Programmiergrundlagen

Vorlesung im WS2020/2021

Prof. Dr. Jens-Matthias Bohli

Hochschule Mannheim

HOCHSCHULE AALEN
GRADUATE
CAMPUS

Python Grundlagen

Programmiergrundlagen am Beispiel Python

- Objektorientierte Skriptsprache, plattformunabhängig
- Einfach erlernbar, interaktiv, Open Source
- Eingebaute Datenstrukturen (Listen, Tupel, Dictionaries, String, ...) und mächtige Ausdrucksweise und Werkzeuge (verketten, abbilden, slicen)
- Viele Bibliotheken, z.B. Web, Email, reguläre Ausdrücke, XML/JSON/CSV, GUI, Unittesting, Bild(be|ver)arbeitung, Graphiken, Data Science, Machine Learning
- Python3 (seit 2008) hat sich durchgesetzt, wir nehmen Python 3.8

Themen

- Einführung
- Mächtige eingebaute Datentypen inklusive Listen und Dictionaries
- Dateien, Imperatives Programmieren, Unit Testing
- Funktionales Programmieren, Primitive, List Comprehension
- Module und Pakete
- Objektorientierung und Ausnahmen, Iteratoren und Generatoren
- Threading und IO-Anwendungen
- Bibliotheken, GUI, reguläre Ausdrücke
- Integration mit C/C++
- Berechnungen (numpy), Diagramme (matplotlib)

Literatur und Online-Quellen

- Python Dokumentation
  https://docs.python.org/3.8/
- Bücher (viele Online über die HS Bibliothek)
  - Learning Python, O'Reilly, Lutz, 2013
  - Python von Kopf bis Fuß, Barry, 2017
  - Einführung in Python 3, Bernd Klein, Hanser, 2017
  - Einstieg in Python, Thomas Theis, Rheinwerk, 2018
  - Learn to Program with Python 3, Irv Kalb, Apress, 2018
  - Programmierung in Python, Ralph Steyer, Springer, 2018

Erste Schritte: Unterschiede zu C

- Dynamische Typisierung: Variablen-Typen werden zur Laufzeit zugewiesen
- Garbage Collection: Speicher ungenutzter Objekte wird automatisch freigegeben
- Blöcke werden durch Einrücktiefe bestimmt, keine {}
- Referenzsemantik: Variablen sind Namen, die an Objekte gebunden sind
- Kein Semikolon nötig
- Kommentare starten mit #

Variablen, Zuweisungen und Ausdrücke

- Variablen müssen nicht deklariert werden
- Variablen müssen vor Verwendung initialisiert werden
- Wert-Gleichheit mit ==, Objektidentität mit `is`
- Zuweisungen haben Referenzsemantik: Linke Seite ist ein Name, der an das Objekt auf der rechten Seite gebunden wird.
- Ausdrücke evaluieren zu neuen Objekten

Python Grundlagen

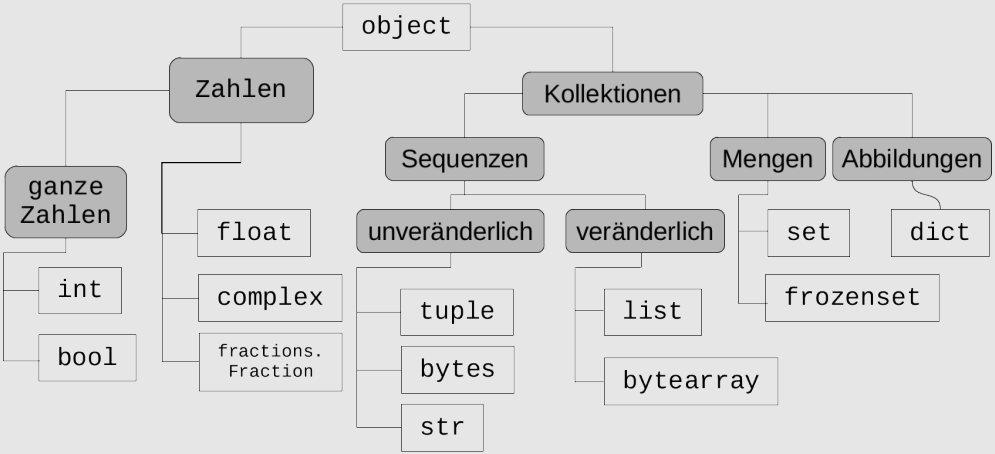
Einführung

Datentypen

Datenstrukturen

Funktionen

Datentypen


object
Zahlen
Kollektionen
ganze Zahlen
float
Sequenzen
Mengen
Abbildungen
int
complex
unveränderlich
veränderlich
set
dict
bool
fractions. Fraction
tuple
list
frozenset
bytes
bytearray
str

Zahlen und Operationen

- ganze Zahlen, `int` und `long` (beliebig lang)
- Gleitkommazahlen, `float`
- komplexe Zahlen `complex`
- Standardarithmetik: +, −, *, / (Fließkomma-Division in Python3) , ** (Exponentiation) , % , // (Integer-Division)
- Vorrangregeln und Klammern wie gewohnt
- Boolesche Werte: `False`, `True`

Ausgabe mit `print`

- Ausgabe von Objekten mit **`print`**
- Lesbare Ausgabe von Objekten
- Ausgabe aller Objekte durch Leerzeichen: **`print`**`(1,2)` getrennt
- Setzen von `end` statt Zeilenumbruch: **`print`**`(1,end="")`
- Ausgabe mit Formatstring, wie in C, mit % -Operator möglich

