # Ausgabe: prints "[0, 1]"
print nums[:] # Get a slice of the whole list;
                # prints ["0, 1, 2, 3, 4"]
print nums[:-1] # Slice indices can be negative;
                # prints ["0, 1, 2, 3"]
nums[2:4] = [8, 9] # Assign a new sublist to a slice
print nums # Prints "[0, 1, 8, 9, 4]"
```

Loops

Container in Python / Loops

- Loops – über jedes Element

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print animal
# Prints "cat", "dog", "monkey", each on its own line.
```

- Loops – über einen Index

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print '#%d: %s' % (idx + 1, animal)
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

```
animals = ['cat', 'dog', 'monkey']
for idx in range(0, len(animals)):
    print '#%d: %s' % (idx + 1, animals[idx])
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

Funktionen und Loops

- Programmieren Sie **jetzt** selbst eine Funktion die Quadrate berechnet und zurückgibt

```
# Funktionsdefinition: def <funktionsname>(param1, param2...)  
def squares(nums):  
    # do compute and return array of squares here  
    pass  
  
sqs = squares([1,2,3,4])  
print sqs
```

```
# Funktionen mit optionalen Parametern  
def hello(name, loud=False):  
    if loud:  
        print 'HELLO, %s!' % name.upper()  
    else:  
        print 'Hello, %s' % name  
hello('Bob') # Prints "Hello, Bob"  
hello('Fred', loud=True) # Prints "HELLO, FRED!"
```

Funktionen und Loops

Python 2.7

```
→ 1 def squares(nums):\n  2     sqs = []
```

[Edit code](#)

→ line that has just executed
→ next line to execute

< Back Program terminated Forward >

Frames Objects

Global frame

squares

function
squares(nums)

Visualized using [Online Python Tutor](#) by [Philip Guo](#)

[Online Python Tutor](#) supports Python 2.7 and 3.3, Java 7, JavaScript ES5 strict, TypeScript 1.4.1, and Ruby 2.2.2 with limited module imports and no file I/O. [Privacy and Terms of Use](#).

Copyright © [Philip Guo](#). All rights reserved.

Funktionen und Loops

- Man kann Loops auch vereinfachen – eine sog. *list comprehension* machen

```
sqs = []
for x in nums:
    sqs.append(x**2)
return sqs

# wird zu

sqs = [x ** 2 for x in nums]

# diese können auch if-Anweisungen enthalten

even_sqs = [x ** 2 for x in nums if x % 2 == 0]
```

Dictionaries

Container in Python / Dictionaries

- Dictionaries sind einfache *(key, value)* Stores ähnlich wie eine *Map* in Java oder ein *Object* in Javascript

```
d = {'cat': 'cute', 'dog': 'furry'} # Ein dictionary erzeugen
print d['cat'] # Dictionary eintrag Ausgabe: "cute"
print 'cat' in d # Gibt es den Key 'cat'? Ausgabe: "True"

d['fish'] = 'wet' # Value zum Key setzen
print d['fish'] # Ausgabe: "wet"
# print d['monkey'] # Fehler: KeyError: 'monkey' not a key of d

print d.get('monkey', 'N/A') # Element mit Defaultwert lesen,
Ausgabe: "N/A"
print d.get('fish', 'N/A') # Ausgabe: "wet" ?

del d['fish'] # Element aus Dictionary löschen
print d.get('fish', 'N/A') # 'fish' ist kein Key - Ausgabe: "N/A"
```

Container in Python / Dictionaries

- Loops über Dictionaries (über die Schlüssel)

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print 'A %s has %d legs' % (animal, legs)
# Ausgabe: "A person has 2 legs", "A spider has 8 legs", "A cat has 4 legs"
```

- Loops über Dictionaries (über die Schlüssel + deren Werte)

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.iteritems():
    print 'A %s has %d legs' % (animal, legs)
# Ausgabe: "A person has 2 legs", "A spider has 8 legs", "A cat has 4 legs"
```

- Comprehensions über Dictionaries

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print even_num_to_square # Ausgabe: "{0: 0, 2: 4, 4: 16}"
```

