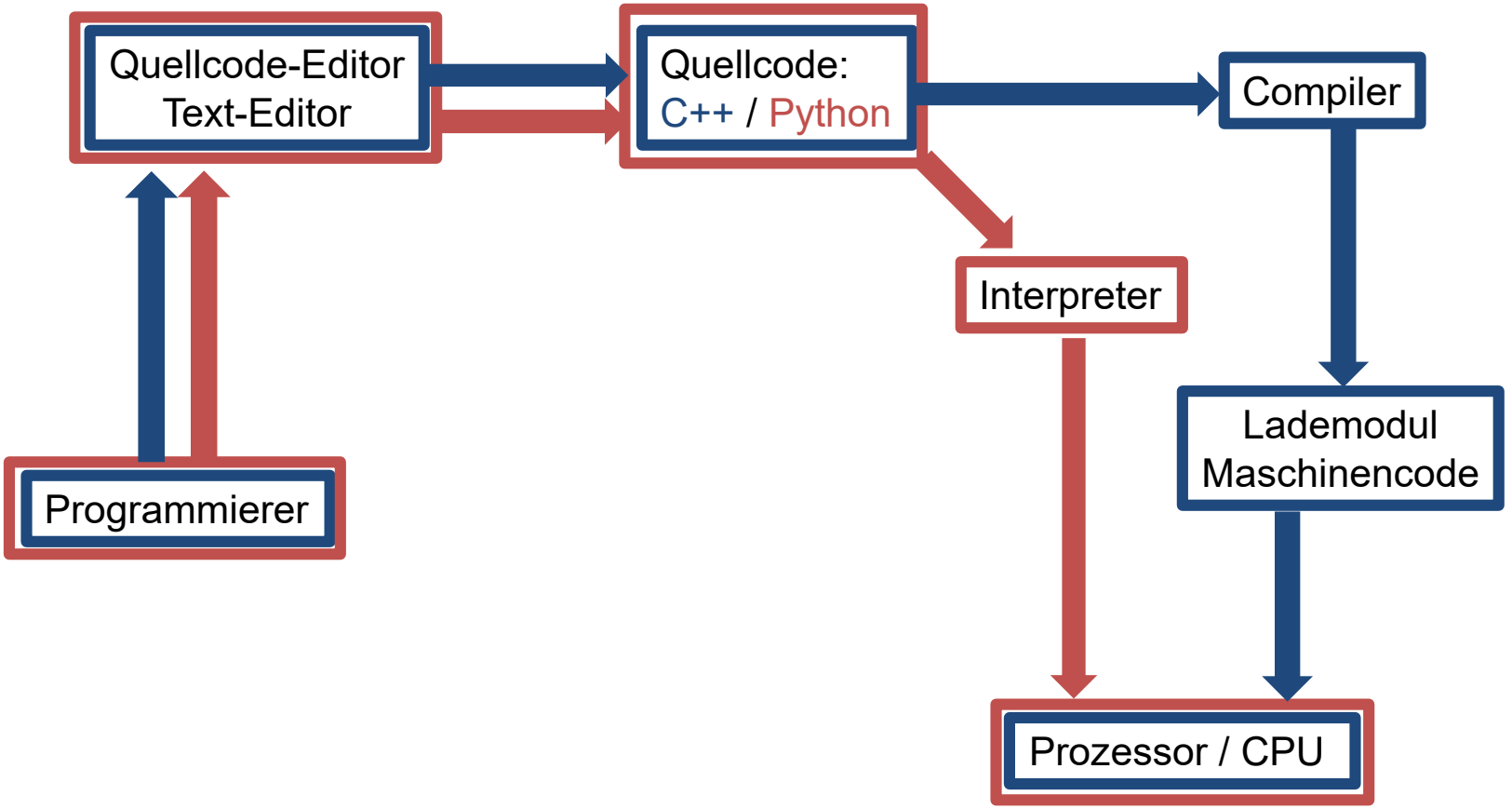
Einführung in das wissenschaftliche Arbeiten

Einheit VII:
Programmiersprachen - Teil 2: Python

Dr. Björn Eichmann (eiche@tp4.rub.de)

Bochum, 03. April 2019

Programmierpfad (C++/Python)



Eine (ganz kurze) Einführung in Python

Hilfreiche Literatur & Web Tutorials

- Allen B. Downey: *Programmieren lernen mit Python*. O'Reilly.
- Bernd Klein: *Einführung in Python 3. Für Ein- und Umsteiger*. Hanser.
- Thomas Theis: *Einstieg in Python. Ideal für Programmieranfänger geeignet*. Galileo Press
- Mark Lutz: *Learning Python*. 5. Auflage. O'Reilly.
- Hans Petter Langtangen: *A Primer on Scientific Programming with Python.* Springer.
- https://py-tutorial-de.readthedocs.io/de/python-3.3/
- https://www.python-kurs.eu/kurs.php
- https://docs.python.org/3/tutorial/
- https://www.programiz.com/python-programming/tutorial

Zur Geschichte

- Anfang der 1990er von Guido van Rossum (NL) als Nachfolger von *ABC* entwickelt, und ursprünglich für das verteilte Betriebssystem Amoeba gedacht.
- 2000: Python2.0 mit einer voll funktionsfähigen automatischen Speicherbereinigung und die Unterstützung für den Unicode-Zeichensatz.
- 2008: Python3.0 mit einigen tiefgreifenden Änderungen an der Sprache, wie das Entfernen von Redundanzen bei Befehlssätzen und veralteten Konstrukten (inkompatibel zu früheren Versionen).
- Ab 2020: Python 2 wird nicht mehr unterstützt.

Interpreter

- Python ist (üblicherweise) eine **Interpretersprache**
- Der Interpreter ist unter Linux häufig bereits installiert unter usr/local/bin und ansonsten frei verfügbar (für Linux, Mac OSX, Windows) unter https://www.python.org/downloads/
- Alternativ: *Integrated Development Environment (**IDE**),* z.B.:
  - *Jupyter Notebook* (https://jupyter.org/) → **Empfehlung!**