Strings

- Einfache oder doppelte Anführungszeichen
- Nicht limitierendes Anführungszeichen im String nutzbar
- Alternativ Escape-Zeichen verwenden Backslash \
- Typ `str`
- Mehrzeilige Strings
  - Drei Anführungszeichen am Anfang und am Ende
  - Zeilenendezeichen als `\n` verfügbar
  - Alle Anführungszeichen verwendbar
- Raw Strings um Steuerzeichen zu quotieren `r"raw \n string"`

String Eigenschaften

- Strings sind unveränderlich
  - Operation mit Strings (+ , [] , etc.) erzeugen immer neue Strings
  - Vorhandene Strings werden niemals verändert
- Strings sind Sequenzen
  - Geordnet, die Position der Elemente ist wichtig
  - Sequenz von Zeichen
- Unicode Strings seit Python3 default
- Übliche Operationen, Konkatenation mit +, Indizierung mit `[]`, Slicing
- weitere nützliche Methoden, `find`, `split`, `join`

Kommandozeilenargumente

- Modul `sys`
- Liste von Argumente `sys.argv`
- Mit dem Programm `argv.py`:

```
#!/usr/bin/python3
import sys
print(sys.argv)
```

-
```
$ python3 argv.py a b c d
['argv.py', 'a', 'b', 'c', 'd']
```

Typ `tuple`

- Sequenz-Typ für statische Felder
- Immutable: Elemente können nicht hinzugefügt oder Entfernt werden
- Elemente im Tupel können unterschiedliche Typen haben (typischerweise)
- Beispiel `a = (3,"eins",[4,5])`
- Tupel mit einem Element: `(1,)`
- Tupel sind schneller als Listen
- Tupel bieten dieselben Zugriffs-Operatoren wie Listen (aber keine Operatoren zum Modifizieren)

Typ `set`

- Datentyp für Mengen, keine Reihenfolge der Elemente, keine doppelten Elemente, mutable
- Elemente in der Menge können unterschiedliche Typen haben
- Kann nur *hashbare* Elemente enthalten (Mutable Typen sind nicht hashbar!)
- Beispiel `a = {1,2,"drei"}`

Typ `dict`

- Datentyp für Mengen von Key-Value-Paaren
- Keine Reihenfolge der Elemente, keine doppelten Elemente, mutable
- Keys müssen *hashbar* sein, Values beliebig
- Beispiel `a = {k1: v1, k2: v2, k3:  v3}`

Verzweigungen

- Schlüsselwörter: **if** , **elif** , **else**

```
if <bedingung>:
    <anweisungen>
[elif <bedigung>:
    <anweisungen>]*
[else:
    <anweisungen>]
```

<bedingung>: Ausdruck, der als Wahrheitswert interpretiert wird

<anweisungen>: Beliebige Folge von Anweisungen

- Bedingung endet mit Doppelpunkt, keine Klammern notwendig
- ACHTUNG! Einrückung ist signifikant und Teil der Syntax, <Tab> und Leerzeichen sind unterschiedliche einrückungen!
- Empfehlung: 4 Leerzeichen je Block
- `if/else` statt ternärem Operator

```
<wert> if <bedingung> else <alternativwert>
```

`while`-Schleife

- Schlüsselwörter: **`while`**, **`else`**, **`break`** , **`continue`**, **`pass`**

```
while <bedingung>:
    <anweisungen>
[else:
    <anweisungen>]
```

- **`else`**-Zweig nur, wenn Schleife ohne **`break`** verlassen wurde
- **`pass`**, leere Anweisung, macht nichts,

`for`-Schleife

- Schlüsselwörter: **`for`**, **`in`**, **`break`**, **`continue`**, **`pass`**
- Iterieren über Elemente in Sequenzen
- `for <variable> in <sequenz>:`
  `<anweisungen>`
  `[else:`
  `<anweisungen>]`
- **`in`** wählt Elemente aus Sequenz sukzessive aus und weist sie je Durchlauf der Variable zu
- Klassische Schleife über Indexe gibt es nicht
  - Nachbau mit range
    ```
    for i in range(1, 4):
    ```
  - `range(von, bis, step)`
  - Erzeugt einen Iterator über Zahlen von inklusive, bis exklusive
  - `von` optional, default 0
  - `step` optional, default 1

Hinweise zu Kontrollstrukturen

- Vergessen Sie die Doppelpunkte nicht
- Beginnen Sie immer in der ersten Spalte
- Rücken Sie konsistent ein (nur Leerzeichen, keine Tabulatoren)
- Rücken Sie immer gleich weit ein (4 Zeichen)
- Am besten mit einer passenden IDE (code , eclipse, IDLE, emacs)
- Leerzeilen beenden im interaktiven Modus einen Anweisungsblock
- Erwarten Sie nicht immer ein Ergebnis (Beispiel: append)
- Schreiben Sie nicht C/C++ in Python!