Sets

Container in Python / Sets

- Set ist eine ungeordnete Liste von Elementen

```
animals = {'cat', 'dog'}  
print 'cat' in animals # Ist Element in Set? Ausgabe: "True"  
print 'fish' in animals # Ausgabe: "False"  
animals.add('fish') # Element hinzufügen  
print 'fish' in animals # Ausgabe: "True"  
print len(animals) # Anzahl der Elemente im Set, Ausgabe: "3"  
animals.add('cat') # Element, dass es schon gibt hinzufügen, tut  
gar nichts  
print len(animals) # Ausgabe: "3"  
animals.remove('cat') # Element aus Set löschen  
print len(animals) # Ausgabe: "2"
```

Container in Python / Sets

- Loops über Sets verhalten sich wie Arrays
- man kann aber keine Annahme über die Abarbeitung machen

```
animals = {'cat', 'dog', 'fish'}  
for idx, animal in enumerate(animals):  
    print '#%d: %s' % (idx + 1, animal)  
# Ausgabe: "#1: fish", "#2: dog", "#3: cat"
```

- Comprehensions über Dictionaries

```
from math import sqrt  
nums = {int(sqrt(x))  
        for x in range(30)}  
print nums  
# Ausgabe: "set([0, 1, 2, 3, 4, 5])"
```

Einbinden von Bibliotheken

Tuples

Container in Python / Sets

- Tuples sind eine geordnete Liste von Elementen
- Unterschied zu Listen:
 - *Tuples können als Schlüssel in Dictionaries und als Elemente in Sets verwendet werden – Listen nicht!*

```
d = {(x, x + 1): x for x in range(10)}  
# Ein Dictionary mit einem Tuple als Schlüssel erzeugen  
t = (5, 6) # Tuple erzeugen  
print type(t) # Ausgabe: "<type 'tuple'>"  
print d[t] # Ausgabe: "5"  
print d[(1, 2)] # Ausgabe: "1"
```

Write code in Python 2.7

1

Visualize Execution

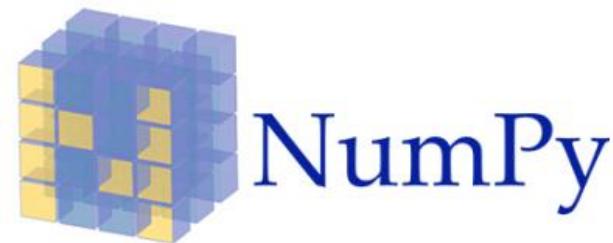
[Create test cases](#)

Online Python Tutor supports Python 2.7 and 3.3, Java 7, JavaScript ES5 strict, TypeScript 1.4.1,

The Zen of Python

>>> import this

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently. Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never. Although never is often better than **right** now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!



Bibliothek für wissenschaftliches
Rechnen und Datenanalyse

Numpy – Allgemeine Informationen

- bietet einfache Handhabung von Vektoren und Matrizen (allg. multi-dimensionalen Arrays)
- effiziente Methoden für numerische Berechnungen
- Numerik? - Was ist das?
 - beschäftigt sich mit Algorithmen und Methoden kontinuierliche Probleme mit Hilfe von Computer näherungsweise zu lösen
 - kontinuierlich? – wir haben hier meist diskrete Punkte/Daten oder Stichproben (Samples)
- Was ist das besondere an Numpy?
 - “NumPy address the slowness problem partly by providing multidimensional arrays and functions and operators that operate efficiently on arrays, requiring (re)writing some code, mostly inner loops using NumPy.”

Numpy – Allgemeine Informationen

- bietet einfache Handhabung von Vektoren, Arrays und Matrizen
- getyped und speichereffizient
- effiziente Methoden für numerische Berechnungen
- Computer Vision Paket OpenCV verwendet intern Numpy Arrays
 - dadurch schneller Verarbeitung möglich

Numerik? Was ist das?