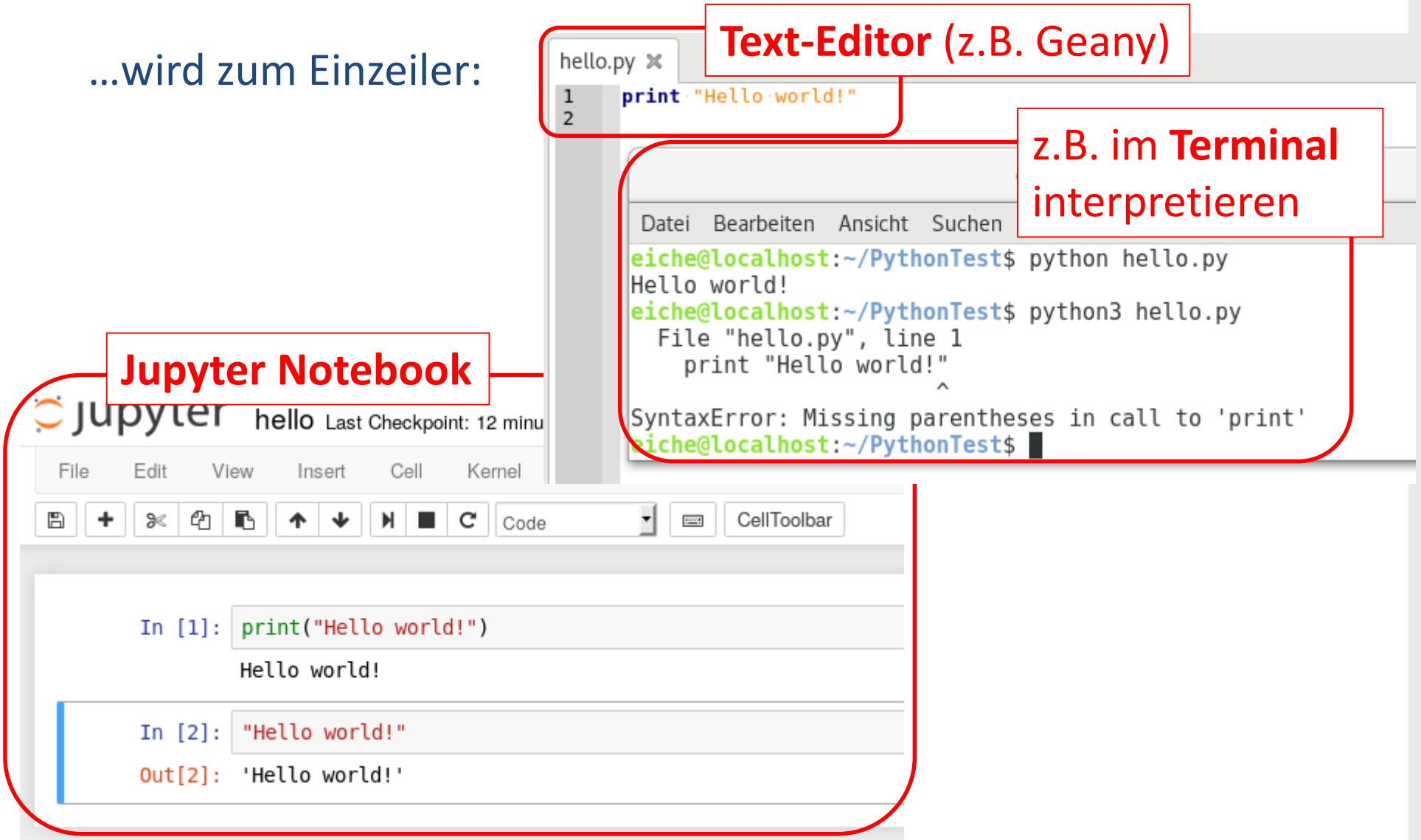
    Für Windows (Anaconda Distribution notwendig: https://www.anaconda.com/), Linux, Mac OSX; open-source, kostenlos, übersichtlich, interaktiver Output (z.B. LaTeX), unterstützt andere Sprachen wie R, Julia, und Scala
  - *KDevelop* (https://www.kdevelop.org/)

    Für Windows, Linux, Mac OSX; open-source, kostenlos
  - *Geany* (https://www.geany.org/)

    Für Windows, Linux, Mac OSX; kostenlos, übersichtlich (umfangarm)

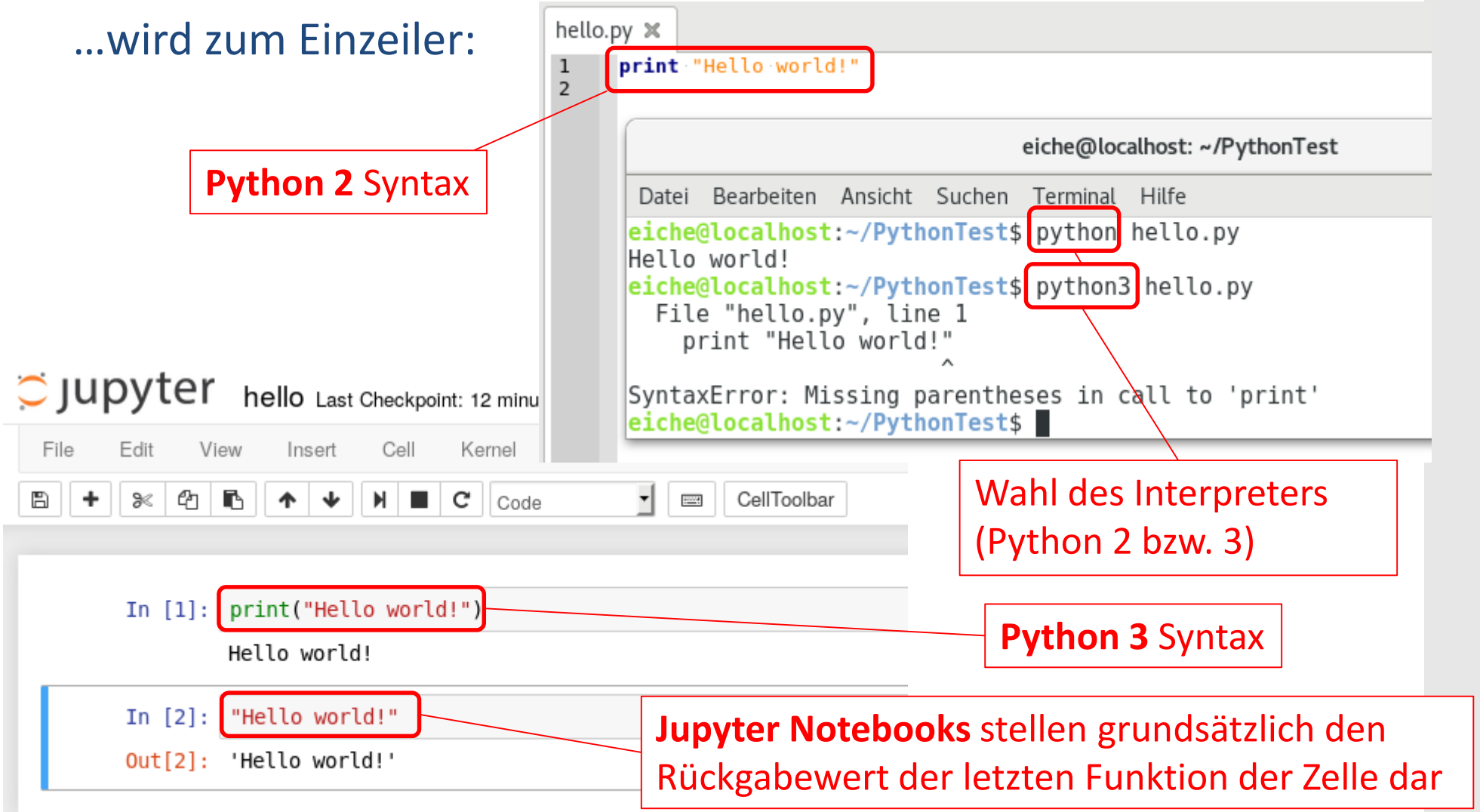
Das „Hello World“ - Programm

…wird zum Einzeiler:



Das „Hello World“ - Programm

…wird zum Einzeiler:



Einfachheit und Übersichtlichkeit

- relativ **wenig Schlüsselworte**
- auf **Übersichtlichkeit** reduzierte/ optimierte Syntax, z.B. **kein Deklarieren** von Variablen notwendig, dynamische Typanpassung
  - *Gefahr:* unsaubere Programmierung
- Vielzahl an nützlichen Bibliotheken (***Module***), z.B.
  - *numpy* (numerische Rechnungen auf *numpy-Arrays*)
  - *scipy* (spezielle Funktionen, numerische Integration, statistische Tests)
  - *matplotlib* (Plots jeglicher Art)
  - *pandas* (Datenverarbeitung bei großen Datenmengen)
  - ➢ *Grundsatz:* Ihr habt ein Problem – jemand anders hat vielleicht schon eine (effiziente) Lösung dazu entwickelt.