Python Grundlagen

Einführung

Datentypen

Datenstrukturen

Funktionen

list

- Sequenz-Typ, implementiert dynamische Felder
- Mutable: Elemente in der Liste können verändert werden
- Elemente in einer Liste können unterschiedliche Typen haben (typischerweise identische Typen)
- Einfache Konstruktion durch Komma-separierte Werte: `[a,b,c]`
- Beispiel `a = [3,1,4,5]`
- Index und Slicing
  - `len(a)`: Anzahl der Elemente in `a`
  - `a[i]`: Gibt i-tes Element für $0 \leq i <$verb|len(a)|
  - `a[-i]` := `a[len(a)-i]`
  - `a[i_0:i_f]` liefert eine Liste mit den Elementen an Index im halboffenen Intervall `[i_0, i_f)` (exklusiv)
  - `a[i_0:i_f:step]` Schrittweite `step` als optionaler Parameter

Python Grundlagen

Einführung

Datentypen

Datenstrukturen

Funktionen

Funktionen

- Definition mit **def**

```
def <fname>([<arg> [, <arg> ...]]):
    [<docstring>]
    <anweisungen>
    [return <name>]
```

- Erstellt Funktionsobjekt und bindet Namen <fname> daran
- **return** für Rückgabe, ansonsten automatisch None
- Eins oder mehrere Argumente/Parameter <arg> werden per Namensbindung übergeben
- Dokumentationsstring <docstring> eingebaut, optionale erste Zeile

- Interaktiv: Leerzeile zum Beenden

Parameterübergabe

Parameterübergabe ist Namensbindung

- Parameternamen werden an übergebene Objekte gebunden
- Neue lokale Namen
- Bindung von Namen an Objekt beeinflusst Objekt nicht

Verwendung

- Call by Reference
- Das Ändern eines änderbaren Objekts hat Auswirkungen (Seiteneffekte)
- Bei unveränderlichen Objekten kann es keine Auswirkungen haben, wie call by value

Vorgabewerte bei Parametern

Vorgabewerte `def fun(x=0, y=1):`

- Vorgabewert kann optional bei jedem Parameter angegeben werden
- Parameter mit Vorgabewerten hinter die Parameter ohne Vorgabewerte
- Wert für Parameter mit Vorgabewert kann weggelassen werden, dann wird Vorgabewert eingesetzt
- Schlüsselwortparameter, falls nur einige Vorgaben nicht gewählt werden sollen: `fun(y=2)`

Flexible Parameter

- `*` für Positionsparameter, „ausbreiten“ der Parameter
- `**` für Schlüsselwortparameter

```
tup = (2, 4)
fun(*tup)
dic = {'a':2, 'b':4}
plus(**dic)
```

Hinweise zu Funktionen

- Parameter und Rückgaben verwenden
- Keine globalen Variablen zur Datenübergabe verwenden
- Modifikation von Parametern vermeiden, besser neues Objekt zurückgeben
- Mehrere Werte als Tupel zurückgeben

Fortgeschrittene Konzepte

Funktionales Programmieren

Module

Objektorientiertes Programmieren

Anonyme Funktion, Lambda-Ausdruck

- Funktionen sind auch Objekte
- Erzeugen eines anonymen Funktionsobjekte mit `lambda`
- `lambda` `<args>: <ausdruck>`
- Funktion mit `<args>` als Parameter
- Wert der Funktion durch `<ausdruck>`, der `<args>` verwendet
- Zuweisung an Namen und Aufruf möglich

Funktionale Primitive `map`

- Für alle Objekte von Sequenzen eine Funktion anwenden
- Schlüsselwort `map`
- Erster Parameter die Funktion
- Ab zweiten Parameter die Sequenzen, so viele Sequenzen wie die Funktion Parameter hat
- Ergebnis ist iterierbares Objekt mit Funktions- ergebnis je Element der Eingabesequenz
- Lazy, mit `list` eine Liste daraus machen
- Längste Sequenz bestimmt Länge des Ergebnisses, Eingabe ergänzen mit `None`

Beispiel: Quadratzahlen

Aufgabe: Berechne Liste der Quadratzahlen von 1 bis 10

Beispiel: Quadratzahlen

Aufgabe: Berechne Liste der Quadratzahlen von 1 bis 10
Lösung

- Mit Schleife

```
def quadratzahlen(von=1, bis=10):
    lis = []
    for i in range(von, bis+1):
        lis.append(i**2)
    return lis
```

- Funktionale Lösung

```
def quadratzahlen(von=1, bis=10):
    return map(lambda x: x**2, range(von, bis+1))
```

Lazy, bei Bedarf eager mit `list`

Funktionale Primitive `filter`

- Für alle Objekte einer Sequenz eine Funktion anwenden und nur die durchlassen, für die der Funktionswert äquivalent zu wahr ist
- Schlüsselwort `filter`
- Erster Parameter die Funktion
- Zweiter Parameter die Sequenz
- Ergebnis ist iterierbares Objekt der ursprünglichen Elemente, deren Funktionswert äquivalent zu wahr ist
- Beispiel

```
>>> list(filter(lambda x: x>0, [1, -1, 2, -2, 3, -3, 4]))
[1, 2, 3, 4]
```

- Aufgabe: Berechne die Liste der Primzahlen zwischen 2 und 100

Funktionale Primitive `reduce`

- Kumuliere alle Objekte einer Sequenz über Funktionswerte auf
- `reduce` muss aus `functools` importiert werden
- Erster Parameter: die Funktion
- Zweiter Parameter: die Sequenz
- Dritter Parameter: der Anfangswert, meist neutral
- Ergebnis ist Auswertung der linksassoziativen sukzessiven Funktionsanwendung
- Beispiele: Summieren, Fakultät, Stringkonkatenation