Bsp. Videobearbeitung



Numpy Datentypen

- Numpy array ist ein '*Gitter*' von Werte desselben Typs
 - Achse 1: Zeile, Achse -2: Spalte

kurze Übung dazu

```
import numpy as np # Einbinden der Numpy Bibliothek

x = np.array([1, 2]) # Numpy wählt Datentyp aus
print x.dtype # Ausgabe: "int64"
x = np.array([1.0, 2.0]) # Numpy wählt Datentyp aus
print x.dtype # Ausgabe: "float64"
x = np.array([1, 2], dtype=np.int64) # Wir wählen Datentyp aus
print x.dtype # Ausgabe: "int64"
```

Numpy Arrays

- Anzahl der Dimensionen des Arrays wird in Numpy als Rang bezeichnet (*rank*)
- Form des Arrays ist das Tuple der Arraygröße in jeder Dimension (*shape*)

```
import numpy as np # Einbinden der Numpy Bibliothek

a = np.array([1, 2, 3]) # Ein 'rank 1' Array erzeugen
print type(a) # Ausgabe: "<type 'numpy.ndarray'>"
print a.shape # Ausgabe: "(3,)"
print a[0], a[1], a[2] # Ausgabe: "1 2 3"
a[0] = 5 # Wert eines Elementes ändern
print a # Ausgabe: "[5, 2, 3]"
b = np.array([[1,2,3],[4,5,6]]) # Ein 'rank 2' Array erzeugen
print b.shape # Ausgabe: "(2, 3)"
print b[0, 0], b[0, 1], b[1, 0] # Ausgabe: "1 2 4"
```

Numpy Arrays

- Numpy stellt eine Reihe von Funktionen zur Verfügung, um Arrays zu erzeugen

```
a = np.zeros((2,2)) # Erstellt ein Array von Nullen
print a # Ausgabe: "[[ 0.  0.] [ 0.  0.]]"

b = np.ones((1,2)) # Erstellt ein Array von Einsen
print b # Ausgabe: "[[ 1.  1.]]"

c = np.full((2,2), 7) # Erstellt ein Array von einem ggb. Wert
print c # Ausgabe: "[[ 7.  7.] [ 7.  7.]]"                                Was war das doch gleich?
d = np.eye(2) # Erzeugt eine 2x2 Identitätsmatrix
print d # Prints "[[ 1.  0.] [ 0.  1.]]"

e = np.random.random((2,2)) # Erzeugt ein Array mit Zufallszahlen
print e # eventuelle Ausgabe: "[[ 0.91940167  0.08143941] [ 0.68744134  0.87236687]]"

np.arange!!!
np.linspace!!!
```

Array Indexing
(sehr wichtig – Üben!)

Numpy Array Indexing

```
# Rang 2 Array der Form (3, 4) -- 3x4 Matrix
# [[ 1 2 3 4]
# [ 5 6 7 8]
# [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

Slicing nutzen, um eine Untermatrix zu extrahieren, die die ersten beiden Zeilen und die Spalten 1 und 2 (2x2 Matrix)

```
# [[2 3]
# [6 7]]
b = a[:2, 1:3]
```

Ein 'Slice' der Daten ist immer nur ein View / Reference auf die Daten - eine Modifikation ändert die Originaldaten

```
print a[0, 1] # Ausgabe: "2"
b[0, 0] = 77 # b[0, 0] sind die gleichen Daten wie a[0, 1]
print a[0, 1] # Ausgabe: "77"
```

Matrix an die Tafel und Zugriff alle zusammen!

Numpy Array Indexing