Basisklassen

Einige der wichtigsten Basisklassen in Python:

- Bool (bool): True oder False
- Integer (int), z.B. a = 42
- Float (float), z.B. a = 3.142
- String (str), z.B. a = 'Hello'
- Liste (list), z.B. a = ['A', 2., [2, 3, 4], 'Hello']
- Tuple (tuple), z.B. a = (1, 2, 3)
- Dictionary (dict), z.B. a = {'A':2., 'Hello':'World', (1, 2):(3., 4.)}
- NumpyArray (numpy.ndarray), z.B. a = numpy.array([1, 2, 3])
  (Hinweis: Erfordert das Einbinden des Moduls *numpy*)

Hinweis: mit typ(Variable) lässt der Variablentyp ausgegeben

Basisklassen

Einige der wichtigsten Basisklassen in Python:

- Bool (bool): True oder False
- Integer (int), z.B. a = 42
- Float (float), z.B. a = 3.142
- String (str), z.B. a = 'Hello'
- Liste (list), z.B. a = ['A', 2., [2, 3, 4], 'Hello']
- Tuple (tuple), z.B. a = (1, 2, 3)
- Dictionary (dict), z.B. a = {'A':2., 'Hello':'World', (1, 2):(3., 4.)}
- NumpyArray (numpy.ndarray), z.B. a = numpy.array([1, 2, 3])

  (Hinweis: Erfordert das Einbinden des Moduls *numpy*)

Merke:

- Variablen müssen nicht deklariert werden
- Variablenbezeichner unterliegt denselben Regeln wie bei C++

Hinweis: mit typ(Variable) lässt der Variablentyp ausgegeben

Operatoren

Operator	Beschreibung	Beispiel(e)
+, -, *	Addition, Subtraktion, Multiplikation	10+3, 4.5-2, 3.3*1.16
%	Rest	27%7 (Ergebnis: 6)
/	Division (*Achtung*: Es wird nicht mehr eine Integer-Zahl (wie noch in Python 2) sondern eine Float-Zahl (in Python 3) als Ergebnis geliefert.	10/3 (Python 3 Ergebnis: 3.333…; Python 2 Ergebnis: 3)
//	Ganzzahldivision (schneller)	10//3 (Ergebnis: 3); 10.0//3.0 (Ergebnis: 3.0)
******	Exponentiation	10**3 (Ergebnis: 1000)

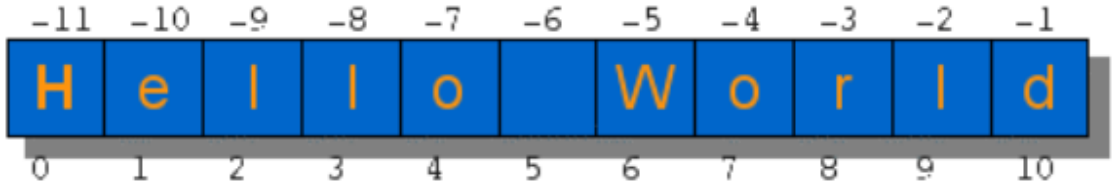
Operatoren

Operator	Beschreibung	Beispiel(e)
or, and, not	Boolsches ODER, UND, NICHT	(a or b) not c
in	„Element von“	3 in [3, 2, 1] (Ergebnis: True)
<, <=, >, >=, !=, ==	„Üblichen“ (siehe C++) Vergleichsoperatoren	10 <= 3 (Ergebnis: False); 4.5 != 4 (Ergebnis: True)

Hinweis: Es gibt zudem noch einige *bitweise Operatoren* (**~, &, |, ^, <<, >>**), auf die hier aber nicht weiter eingegangen werden soll! Siehe: https://wiki.python.org/moin/BitwiseOperators

Sequentielle Datentypen

- *Sequentielle Datentypen* beinhalten eine ***Folge an*** *gleichartigen oder verschiedenen* ***Elementen*** mit einer ***definierten Reihenfolge***, auf die man mittels ***Indizes*** zugreifen kann
  - Zeichenketten (Strings)
  - Listen
  - Tupel
  - NumpyArrays (bei Verwendung von numpy)
- *Indizierung* am Beispiel „Hello World“:




```
txt = 'Hello World'
print('Der 5. Buchstabe: ',txt[4])
print('Der 3.letzte Buchstabe: ',txt[-3])
print('Alle Buchstaben von der 2. bis zur 8. Stelle: ',txt[1:7])
```

```
Der 5. Buchstabe:  o
Der 3.letzte Buchstabe:  r
Alle Buchstaben von der 2. bis zur 8. Stelle:  ello W
```

Liste oder Tupel?

- ***Listen*** werden von eckigen Klammern umgeben und ihre Elemente mit Kommas getrennt. Sei **s** eine *Liste*, so lässt sich diese wie folgt verändern:
  - **s.append(x)**: Hängt das Element **x** der Liste an (Alternativ: **s + [x]**).
  - **s.extend(X)**: Hängt die Elemente aus **X** der Liste an.
  - **s.insert(i, x)**: Fügt das Element **x** der Liste an der Stelle **i** ein.
  - **s.pop(i)**: Gibt das **i**‘te Element **x** zurück und entfernt es aus der Liste.
  - **s.remove(x)**: Entfernt das (erste) Element **x** (einmal) aus der Liste.
- Im Gegensatz zu Listen sind ***Tuple*** unveränderlich (**immutable**), mit folgenden Eigenschaften:
  - Bessere Performance.
  - Als *Schlüssel* in *Dictionaries* geeignet.
  - 2 Methoden: **t.count(x)**: Anzahl der Elemente im Tupel **t** die **x** gleichen.
    t.index(x): Index des ersten Elements im Tupel **t** das **x** gleicht.