List Comprehension

- Listentransformation, Anwendungsgebiet wie funktionales Programmieren
- Einfachere Verwendung der Möglichkeiten von `map` und `filter` zusammen mit `lambda` und `for`
- Syntax [<ausdruck> for <var> in <seq>]
  - <ausdruck> statt lambda -Ausdruck
  - Durch for wird <var> sukzessive ein Wert aus <seq> zugewiesen
  - Zusätzlich if <bedingung>, als Filter
  - Beliebig kombinierbar
- Beispiel

```
>>> [x**2 for x in range(1, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

- Wie mathematische Mengenschreibweise (aber Reihenfolge relevant)

$$x^2 | 1 \leq x \leq 10, x \bmod 2 = 0 = 4, 16, 36, 64, 100$$

Eager und Lazy

- Eager
  - Mit []
  - Default
- Lazy
  - Mit ()
  - Berechnet die Werte erst wenn nötig
  - Iterierbar
  - Wieder mit list auflösen
- Eager versus lazy
  - Eager erzeugt die Liste im Speicher
  - Eager iteriert über die erzeugte Liste
  - Lazy erzeugt nur was iterierbares
  - Lazy generiert immer nur das nächste zurückzugebende Element
  - Wann immer es funktioniert lazy verwenden, sonst eager
  - Lazy häufiger default in Python3

Zusammenfassung Funktionales Programmieren

Vorteile

- Verminderung der Fehleranfälligkeit
- Kompakter, ausdrucksstarker Code nahe an der Spezifikation
- Gut zu lesen und zu verstehen

Nachteile

- Syntax/Ausdrucksweise etwas gewöhnungsbedürftig
- Funktionales Denken erfordert etwas Einarbeitungszeit

Verbreitung – wieder sehr modern

- Funktionale Primitive in allen höheren Programmiersprachen zu finden (Ruby, Perl, Lisp, C++)
- Lambda-Ausdrücke sind in Java ab 8 (Stream API/Lazy), in C++ ab C++11
- MapReduce, Framework (Google) zum Suchen in riesigen Datenmengen

Referenzen: docs.python.org/3/howto/functional.html

Fortgeschrittene Konzepte

Funktionales Programmieren

Module

Objektorientiertes Programmieren

Module

Warum Module? Ziele

- Wiederverwendung von Code
- Bereitstellung von allgemein nutzbaren Diensten und Bibliotheken
- Unterteilung des Namensraums bei großen Projekten

Python Module

- Jede Python-Datei (Endung mit .py) ist ein Modul
- C-Erweiterungen sind ein Modul
- Nutzen mit `import` und `from import` Suche nach Dateien/Module
- Umgebungsvariable PYTHONPATH
- Einträge in sys.path

```
>>> import sys
>>> sys.path
['', # sucht auch im aktuellen Verzeichn
# sucht in speziellen Zips
'/usr/lib/python36.zip',
'/usr/lib/python3.6',
'/usr/lib/python3.6/lib-dynload',
'/usr/local/lib/python3.6/dist-packages',
'/usr/lib/python3/dist-packages']
```

Definieren von Modulen

- Ein Modul ist eine Datei mit der Endung .py
- Optional vorkompiliert
  - Unterordner `__pycache__`
  - `__pycache__/<modul>.<version>.pyc`
  - `<version>` zum Beispiel `cpython-36`
- In einem Verzeichnis
- Beliebiger Python-Code
- Keine Python-Schlüsselwörter als Dateiname
- Im Suchpfad erreichbar
  - Vorgabe ist aktuelles Verzeichnis und Standard-Bibliothek von Python
  - `sys.path`

Importieren von Modulen

Modul importieren

- Im aktuellen Verzeichnis
- Wechseln mit os.chdir()
- Importiert Modul mit dem Namen <modul>
- Objekte nur über den Modul-Präfix erreichbar

Namen von Modulen importieren

- Modul selbst nicht sichtbar
- Alle Namen `<name>` im aktuellen Namensraum verfügbar
- Aber an Objekt in Modul gebunden
- * für alle Namen außer Namen, die mit einem Unterstrich _ beginnen

Module und Namensräume

Ausführung beim Import

- Modulanweisungen laufen nur beim ersten Import einmal ab
- Achtung bei Interaktion, Module werden nicht automatisch neu geladen
- Reload bei interaktiver Entwicklung:
  `importlib.reload(<modul>)`

Modulattribute

- Zuweisungen und Funktionsdefinitionen auf der obersten Ebene erstellen Modulattribute
- Namen, die an Objekte gebunden sind

```
dir(<modul>)
<modul>.__dict__
```

Module sind Namensräume

- Ein Modul ist globaler Namensraum
- Auf den Namensraum eines Moduls können Sie zugreifen
- `from <modul> import *` vermeiden wegen möglicher Namenskonflikte

Hauptprogramm mit `main`

- Ziel: Ausführen eines Code-Blocks nur, wenn es als Hauptprogramm gestartet wurde, nicht wenn es importiert wurde
- Simulieren der `main`-Funktion/Methode von C/C++
- Der Name des interaktiven oder Haupt-Moduls ist `__main__`
- Test auf Name und bedingtes Ausführen

```
if __name__ == '__main__':
```

Fortgeschrittene Konzepte

Funktionales Programmieren

Module

Objektorientiertes Programmieren

Klassen in Python

- Klassen bündeln Daten (Attribute) und Funktionalität (Methoden)
- Eine Klasse ist ein *Typ* für neue Objekte. *Instanzen* von diesem Typ können angelegt werden.

Klassen erstellen

Klassendefinition:

- `class <klassenname>(object):`
  `[<docstring>]`
  `<definitionen>`
- `class`: Schlüsselwort
- `<klassenname>`: Name für die Klasse
- `object`: Wurzelklasse, Angabe optional
- `<docstring>`: Dokumentationsstring, optional
- `<definitionen>`: Funktionsdefinitionen für Methoden

Beispiel: `klasse.py`