```
# Es gibt zwei Wege die 1.Zeile zu extrahieren
# 1. Mit einem Mix aus Index+Slicing (ergibt 1D Array)
# 2. Mit Slicing (ergibt 2D Array) - gleicher Rang wie Original

row_r1 = a[1, :]    # Rang 1 Ansicht von a
row_r2 = a[1:2, :]  # Rang 2 Ansicht von a
print row_r1, row_r1.shape # Ausgabe: "[5 6 7 8] (4,)"
print row_r2, row_r2.shape # Ausgabe: "[[5 6 7 8]] (1, 4)"

# Das gleiche gilt für Spalten:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print col_r1, col_r1.shape # Ausgabe: "[ 2 6 10] (3,)"
print col_r2, col_r2.shape # Ausgabe: "[[ 2] #[ 6] #[10]] (3, 1)"

a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
a[1:7:2] # Ausgabe: array([1, 3, 5])
```

Numpy Array Indexing

```
# Ein Beispiel für Integer Indexing
a = np.array([[1,2], [3, 4], [5, 6]])
# a hat die Form (shape) (3,) ← Hä???
print a[[0, 1, 2], [0, 1, 0]] # Ausgabe "[1 4 5]" ← Erzeugt Kopie!
# Erzeugt das gleiche Ergebnis
print np.array([a[0, 0], a[1, 1], a[2, 0]]) # Ausgabe:"[1 4 5]"
print np.array([a[0, 1], a[0, 1]]) # Ausgabe: "[2 2]"
```

```
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print a ← Wie sieht das Ergebnis aus?
# Erzeuge ein Array von Indices
b = np.array([0, 2, 0, 1]) # Selektiere ein Element pro Zeile in neuem Array b
                                Was macht np.arange()?
print a[np.arange(4), b] # Ausgabe: "[ 1 6 7 11]"
# Verändere ein Element pro Zeile unter Nutzung von b
a[np.arange(4), b] += 10
print a # Ausgabe: "array([[11, 2, 3], # [ 4, 5, 16], # [17, 8, 9],
# [10, 21, 12]])"
```

Numpy Array Indexing

```
# Ein Beispiel für Boolean Indexing
a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 2) # Findet ein Element größer als 2;

# Erzeugt ein Numpy Array mit Booleans derselben Größe wie a, wo
# jeder Eintrag bedeutet, ob das Element größer als 2 ist.
print bool_idx
# Ausgabe: "[[False False]
#           [ True  True]
#           [ True  True]]"

# Man kann sich auf Basis von bool_idx ein Rang 1 Array
# erzeugen, welches alle Werte zurückgibt, die zutreffen
print a[bool_idx] # Ausgabe: "[3 4 5 6]"

# verkürzte Darstellung
print a[a > 2] # Ausgabe: "[3 4 5 6]"
```

Mathematische Funktionen auf Arrays

Mathematische Funktionen auf Arrays

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)
# Elementweise Summe [[ 6.0 8.0] [10.0 12.0]]
print x + y
print np.add(x, y)
# Elementweise Subtraktion [[-4.0 -4.0] [-4.0 -4.0]]
print x - y
print np.subtract(x, y)
# Elementweise Multiplikation [[ 5.0 12.0] [21.0 32.0]]
print x * y
print np.multiply(x, y)
# Elementweise Division [[ 0.2 0.33333333] [ 0.42857143 0.5 ]]
print x / y
print np.divide(x, y)
# Elementweises Wurzelziehen [[ 1. 1.41421356] [ 1.73205081 2. ]]
print np.sqrt(x)

# ...
```

Mathematische Funktionen auf Arrays

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
v = np.array([9,10])
w = np.array([11, 12])
# Skalarprodukt (inneres Produkt) zweier Vektoren, ergibt hier 219
print v.dot(w)
print np.dot(v, w)
# Matrix-Vektor-Multiplikation; erzeugt Rang 1 Array [29 67]
print x.dot(v)
print np.dot(x, v)
# Matrix-Matrix-Mult.; erzeugt Rang 2 Array [[19 22] [43 50]]
print x.dot(y)
print np.dot(x, y)

print np.sum(x) # Berechnet Summe aller Elemente, Ausgabe: "10"
print np.sum(x, axis=0) # Berechnet Spaltensumme; Ausgabe: "[4 6]"
print np.sum(x, axis=1) # Berechnet Zeilensumme; Ausgabe "[3 7]"
```

Mathematische Funktionen auf Arrays