Assoziatives Feld (*Dictionary*)

- Ein ***Dictionary*** besteht aus ***Schlüssel-Objekt-Paaren***
  - *Schlüssel* müssen unveränderlich (**immutable**) sein.
  - *Objekt* kann **beliebig** sein.
- *Dictionary* am Beispiel eines Deutsch-Englisch Wörterbuchs:

```
en_dt = {"red":"rot", "blue":"blau"}
print('en_dt:', en_dt)
print('Übersetzung von blue:', en_dt['blue'])
en_dt["green"] = 'grün' # fügt das Paar 'green':'grün' hinzu
print('Schlüssel:',en_dt.keys())
print('Objekte:',en_dt.values())
print('Werte-Paare:',en_dt.items())
en_dt2 = {'cheese':'Käse', 'ham':'Schinken' , 'red':'Rot'}
en_dt.update(en_dt2)  # hängt en_dt2 an en_dt an
print('en_dt:', en_dt)
```

```
en_dt: {'blue': 'blau', 'red': 'rot'}
Übersetzung von blue: blau
Schlüssel: dict_keys(['green', 'blue', 'red'])
Objekte: dict_values(['grün', 'blau', 'rot'])
Werte-Paare: dict_items([('green', 'grün'), ('blue', 'blau'), ('red', 'rot')])
en dt: {'green': 'grün', 'ham': 'Schinken', 'cheese': 'Käse', 'blue': 'blau', 'red': 'Rot'}
```

Kontrollstrukturen (Verzweigungen)

- **if** - Verzweigung:

```
if Ausdruck:
        Anweisung(en)
```

- Alternative **else** - Verzweigung:

```
if Ausdruck:
        Anweisung(en)1
else:
        Anweisung(en)2
```

- **else if** - Verzweigung:

```
if Ausdruck1:
        Anweisung(en)1
elif Ausdruck2:
        Anweisung(en)2
// weitere elif-Anweisungen möglich
// zusätzliche else-Anweisung auch möglich
```

Achtung: Python ist sensibel auf **Einrückungen**, d.h. alles was um einen Tab eingerückt ist, gehört zur jeweiligen Verzweigung!

Kontrollstrukturen (Schleifen)

- **while** - Anweisung:

  while Ausdruck:

  Anweisung(en)

Hinweis: Auch in Python sind die *Sprunganweisungen* **break** und **continue** definiert (gleiche Funktionsweise wie in C++)!

- **for** - Anweisung:

  for Variable **in** Sequenz:

  Anweisung(en)

Für Zählschleifen gibt es die **range()-**Funktion, welche einen Iterator, der Zahlen in einem bestimmten Bereich (range) liefert.

- **range(begin, end)** liefert alle ganzen Zahlen von begin (einschließlich) bis end (ausschließlich).
- **range(end)** ist identisch zu **range(0, end)**

ODER (falls Index und Variable benötigt wird):

for index**,** Variable **in enumerate(**Sequenz):

Anweisung(en)

Kontrollstrukturen (Ausnahmen)

- **try** – Anweisungen ermöglichen das Abfangen eines Fehlers (Fehlertyp, z.B. ValueError, AttributeError,…) im except-Block:

```
try:
    Anweisung(en)
except Fehlertyp:
    Anweisung(en)
```

- *Mehrer Fehlertypen abfangen:* **except (**Fehlertyp1, Fehlertyp2**):**
- *Alle Fehlertypen abfangen (nicht zu empfehlen!):* **except:**

Build-in Funktionen

Folgende Funktionen sind dem Python Interpreter stets bekannt:

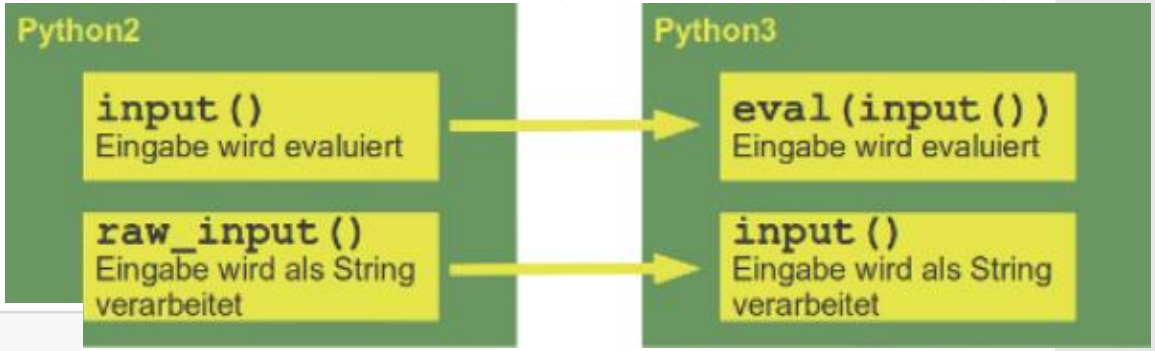
Built-in Functions				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Für weitere Details: https://docs.python.org/3/library/functions.html

input()

- Die Funktion **input(**eingabeprompt**)** ermöglicht die Eingabe von Werten über die Tastatur
  - Damit der User auch weiß, was er einzugeben hat, wird der String des Parameters "eingabeprompt" ausgegeben, sofern er existiert
  - **input()** liefert immer einen string zurück (Python 3)
  - **eval()** interpretiert die Eingabe und liefert den entsprechenden Datentyp zurück (Python 3)
  - Unterschied

    Python 2 – Python 3:
  - Ein Beispiel:




```
eingabe1 = input('Ihre 1. Eingabe ?')
print('1. Eingabe:',eingabe1, type(eingabe1))
eingabe2 = eval(input('Ihre 2. Eingabe ?'))
print('2. Eingabe:',eingabe2, type(eingabe2))
```

```
Ihre 1. Eingabe ?12.3
1. Eingabe: 12.3 <class 'str'>
Ihre 2. Eingabe ?12.3
2. Eingabe: 12.3 <class 'float'>
```

Funktionen

- Funktionen sind eines der nützlichsten Handwerkszeuge um Python Code les- und wiederverwendbar zu gestalten.
- Im Grunde funktionieren sie (analog zu C++) wie eine einfache Zuordnungsvorschrift $x \rightarrow f(x)$.
- Funktionen werden mit einem **def** eingeleitet und ein **return** bestimmt den Rückgabewert:

```
def Funktionsname ( Paramter):
    Anweisung(en)
    return Rückgabewert
```

Wie bereits bei den Kontrollstrukturen ist der Anweisungsblock durch die **Tab-Einrückungen** festgelegt!