```
class AClass(object):
    "Eine Klasse AClass"
    def __init__(self):
        self.aval = 17
    def inca(self):
        self.aval += 1
print(AClass)
```

```
$ python3 klasse.py
<class ' __main__ .AClass'>
```

Besonderheiten und Unterschiede zu C++

- Erstes Methodenargument ist immer Instanzobjekt
- Per Konvention der Name `self`
- Spezielle Methode `__init__` statt Konstruktor
- Attribute sind Klassenattribute und nicht, wie von C++ erwartet, Instanzattribute
- Instanzattribute dynamisch (`__init__`) in Namensraum self schreiben

Klassendefinition – Beispiel Stack

Stack auf Basis von Listen (Komposition)

- Initialisierung der Instanzvariable `liste` mit leerer Liste
- `self` ist immer das erste Argument bei Methoden
- `self` muss explizit aufgeführt werden bei der
- `self` ist Namensraum einer Instanz
- Weitere Argumente bei Methoden erlaubt
- Manipulation der Instanzattribute in `self` sollte nur durch Methoden erfolgen
- Keine get/set Methoden schreiben!
- Später mit Properties eingreifen

Generieren und Nutzen von Instanzen

Klassendefinition

- Klassenname ist Name gebunden an
- Im Beispiel Stack in stack.py
  ```
  from stack import Stack
  ```

Instanziierung

- Neue Instanz durch Aufruf Klassennamen, kein `new`
- Implizit wird `__init__` aufgerufen
- Zuweisung der Instanz an Namen
- Kapselung nicht erzwungen aber Konvention
- Zuweisung von Objekten erzeugt kein neues Objekt, sondern erstellt einen neuen Namen für das existierende Objekt (Referenz)

Vererbung

```
class <name>(<superkl> [, <superkl> ]*):
    ...
```

- Superklassen in Klammern dahinter
  `class FancyStack(Stack):`
- Mehrfachvererbung möglich
- Vermeide das mehrmalige Auftauchen derselben Klasse in der Vererbungshierarchie

```
from stack import Stack
class FancyStack(Stack):
    def peek(self, idx):
        return self.liste[idx]
```

Properties (statt getter und setter)

Durchgriff auf Attribute

- Ist in Python in Ordnung
- Keine Getter und Setter
- Man kann nachträglich ändern und

Kontrolle wieder erlangen: Properties

- Eigentliches Attribut verstecken, _ am Namensanfang als Hinweis
- Dann ist Durchgriff unhöflich
- Definition Getter, Setter und Löschen
- Definition des Attributs mit
  `property(getter,setter,delete,doc)`

Bibliotheken

Arrays und numerische Berechnungen: numpy

Visualisierungen: matplotlib

Datenanalyse: pandas

Arbeit mit Bildern: pillow

NumPy, Numerical Python

Ziel

- Ausdrucksmächtigkeit von Python
- Ausführungsgeschwindkeit von C/C++

Zentrale Datenstruktur: Array

- Matrix, mehrdimensionales Feld von Werten eines Typs, C/C++-Layout
- Slicing, Shaping, Extraktion, Sichten
- Schnelle mathematische Operationen: Vektorisierung, Mapping, Aggregation

Vergleich Python und Numpy

```
# with python3
def inpy3(a):
    return sum([math.sin(x) for x in a])
```

```
# with numpy
def withnp(a):
    return np.sum(np.sin(a))
```

Auswertung mit IPython Builtin

```
%timeit
>>> a = list(map(float,
             range(1234567)))
>>> print("sum=%g" % inpy3(a))
sum=1.58531
>>> %timeit inpy3(a)
118 ms ± 416  s per loop
```

```
>>> a = np.arange(1234567, dtype=floa
>>> print("sum=%g" % withnp(a))
sum=1.58531
>>> %timeit withnp(a)
14.5 ms ± 45.2  s per loop
```

- Numpy ist ca. eine Größenordnung schneller (kann mehr werden)

Numpy verwenden

- Nicht Teil der Standardbibliothek
- Externes Modul importieren

```
import numpy as np
```

- Etablierter abkürzender Name: np
- Datenstruktur array

  `np.array([1,2,3])`
  - Konstruktor
  - Aus Python Liste ein Array machen
  - Mehrdimensional

    `x = np.array([[1, 2, 3], [4, 5, 6]], np.int32)`
- Verwendung
  - Zugriff wie Liste
  - Slicing
  - Slicing macht eine Sicht, keine Kopie!
  - Operationen sind Vektoroperation

Arrays erzeugen

- Aus Listen
  - Name als Konstruktor
  - Parameter ist Liste/Tupel/Iterable
  - Mehrdimensional möglich, Eigenschaft shape
  - Achtung: gleiche Dimensionen, sonst Quatsch
- Spezial: Nullen oder Einsen
- Datentypen, Positionsparameter dtype
  - Klassische Datentypen int , float
  - NumPy diskrete Datentypen:
    `int`, `int8`, `int16`, `int32`, `int64` , `int0` (intp) ist `size_t`,
    `unsigned uint`, …
  - NumPy Fließpunkt-Datentypen:
    `float`, `float64` (double), `float32` (float)
  - Strukturierte Typen (struct) möglich
  - align -Flag (C-Compiler nachahmen)

Arrays erzeugen und formen

- `arange`: 1-dimensionales Array, wie range
- `reshape`: Sicht auf Array mit passender Dimension, keine Kopie (`a = a.reshape(...)`)
- `flatten`: Kopie in 1-dimensionales Array

Beispiele