```
x = np.array([[1,2], [3,4]])
print x # Ausgabe: "[[1 2] [3 4]]"
print x.T # Ausgabe: "[[1 3] [2 4]]"

# Die Transponierte eines Rang 1 Arrays macht nichts (zumindest in der Ausgabe):
v = np.array([1,2,3])
print v # Ausgabe: "[1 2 3]"
print v.T # Ausgabe: "[1 2 3]"

# ...
```

Broadcasting

Broadcasting

- Mechanismus um mit unterschiedlich geformten Arrays zu arbeiten, z.B. mit dem kleineren Array mehrfach mit dem größeren multiplizieren oder ähnliches

Wie würden wir das machen?

```
# Wir addieren Vektor v zu jeder Zeile in Matrix x und speichern  
# das Resultat in Matrix y  
  
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])  
v = np.array([1, 0, 1])  
  
# Wir erzeugen eine neue leere Matrix der Form x und addieren  
# den Vektor v zu jeder Zeile in einem Loop  
  
y = np.empty_like(x)  
for i in range(4):  
    y[i, :] = x[i, :] + v  
  
print y  
# Ausgabe [[ 2  2  4] [ 5  5  7] [ 8  8 10] [11 11 13]]
```

Das wird sehr langsam für große Loops

Broadcasting

- Idee zum **Performancegewinn**:
 - Addition von Vektor v mit jeder Zeile in Matrix x ist equivalent mit der **Erzeugung von Matrix vv als Stapel von Kopien von v**
 - Dann bedarf es nur noch einer **elementweisen Summe** von x und vv

```
# Wir addieren Vektor v zu jeder Zeile in Matrix x und speichern das Resultat in Matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])

vv = np.tile(v, (4, 1))
# Stapelt 4 Kopien von v aufeinander
print vv
# Ausgabe "[[1 0 1] [1 0 1] [1 0 1] [1 0 1]]"
y = x + vv # Addiere x and vv elementweise

print y # Ausgabe "[ 2 2 4] [ 5 5 7] [ 8 8 10] [11 11 13]]"
```

Broadcasting

- Broadcasting erlaubt die Berechnung ohne der Erzeugung der zusätzlichen Matrix:

```
# Wir addieren Vektor v zu jeder Zeile in Matrix x und speichern das Resultat in Matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])

y = x + v # Addiere v auf jede Zeile von x - mit 'broadcasting'

print y # Ausgabe "[[ 2 2 4] [ 5 5 7] [ 8 8 10] [11 11 13]]"
```

Broadcasting Beispiel: Aussenprodukt

```
v = np.array([1,2,3]) # v hat die Form (3,)  
w = np.array([4,5]) # w hat die Form (2,)  
  
# Um das Aussenprodukt zu berechnen muss v zunächst ein  
# Spaltenvektor sein (shape (3, 1)). Danach können wir gegen w  
# broadcasten und ein Array der Form (3,2) entsteht: das  
# Aussenprodukt von v und w [[ 4 5] [ 8 10] [12 15]]  
print np.reshape(v, (3, 1)) * w  
  