Funktionen (Parameterübergabe)

- Die Parameterübergabe wird bei Python durch den *„Call-by-Object“* Mechanismus beschrieben. Grundsätzlich wird die Referenz an den formalen Parameter der Funktion übertragen, wobei:
  - ***Bei unveränderlichen Objekten*** kann der Inhalt des Objekts innerhalb der Funktion **nicht verändert** werden, weil es schließlich unveränderlich ist, daher wirkt es nach außen eher wie eine Wertübergabe („**call by value**“)
  - ***Bei veränderlichen Objekten*** kann der Inhalt des Objekts innerhalb der Funktion **verändert** werden, weil es schließlich veränderlich ist, daher wirkt es auch nach außen wie eine Referenzübergabe („**call by referenz**“)

Funktionen (Parameterübergabe)

- Die Parameterübergabe wird bei Python durch den ***„Call-by-Object“*** Mechanismus beschrieben. Grundsätzlich wird die Referenz an den formalen Parameter der Funktion übertragen, wobei:
  - ***Bei unveränderlichen Objekten*** kann der Inhalt des Objekts innerhalb der Funktion **nicht verändert** werden, weil es schließlich unveränderlich ist, daher wirkt es nach außen eher wie eine Wertübergabe („**call by value**“)
  - ***Bei veränderlichen Objekten*** kann der Inhalt des Objekts innerhalb der Funktion **verändert** werden, weil es schließlich veränderlich ist, daher wirkt es auch nach außen wie eine Referenzübergabe („**call by referenz**“)
- Eine Funktion f läßt sich auch definieren mit
  - einer **variablen Anzahl an Parametern** args: def f (*args)
  - einer **variablen Anzahl an Schlüsselwortparametern** kwargs: def f (**kwargs)
  - einem **beliebigen Parameter-Mix** (von allem): def f (x, y, *args, **kwargs)
- In Funktionsaufrufen wird durch * das Argument entpackt, z.B.:

  Sei def f (x, y, z), so ist für p=(1,2,3) der Aufruf f(p[0], p[1], p[2]) identisch zu f(*p)

(Flaches) Kopieren

- *Kopieren von **einfachen, unveränderlichen (immutable) Objekten*** entspricht der naiven Erwartung (→← C++ Zeiger), z.B.:

```
x = 3
y = x
y = 4
print("x:",x)
print("y:",y)
```

```
x: 3
y: 4
```

- ***Flache*, veränderliche Objekte** (Listen, Dictionaries die nicht verschachtelt sind) werden nicht komplett kopiert, es sei denn man verwendet den *Teilbereichsoperator* **[:]**. z.B.:

```
a = [1,2,3]
b = a
b[0] = 10
print("a:",a)
print("b:",b)
```

```
a: [10, 2, 3]
b: [10, 2, 3]
```

Komplett kopieren

```
a = [1,2,3]
b = a[:]
b[0] = 10
print("a:",a)
print("b:",b)
```

```
a: [1, 2, 3]
b: [10, 2, 3]
```

Tiefes Kopieren

- ***Nicht flache (tiefe)*, veränderliche Objekte** (Listen, Dictionaries die verschachtelt sind) werden auch mit dem Teilbereichsoperator [:] nicht komplett kopiert, zu diesem Zweck stellt das Modul copy die Methode deepcopy zur Verfügung z.B.:

```
A = [1,2,[10,20]]
B = A[:]
B[0] = 100
print("A:",A)
print("B:",B)
B[2][0] = 100
print("A:",A)
print("B:",B)
```

```
A: [1, 2, [10, 20]]
B: [100, 2, [10, 20]]
A: [1, 2, [100, 20]]
B: [100, 2, [100, 20]]
```


```
from copy import deepcopy
A = [1,2,[10,20]]
B = deepcopy(A)
B[0] = 100
print("A:",A)
print("B:",B)
B[2][0] = 100
print("A:",A)
print("B:",B)
```

```
A: [1, 2, [10, 20]]
B: [100, 2, [10, 20]]
A: [1, 2, [10, 20]]
B: [100, 2, [100, 20]]
```

- Manche Python Klassen besitzt eine **eingebaute Kopierfunktion**, z.B. numpy.array**.copy(),** die eine komplette Kopie des Objektes anlegt.

Klassen

- Eine *Klasse* besitzt folgende Syntax:

```
class Klassenname[(Oberklasse(n))]:
    Anweisung(en)
```

- *Eine Instanz der Klassen* erhält man durch die Zuweisung:

```
Instanzname = Klassenname()
```

- *Klassen* besitzen *Attribute* (Eigenschaften) und *Methoden*:
  - Einer *Instanz* (oder auch dem Klassenobjekt selbst) kann man mit dem **.**-Operator beliebige *Attributnamen* zuordnen.
  - *Methoden* sind Funktionen innerhalb einer Klasse, wobei der erste Parameter (genann: self) einer Methode immer eine Referenz auf die Instanz selbst ist.

Klassen

- Eine *Klasse* besitzt folgende Syntax:

  class Klassenname[(Oberklasse(n))]**:**

  Anweisung(en)

  optional: falls von Oberklasse(n) geerbt wird

- *Eine **Instanz der Klassen*** erhält man durch die Zuweisung:

  Instanzname = Klassenname()

- *Klassen* besitzen *Attribute* (Eigenschaften) *und Methoden*:
  - Einer ***Instanz*** (oder auch dem Klassenobjekt selbst) kann man mit dem **.**-Operator beliebige *Attributnamen* zuordnen.
  - ***Methoden*** sind Funktionen innerhalb einer Klasse, wobei der erste Parameter (genann: **self**) einer Methode immer eine Referenz auf die Instanz selbst ist.

Klassen

Die **__init__** *- Methode:*

- um *Attribute* direkt nach der Erzeugung einer Instanz zu definieren (**Initialisierung der Instanz**)
- Vergleichbar mit einem *Konstruktor* (wobei Python keine expliziten Konstruktoren bzw. Destruktoren kennt)
- Üblicherweise direkt unter dem Klassenheader
- Allgemeine Syntax:

```
class Klassenname:
        def __init__(self [, Parameter]):
                Anweisung(en)
```

Klassen

Public-, Protected- und Private-Attribute:

- ***Public*** (Attribute ohne führende Unterstriche sind sowohl innerhalb einer Klasse als auch von außen les- und schreibbar): name
- ***Protected*** (auf Attribute kann von von außen lesend und schreibend zugegriffen werden, aber nicht mehr nach Vererbung): _name
- ***Private*** (Attribute sind von außen nicht sichtbar und nicht benutzbar): __name

Durch *Vererbung* lässt sich eine Beziehung zwischen einer allgemeinen Klasse (*Basisklasse* oder *Oberklasse*) und einer spezialisierten Klasse (*Unterklasse*) definieren

- Zugriff von außen auf Attribute der Basisklasse ist nur möglich falls diese *public* sind.

Klassen (Beispiel)

```
In [1]: class Roboter:
            pass
```

Null Operator (nix passiert; reiner Platzhalter)

```
In [2]: Robo1 = Roboter()
        Robo1.name = "R2-D2"
        print(Robo1.name)
        Roboter.baujahr = 1977
        Robo2 = Roboter()
        print(Robo2.baujahr)
        print(Robo2.name)
```

Zuordnung des Attributs name an die *Instanz* Robo1

Die *Instanz* Robo2 kennt zwar baujahr (weil der Klasse zugewiesen), aber nicht name

```
R2-D2
1977
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-2-cd487358cc4c> in <module>()
      5 Robo2 = Roboter()
      6 print(Robo2.baujahr)
----> 7 print(Robo2.name)

AttributeError: 'Roboter' object has no attribute 'name'
```

Klassen (Beispiel)

In [5]:
```
class Roboter():
    def __init__(self, name, baujahr):
        self.name = name
        self.baujahr = baujahr
    def float2int(self, x):
        return int(x)
```

int() - Funtion liefert den Integer-Wert des Parameters

In [6]:
```
Robo3 = Roboter('R2-D2', 1977)
print(Robo3.baujahr)
print(Robo3.name)
print(Robo3.float2int(1.56))
```

```
1977
R2-D2
1
```

Initialisierung der Instanz Robo3

Zugriff auf die Attribute und Methoden der Klasse Roboter.