```
>>> a = np.arange(8); a
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> b = a.reshape(2, 4); b
array([[0, 1, 2, 3],
   [4, 5, 6, 7]])
>>> a[0] = 17
>>> b
array([[17, 1, 2, 3],
[ 4, 5, 6, 7]])
>>> c = b.flatten(); c
array([17, 1, 2, 3, 4, 5, 6, 7])
>>> c[0] = 42
>>> a
array([17, 1, 2, 3, 4, 5, 6, 7])
```

Arrays elegant erzeugen

- `linspace(fro, to, n)`
  - `float`-Array mit gleichem Abstand zwischen den Werten erzeugen
  - `fro`: Anfangswert, inklusive
  - `to`: Endwert exklusive
  - `n`: Anzahl der Werte
- `logspace(fro, to, n, base=10.0)`
  - `float`-Array mit logarithmischen Abstand zwischen den Werten erzeugen
  - Werte sind Exponente, linear in den Exponenten
  - Basis kann angegeben werden

Zugriff auf Arrays

Einfacher Zugriff

- Kommasepariert in einer Klammer `[]`
- Sichten auf altes Feld
- z.B: `a[1, 2]` ein Element, `a[0:2, 1:3]` 2×2 Unterarray

Elegant

```
>>> a = np.arange(12).reshape(3, 4)
>>> a[[0,-1]]
array([[ 0, 1, 2, 3],
       [ 8, 9, 10, 11]])
>>> a[[0,-1], 0] # vector remaining
array([0, 8])
>>> a[[0,-1], [0, -1]] # matching pair
array([ 0, 11])
>>> a[[0,-1]][:,[0,-1]] # slice of res
array([[ 0, 3],
       [ 8, 11]])
>>> a[::2, ::3] # stride
array([[ 0, 3],
       [ 8, 11]])
```

Ändern von Arrays

- Slices/Strides sind Sichten auf ursprüngliche Arrays und können zum Verändern des Arrays verwendet werden
- Beispiele mit `a = np.arange(12).reshape(3, 4)`

```
>>> a[1, 2] = 17
>>> a
array([[ 0, 1, 2, 3],
       [ 4, 5, 17, 7],
       [ 8, 9, 10, 11]])
>>> a[1] = 17
>>> a
array([[ 0, 1, 2, 3],
       [17, 17, 17, 17],
       [ 8, 9, 10, 11]])
>>> a[::2, ::3] = 17
```

Operationen auf Arrays

- Komponentenweise
  - Üblicherweise sind alle Operationen komponentenweise, z.B: `+`, `*`
  - Operationen klappen bei gleicher Dimension
  - Wenn eine Dimension 1 ist, wird es bei Fehlendem wiederholt
- Skalare Operationen, Vektoroperationen
  - Für Multiplikation und Power klar, `a*2`, `a**2`
  - Geht auch unüblich als Argument: `2**a`
- Spezialoperationen
  - Dedizierte Funktionsnamen
  - Beispiel: Skalarprodukt (`dot` product)
  - Beispiel: Summe (`sum`), Summe über Achsen durch Angabe der Reduktionsachse (Dimension um 1 erniedrigt), z.B. `sum(.., axis=0)`

Viele eingebaute Funktionen

Mehr in der Doku: *https://numpy.org/doc/stable/*

- Vergleiche: `<` , `<=`, `==`, `!=`, `>=`, `>`
- Arithmetik: `+`, `-`, `*`, `/`, `reciprocal`, `square`
- Exponential: `exp`, `exp2`, `expm1` , `log`, `log10`, `log2`, `power`, `sqrt`
- Trigonometrisch: `sin`, `cos`, `tan`, `arc`, ...
- Hyperbolisch: `sinh`, `cosh`, `tanh`, `arc`, ...
- Bits: `&`, `|`, `~`, `^`, `left_shift`, `right_shift`
- Logisch: `logical_and`, `logical_or`, `logical_not`, ...
- Prädikate: `isfinite`, `isinf`, `isnan`, `signbit`
- Sonst: `abs`, `ceil`, `floor`, `mod`, `modf` , `round`, `sinc`, `sign`, `trunc`

Bibliotheken

Arrays und numerische Berechnungen: numpy

Visualisierungen: matplotlib

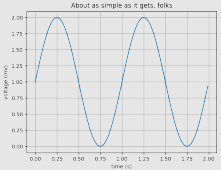
Datenanalyse: pandas

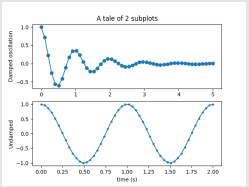
Arbeit mit Bildern: pillow

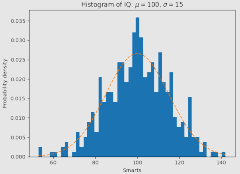
Mit `matplotlib` Daten visualisieren

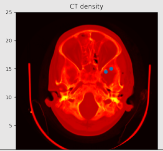
Ziel Datenvisualisierung

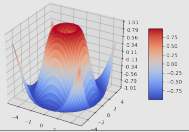
- Professionale 2D-Graphiken skripten mit Python und numpy
- Buchdruckqualität, viele Backends: PDF, PS, SVG, PNG, Tkinter, ...
- Beispiele (*matplotlib.org/tutorials/introductory/sample_plots.html*)











`matplotlib` verwenden

- Nicht Teil der Standardbibliothelk
- Externes Modul importieren