x = np.array([[1,2,3], [4,5,6]])  
# Addiere einen Vektor zu jeder Zeile. x hat die Form (2, 3) und v  
# hat die Form (3,) -> diese 'broadcasten' zu (2, 3), und ergeben #  
die folgende Matrix [[2 4 6] [5 7 9]]  
print x + v
```

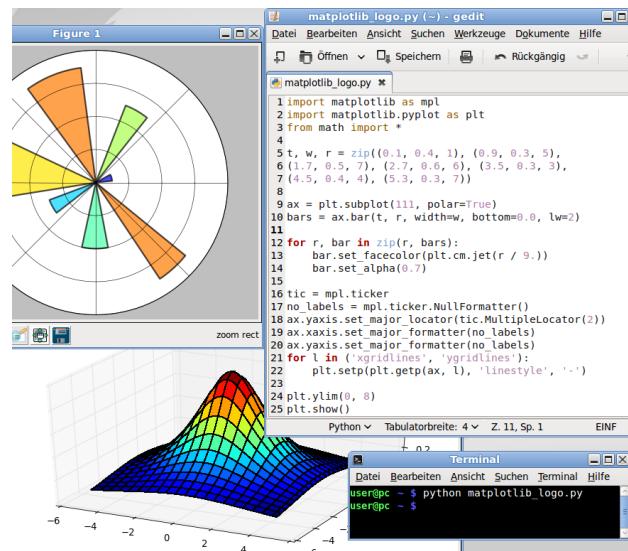
Broadcasting Beispiel: Aussenprodukt

```
# Addiere einen Vektor auf jede Spalte der Matrix x hat die Form
# (2, 3) and w die Form (2,). Wenn wir x transponieren hat Sie die
# Form (3, 2) und kann gegen w gebroadcastet werden Resultat ist
# der Form (3,2). Transponieren wir das Ergebnis kommen wir auf die
# finale Form (2,3) - Matrix x mit einem Vektor w addiert auf jede
# Spalte [[ 5 6 7] [ 9 10 11]]
print (x.T + w).T

# Eine andere Lösung ist die Umformung von w in einen Zeilenvektor
# der Form (2,1) # Diesen können wir direkt gegen x 'broadcasten' -
# mit dem gleichen Ergebnis
print x + np.reshape(w, (2, 1))

# Multiplikation einer Matrix mit einer Konstante. In numpy hat
# eine Skalar die Form (). Dieser kann zusammengeführt werden - mit
# dem Ergebnis [[ 2 4 6] [ 8 10 12]]
print x * 2
```

Mathematische Darstellung Plotting / Subplots / Images



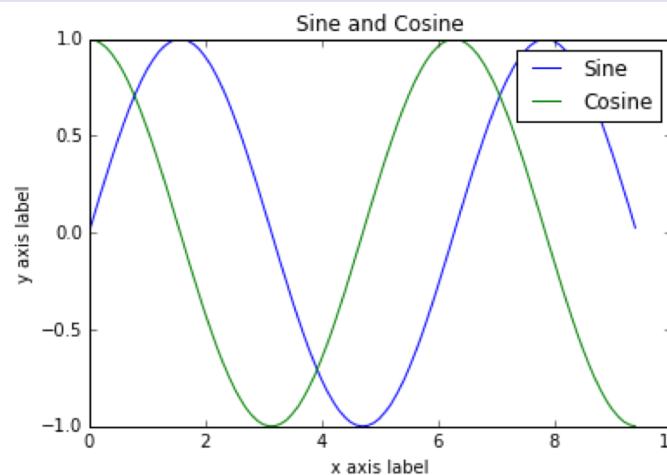
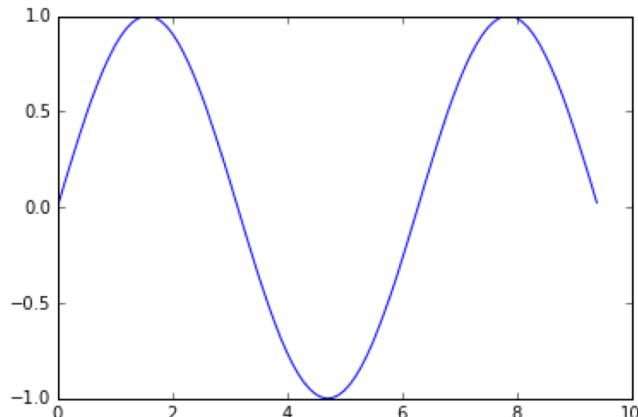
Plotting

- Matplotlib ist eine Bibliothek zur mathematischen Darstellung von Daten