Klassen (Beispiel)

In [15]:

```
class Person:

    def __init__(self, vorname, nachname, geburtsdatum):
        self._vorname = vorname
        self._nachname = nachname
        self.geburtsdatum = geburtsdatum

    def __str__(self):
        ret = self._vorname + " " + self._nachname
        ret += ", " + self.geburtsdatum
        return  ret

class Angestellter(Person):

    def __init__(self, vorname, nachname, geburtsdatum, personalnummer):
        Person.__init__(self, vorname, nachname, geburtsdatum)
        # alternativ:
        #super().__init__(vorname, nachname, geburtsdatum)
        self.__personalnummer = personalnummer

    def __str__(self):
        #return super().__str__() + " " + self.__personalnummer
        return Person.__str__(self) + ", Pers.Nr.: " + str(self.__personalnummer)
```

Klasse Angestellter erbt die Attribute und Methoden von Person.

Klassen (Beispiel)

In [15]:

```
class Person:

    def __init__(self, vorname, nachname, geburtsdatum):
        self._vorname = vorname
        self._nachname = nachname
        self.geburtsdatum = geburtsdatum

    def __str__(self):
        ret = self._vorname + " " + self._nachname
        ret += ", " + self.geburtsdatum
        return ret

class Angestellter(Person):

    def __init__(self, vorname, nachname, geburtsdatum, personalnummer):
        Person.__init__(self, vorname, nachname, geburtsdatum)
        # alternativ:
        #super().__init__(vorname, nachname, geburtsdatum)
        self.__personalnummer = personalnummer

    def __str__(self):
        #return super().__str__() + " " + self.__personalnummer
        return Person.__str__(self) + ", Pers.Nr.: " + str(self.__personalnummer)
```

protected Attribut

public Attribut

Methode zur Umwandlung in einen String

Methode __str__() wird überschrieben

Klasse Angestellter erbt die Attribute und Methoden von Person.

Klassen (Beispiel)

```
In [15]: class Person:

             def __init__(self, vorname, nachname, geburtsdatum):
                 self._vorname = vorname
                 self._nachname = nachname
                 self.geburtsdatum = geburtsdatum

             def __str__(self):
                 ret = self._vorname + " " + self._nachname
                 ret += ", " + self.geburtsdatum
                 return  ret

         class Angestellter(Person):

             def __init__(self, vorname, nachname, geburtsdatum, personalnummer):
                 Person.__init__(self, vorname, nachname, geburtsdatum)
                 # alternativ:
                 #super().__init__(vorname, nachname, geburtsdatum)
                 self.__personalnummer = personalnummer

             def __str__(self):
                 #return super().__str__() + " " + self.__personalnummer
                 return Person.__str__(self) + ", Pers.Nr.: " + str(self.__personalnummer)
```

```
In [21]: A1 = Angestellter("Homer", "Simpson", "09.08.1969", 10800324567)
         print(A1.__str__())
         # A1.__personalnummer liefert Fehlermeldung, weil das Attribut private ist
         print(A1.geburtsdatum)
         print(A1.nachname)

         Homer Simpson, 09.08.1969, Pers.Nr.: 10800324567
         09.08.1969

         ---------------------------------------------------------------------------
         AttributeError                            Traceback (most recent call last)
         <ipython-input-21-2a31a7fb5540> in <module>()
               3 # A1.__personalnummer liefert Fehlermeldung, weil das Attribut private ist
               4 print(A1.geburtsdatum)
         ----> 5 print(A1.nachname)

         AttributeError: 'Angestellter' object has no attribute 'nachname'
```

Möglich, weil *public* Attribut in der Basisklasse

Nicht möglich, weil *protected* Attribut in der Basisklasse

Einbinden von Bibliotheken

Eine *Bibliothek*, egal ob aus der Standardbibliothek oder eine eigene, wird mit der **import**-Anweisung eingebunden, z.B.: import numpy

- Der Interpreter sucht beim importieren in folgender Reihenfolge:
  1. Im aktuellen Verzeichnis
  2. PYTHONPATH
  3. Falls letzteres nicht gesetzt ist, wird im Default-Pfad gesucht

Verschiedene Importmöglichkeiten eines Pakets (modul):

- import modul (Verwendung des gesamten Moduls mit modul)
- import modul as name (Verwendung des gesamten Moduls mit name)
- from modul import name (Aus modul name importieren, und unter name verwenden)
- from modul import * (importiert alles aus modul und kann *ohne* modul-Namen verwendet werden – nicht zu empfehlen, wegen Überschreibungsgefahr!)

Einbinden von Bibliotheken

Eigene Module einbinden:

- Sei die **Funktion** f(x) in der Datei eigeneFunktion.py definiert, so **importiert/ verwendet** man sie von einem anderen Programm aus:

```
import eigeneFunktion
eigeneFunktion.f(Parameterwert)
```

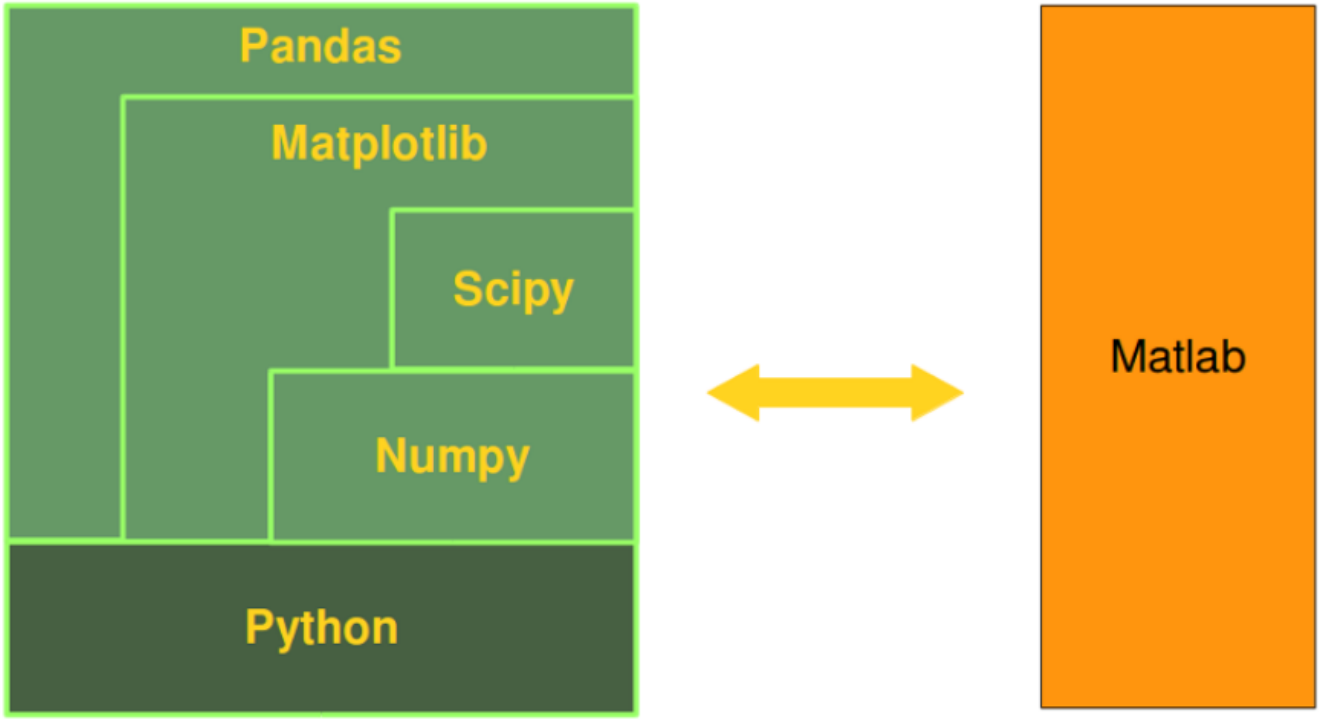
- Ebenso lässt sich auch die **Klasse** K aus der Datei eigeneKlasse.py **importieren** durch:

```
from eigeneKlasse.py import K
```