```
#!/usr/bin/python3
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 4*np.pi, 1000)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```

- Etablierter Namen für die reine Plotting-Funktionalität: `plt`
- Wird meist zusammen mit numpy verwendet

Im Beispiel:

Interaktion mit Plot

- Settings, Slider klicken
  - 6 einfache Einstellungen
  - Außendimendionen des Bildes
- Speichern als
  - PNG (vermeiden, pixelig!)
  - PDF, PS, EPS, SVG (gut)
  - PGF (Latex)
  - RAW, TIFF (vermeiden, pixelig, unkomprimiert)
- Sonstiges
  - Achsen verschieben (pan)
  - Zoom

Grafik mit Skript automatisch erstellen

- Erstellen, Speichern, auch mehr als eine
- bbox tight, weniger Rand, Verwendung in Druck
- Mehrere Plots möglich

```
plt.savefig("pltsimple_save.pdf", bbox_inches='tight')
```

Plots konfigurieren – Beispiel

- Mehrere Aufrufe für mehrere Plots im Bild
- Separat konfiguriert
- Zum Beispiel Linienbreite, Anzeige als Punkte
- Titel, Beschriftung Achsen (Latex Syntax möglich)
- Ein Grid im Hintergrund

```
x = np.linspace(0, 4*np.pi, 1000)
y1 = np.sin(x)
y2 = np.sin(.5*x)*.8
y3 = np.sin(.3*x[::42])*.4
plt.plot(x, y1)
plt.plot(x, y2, linewidth=2)
plt.plot(x[::42], y3, '.')
plt.title("Drei Sinus-Funktionen")
plt.xlabel("t in Sekunden")
plt.ylabel("sin(t), 0.8 $\cdot$
       sin(t), dotted")
plt.grid(True)
```

Dokumentation: *https://matplotlib.org/api/pyplot_summary.html*

Objektorientierte Syntax

- Statt zustandsbehaftetem Matlab-Style können Objekte verwendet werden.
- `fig, ax = plt.subplots()` erstellt ein neues Figure-Objekt mit einem einzelnen Plot.
- Anschließend wie im vorherigen Beispiel

```
ax.plot(x, y1)
ax.plot(x, y2, linewidth=2)
ax.plot(x[::42], y3, '.')
ax.set(xlabel="t in Sekunden", ylabel="sin(t),
       0.8 $\cdot$ sin(t), dotted", title="Drei Sinus-Funktionen")
ax.grid()
```

Mehrere Teilplots

```
x = np.linspace(0, 1*np.pi, 1000)
unged = np.cos(2*np.pi*x)
ged = np.cos(2*np.pi*x)*np.exp(-.5*x)
fig, axs = plt.subplots(nrows=2, ncols=1, sharex=True)

axs[0].plot(x, unged)
axs[0].set(ylabel="Ungedämpfte Schwingung")
axs[1].plot(x, ged)
axs[1].set(xlabel="t",
       ylabel="Gedämpfte Schwingung")
axs[0].set(title="Zwei Subplots")
axs[0].grid()
```

- Teilplots durch Angabe der Anzahl je Achse
- Rückgabe der Rahmen-Figure und Liste der Teilplots
- Unabhängig, außer explizit gesharter X-Achse
- Grid nur oben

Weitere Plots

- Balkendiagramm
- Histogramm
- Scatter-Plot

Bibliotheken

Arrays und numerische Berechnungen: numpy

Visualisierungen: matplotlib

Datenanalyse: pandas

Arbeit mit Bildern: pillow

Datenanalyse mit `pandas`

- Bibliothek zur Datenanalyse und Vorbereitung von Daten für maschinelles Lernen
  - I/O zu Dateien und Datenbanken
  - Behandlung fehlender Daten
  - Slicing, indexing, reshaping, neue Zeilen berechnen
  - Mächtige Werkzeuge zum Aggregieren, Gruppieren und Transformieren von Daten
  - Zusammenführen von Daten
  - Zeitreihen
- wesentliche Objekte:
  - 1-dimensionale `Series`
  - 2-dimensionaler `DataFrame`
- Baut auf Numpy auf, integriert matplotlib
- Weiterverarbeitung z.B. mit `scikit-learn`

Series

- 1-dimensionales Array
- Kann unterschiedliche Typen enthalten, aber typischerweise nicht
- Fehlende Werte werden mit `NaN` repräsentiert.
- Jedes Element kann ein Label haben, Index genannt (ähnlich `dict`, aber mit Reihenfolge)
  - Series kann als Argument für die meisten NumPy Funktionen benutzt werden, NaN Werte werden ignoriert
  - Auf Werte der Series kann durch den Index-Label zugegriffen werden
  - Operationen mit mehreren Series mit nicht-identischen Label ist möglich: dabei wird auf der Vereinigung der Index-Label operiert, fehlende Werte liefern NaN.