```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show() # You must call plt.show() to make graphics appear.
```



Subplots

- Mehrere Darstellungen in einer Abbildung nennt man *subplots*

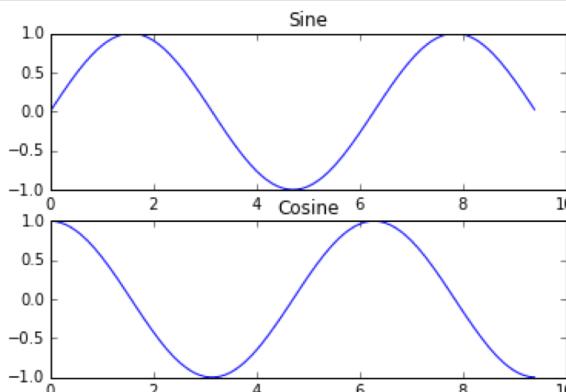
```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1, and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)

plt.subplot(2, 1, 2)
plt.plot(x, y_cos)

plt.show() # Show the figure.
```



Images

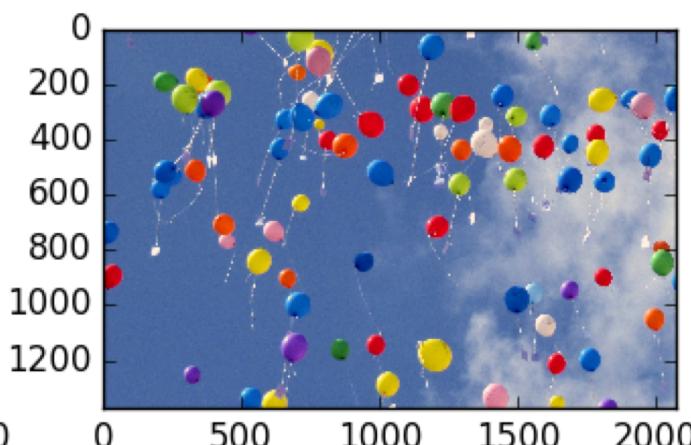
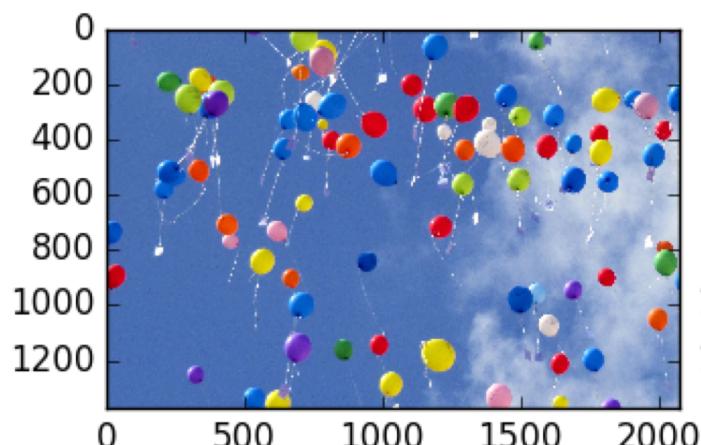
Images

- Anzeigen von Bildern möglich

```
import matplotlib
matplotlib.use('TkAgg') # MacOSX
import matplotlib.pyplot as plt

img = imread('balloons.jpg')
img_tinted = img * [1, 0.95, 0.9]
# Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)
# Show the tinted image
plt.subplot(1, 2, 2)

plt.imshow(np.uint8(img_tinted))
plt.show()
```



Dokumentationen

- **Bitte arbeiten Sie sich weiter selbstständig in die Dokumentationen ein:**
 - Numpy (<https://docs.scipy.org/doc/numpy/user/index.html>)
 - Matplotlib (<http://matplotlib.org/>)
 - Python (<https://docs.python.org/2.7/tutorial/>) oder (<https://docs.python.org/3.6/tutorial/>)