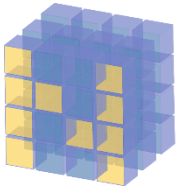
Module neu laden mit reload() – nützlich in der interaktiven Shell, falls Module zwischendurch geändert wurden.

Die Python-Alternative zu MATLAB

Python in Kombination mit *NumPy, SciPy, Matplotlib* und *Pandas* kann prinzipiell als vollwertiger Ersatz für MATLAB genutzt werden und ist zudem „open-source“!



Numerisches Rechnen



http://www.numpy.org/

- NumPy ist eine Programmbibliothek, die eine einfache Handhabung von Vektoren, Matrizen oder generell großen mehrdimensionalen *Arrays* ermöglicht.
- Bessere Performance (als der standardmäßig installierte Interpreter CPython) bei mathematischen Algorithmen
- Kernfunktionalität basiert auf der Datenstruktur ***ndarray*** *(n-dimensionales Array)*
  - unveränderlich (**immutable**)
  - homogen typisiert (alle Elemente müssen **vom selben Datentyp** sein)
  - stellt selbst schon **viele Funktionen** (sum(), mean), std(),…), die mit dem Punkt-Operator ndarray.function() ausgeführt werden, zur Verfügung.
- …und vieles mehr, siehe:

  https://docs.scipy.org/doc/numpy-1.13.0/reference/index.html

Numerisches Rechnen (NumPy-Beispiele)

```
In [1]: import numpy as np
```

üblicher *Alias* für numpy

```
In [2]: # 1-dim Array von 1...27:
        Arr_1dim = np.linspace(1,27, 27)
        print("Arr_1dim:", Arr_1dim, "\n")
        # Wir bilden nun ein 3x9 array:
        Arr_2dim = Arr_1dim.copy().reshape(3, 9)
        print("Arr_2dim:", Arr_2dim, "\n")
        # Bilden wir nun ein 3x3x3 Array:
        Arr_3dim = Arr_1dim.copy().reshape(3, 3, 3)
        print("Arr_3dim:", Arr_3dim)
```

ndarrays definieren, z.B mit linspace(begin, end, total_N) in einer Dimension, danach reshapen in mehrere Dimensionen

```
Arr_1dim: [  1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.  12.  13.  14.  15.
  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.]

Arr_2dim: [[  1.   2.   3.   4.   5.   6.   7.   8.   9.]
 [ 10.  11.  12.  13.  14.  15.  16.  17.  18.]
 [ 19.  20.  21.  22.  23.  24.  25.  26.  27.]]

Arr_3dim: [[[  1.   2.   3.]
  [  4.   5.   6.]
  [  7.   8.   9.]]

 [[ 10.  11.  12.]
  [ 13.  14.  15.]
  [ 16.  17.  18.]]

 [[ 19.  20.  21.]
  [ 22.  23.  24.]
  [ 25.  26.  27.]]]
```

Numerisches Rechnen (NumPy-Beispiele)

In [3]:
```
print("Form des Arrays Arr_1dim:", Arr_1dim.shape)
print("Letzten drei Einträge des Arrays Arr_1dim:", Arr_1dim[-3:])
print("Jeder dritte Eintrag des Arrays Arr_1dim:", Arr_1dim[::3], "\n")

print("Form des Arrays Arr_2dim:", Arr_2dim.shape)
print("1. Spaltenvektor des Arrays Arr_2dim", Arr_2dim[:,0])
print("1. Zeilenvektor des Arrays Arr_2dim", Arr_2dim[0,:], "\n")

print("Form des Arrays Arr_3dim:", Arr_3dim.shape)
# Der Eintrag mit Index (0,0,0)
print("Eintrag (0,0,0) des Arrays Arr_3dim:", Arr_3dim[0,0,0])
```

```
Form des Arrays Arr_1dim: (27,)
Letzten drei Einträge des Arrays Arr_1dim: [ 25.  26.  27.]
Jeder dritte Eintrag des Arrays Arr_1dim: [  1.   4.   7.  10.  13.  16.  19.  22.  25.]

Form des Arrays Arr_2dim: (3, 9)
1. Spaltenvektor des Arrays Arr_2dim [  1.  10.  19.]
1. Zeilenvektor des Arrays Arr_2dim [ 1.  2.  3.  4.  5.  6.  7.  8.  9.]

Form des Arrays Arr_3dim: (3, 3, 3)
Eintrag (0,0,0) des Arrays Arr_3dim: 1.0
```

ndarrays auslesen (Form/ Einträge)

Numerisches Rechnen (NumPy-Beispiele)

Mit ndarrays rechnen

In [4]:
```
print("Summe über alle Elemente von Arr_2dim: ", Arr_2dim.sum())
print("Mittelwert der Elemente von Arr_2dim: ", Arr_2dim.mean())
print("Summe über alle Elemente von Arr_2dim: ", Arr_2dim.sum(axis=0))
print("Summe über alle Elemente von Arr_2dim: ", Arr_2dim.sum(axis=1))
```

```
Summe über alle Elemente von Arr_2dim:  378.0
Mittelwert der Elemente von Arr_2dim:  14.0
Summe über alle Elemente von Arr_2dim:  [ 30.  33.  36.  39.  42.  45.  48.  51.  54.]
Summe über alle Elemente von Arr_2dim:  [  45.  126.  207.]
```