Konstruktoren

- `pd.Series(ndarray, index=None)`: Series aus einem Numpy-Array
  - `ndarray` muss 1-dimensional sein
  - `index` enthält die Label und muss dieselbe Länge wie `ndarray` haben.
  - Falls `index` nicht gegeben ist, sind die Label `0`, …, `len(ndarray)`
- `pd.Series(dict, index=None)` Series aus einem Dictionary
  - Label stammen aus dem Dictionary
  - `index` bestimmt die Reihenfolge und muss alle Label aus `dict` enthalten
  - Zusätzliche Label in `index` bekommen den Wert `NaN`
  - Reihenfolge ohne `index`: Einfügereigenfolge in `dict`
- `pd.Series(scalar, index)`: Wiederholter Wert `value`, Index muss angegeben werden

Indexing und Slicing

- Indizes müssen hashable sein
- Index-Labels müssen nicht eindeutig sein, aber einige Funktionen erfordern Eindeutigkeit
- Series-objekte können entweder durch ihre Indexkennungen oder durch die ihnen zugrunde liegenden Reihenfolge-Index indiziert werden

DataFrame

- 2-dimensionale Struktur
- Unterschiedliche Typen möglich und zwischen Spalten häufig
- Intuition: Spreadsheet oder SQL Tabelle
  - Jede Zeile ist ein Datensatz
  - Jede Spalte ist ein Attribut (=Series)

Visualisierung

Pandas integriert die Plotting-Funktionalität von `matplotlib`

`[df/s].plot()`: plottet DataFrame oder Series

- Für DataFrame werden alle Reihen mit entsprechenden Labeln geplottet
- `kind` ist `'bar'`, `'hist'`, `'box'`, `'density'`, `'area'`, `'scatter'`, `'hexbin'` oder `'pie'`

Bibliotheken

Arrays und numerische Berechnungen: numpy

Visualisierungen: matplotlib

Datenanalyse: pandas

Arbeit mit Bildern: pillow

Arbeiten mit Bildern

Ziele

- Anwenden von Transformationen a
- Verbessern der Bilder
- Verändern von Größe und Form
- Extrahieren von Bildinformation
- Vorbereiten der Bilder für Computervision/AI

Die Klasse `Image`

- Definiert in PIL: `from PIL import Image`
- Kann aus einer Datei, aus einem anderen Bild, oder neu erstellt werden.
- Attribute
  - `im.format`: Typ (jpeg, ppm, png, ...) falls aus Datei gelesen
  - `im.size`: Tupel mit `(width, height)`
  - `im.mode`: Farbschema (RGB, L für Graustufen, HSV, ...)
- `im.show()`: Bild anzeigen

Bilder lesen und schreiben

- `Image.open(filename)`: Bild aus Datei einlesen
  - Format ergibt sich aus dem Dateiinhalt
  - Liest nur die Eigenschaften zum Ladezeitpunkt, Daten/Pixel werden geladen, wenn sie benötigt werden. (z.B. nützlich, wenn nur eine Statistik über Bildgrößen erstellt werden soll.)
- `Image.save(filename, format=None)`: Speichert ein Bild in eine Datei
  - Format wird aus dem Filenamen ermittelt, falls nicht angegeben.

Crop und Paste

- `im.crop(box)`: schneidet eine Box aus dem Bild aus
  - `box` ist ein Tupel `(left, top, right, bottom)`
  - `(top, left)` eines Bildes hat die Koordinaten `(0, 0)`
  - `right, bottom` werden nicht mit einbezogen.
- `im.paste(region, box)`: Ersetze `box` im Bild `im` mit `region`
  - Die Größen von `box` und `region` müssen übereinstimmen
  - `box` muss innerhalb von `im` liegen.

Split und Merge von Kanälen

- `im.convert(mode)`: Konvertiert ein Bild
  - Konvertiert von oder nach `'L'` oder `'RGB'`
  - Andere konvertierung über `'RGB'` als Zwischenschritt
- `im.split()`: gibt jeden Kanal als separates Bild zurück, z.B. um die die R, G, B Kanäle eines RGB-Bildes separat zu bearbeiten
- `Image.merge(mode, bands)`: Kombiniere Kanäle zu einem Bild

Geometrische Transformationen

- `im.resize(newsize)`: Gibt Bild mit neuer Größe zurück
- `im.rotate(angle)`: Rotiert ein Bild gegen den Uhrzeigersinn
- `im.transpose(type)`: Flip oder in 90-grad Schritten rotieren (`type` ist `Image.FLIP_LEFT_RIGHT`, `Image.FLIP_TOP_BOTTOM`, oder `Image.ROTATE_90/180/270`)
- `im.transform(size, method)`: Anwenden anderer Transformationen.
- Diese Methoden werden häufig verwendet um Bilder für Deep Learning vorzubereiten!

Bilder verbessern

- `from PIL import ImageFilter, ImageEnhance`
- `im.filter(filtertype)`: Wende Filter an
  - `filtertype` definiert in defined in `ImageFilter`
  - `BLUR, DETAIL, CONTOUR, EDGE_ENHANCE, EDGE_ENHANCE_MORE, EMBOSS, FIND_EDGE`
- `im.point(operation)` Anwenden eine Operation auf jeden Pixel
- `enh=ImageEnhance.Color/Contrast/Brightness/Sharpness(im)`: Erstellt einen Enhancement Operator für `im`

PIL und numpy

- `np.array(im)`
  - Erstelle ein NumPy array aus Bilddaten
  - Sehr häufig verwendet für Maschinelles Lernen auf Bildern
- `Image.fromarray(im)`
  - Erstellt ein Bild aus einem Array
  - Der Typ des Arrays muss `unit8` sein