In [5]:
```
print("Kosinus(Arr_2dim*pi): ", np.cos(Arr_2dim*np.pi))
vec1 = np.array([1.,0.,0.])
vec2 = np.array([0.,1.,0.])
print("vec1:", vec1, "und vec2:", vec2)
print("Skalarprodukt von vec1 mit vec2: ", np.dot(vec1,vec2))
print("Kreuzprodukt von vec1 mit vec2: ", np.cross(vec1,vec2))
print("Natürliche Logarithmus der Elemente von vec1+10: ", np.log(vec1+10))
print("Zehner-Logarithmus der Elemente von vec1+10: ", np.log10(vec1+10))
```

```
Kosinus(Arr_2dim*pi):  [[-1.  1. -1.  1. -1.  1. -1.  1. -1.]
 [ 1. -1.  1. -1.  1. -1.  1. -1.  1.]
 [-1.  1. -1.  1. -1.  1. -1.  1. -1.]]
vec1: [ 1.  0.  0.] und vec2: [ 0.  1.  0.]
Skalarprodukt von vec1 mit vec2:  0.0
Kreuzprodukt von vec1 mit vec2:  [ 0.  0.  1.]
Natürliche Logarithmus der Elemente von vec1+10:  [ 2.39789527  2.30258509  2.30258509]
Zehner-Logarithmus der Elemente von vec1+10:  [ 1.04139269  1.          1.        ]
```

Numerisches Rechnen (Input & Output)

NumPy stellt auch diverse Routinen zum Einladen und Rausschreiben von Daten zur Verfügung:

NumPy binary files (NPY, NPZ)

load(file[, mmap_mode, allow_pickle, ...])	Load arrays or pickled objects from `.npy`, `.npz` or pickled files.
save(file, arr[, allow_pickle, fix_imports])	Save an array to a binary file in NumPy `.npy` format.
savez(file, *args, **kwds)	Save several arrays into a single file in uncompressed `.npz` format.
savez_compressed(file, *args, **kwds)	Save several arrays into a single file in compressed `.npz` format.

The format of these binary file types is documented in **numpy.lib.format**

Text files

loadtxt(fname[, dtype, comments, delimiter, ...])	Load data from a text file.
savetxt(fname, X[, fmt, delimiter, newline, ...])	Save an array to a text file.
genfromtxt(fname[, dtype, comments, ...])	Load data from a text file, with missing values handled as specified.
fromregex(file, regexp, dtype[, encoding])	Construct an array from a text file, using regular expression parsing.
fromstring(string[, dtype, count, sep])	A new 1-D array initialized from text data in a string.
ndarray.tofile(fid[, sep, format])	Write array to a file as text or binary (default).
ndarray.tolist()	Return the array as a (possibly nested) list.

für mehr Details: https://docs.scipy.org/doc/numpy-1.15.1/reference/routines.io.html#text-files

Numerisches Rechnen (Input & Output)

Hier noch ein einfaches Beispiel dazu:

testInput.txt

```
1  # This is a test input
2  A, B, C
3  1, 10, 100
4  2, 20, 200
5  3, 30, 300
6  4, 40, 400
7
```

```
import numpy as np
```

```
data = np.genfromtxt('testInput.txt', comments='#',
                     delimiter=',', skip_header=2, names=['A','B','C'])
# ODER:
#data = np.genfromtxt('testInput.txt', comments='#', delimiter=',', skip_header=1, names=True)

data['A']
```

```
array([ 1.,  2.,  3.,  4.])
```

```
A, B, C = data['A']**2., data['B']**2., data['C']**2.
out = np.array([A,B,C]).transpose()
np.savetxt('testOutput.txt', out, delimiter=', ', fmt='%i',
           header='This is the modified test file\n newA, newB, newC')
```

testOutput.txt

```
1  # This is the modified test file
2  #  newA, newB, newC
3  1, 100, 10000
4  4, 400, 40000
5  9, 900, 90000
6  16, 1600, 160000
7
```

Mehr Details zu genfromtxt():

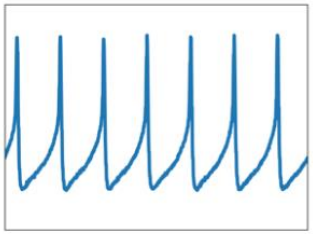
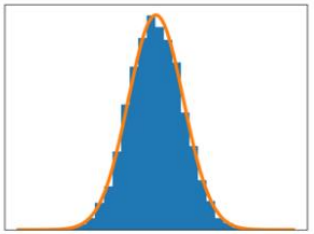
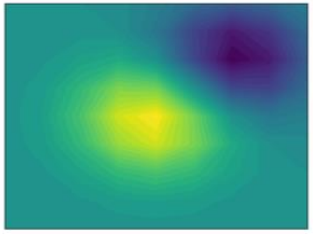
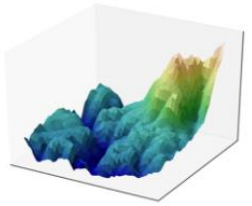
https://docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html

Plotten

matplotlib

http://www.matplotlib.org/

- *Matplotlib* ist eine Programmbibliothek zur Darstellung von zwei- oder dreidimensionale Zeichnungen die mit Hilfe von Punkten, Kurven, Balken oder anderem einen Zusammenhang herstellen.
- Das **Untermodul pyplot** stellt eine prozedurale Schnittstelle zur objektorientierten Plot-Bibliothek von Matplotlib zur Verfügung, welche eine hohe Ähnlichkeit zu MATLAB aufweist
  - Vielzahl an Funktionen, z.B. scatter(), plot(), errorbar(), hist(), hist2d(), contour(), u.v.m. (https://matplotlib.org/api/pyplot_summary.html)

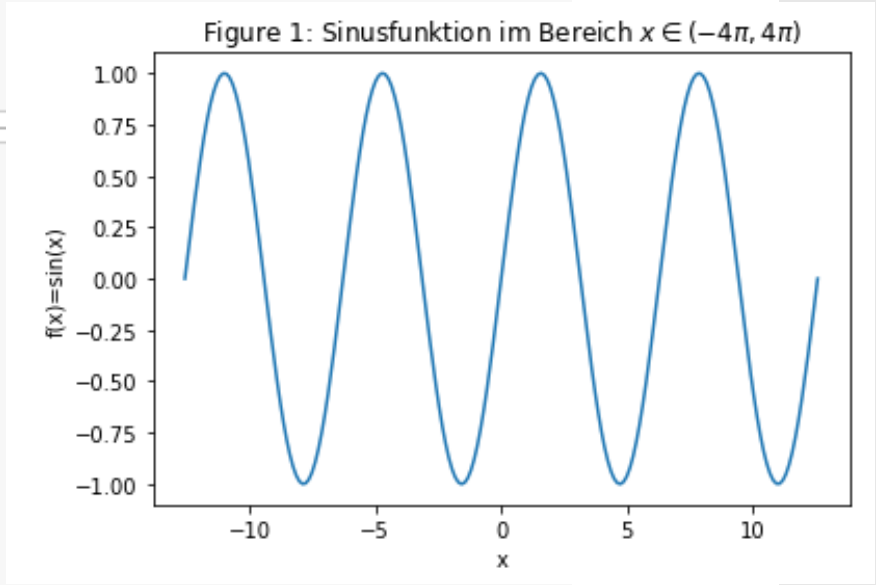





Plotten (Beispiel)

- „einfacher Plot“:

```
#jupyter line magic: sorgt dafür, dass die Plots in die Seite eingebunden werden.
%matplotlib inline
# importiert die Graphik-Bibliothek
import matplotlib.pyplot as plt
import numpy as np
```

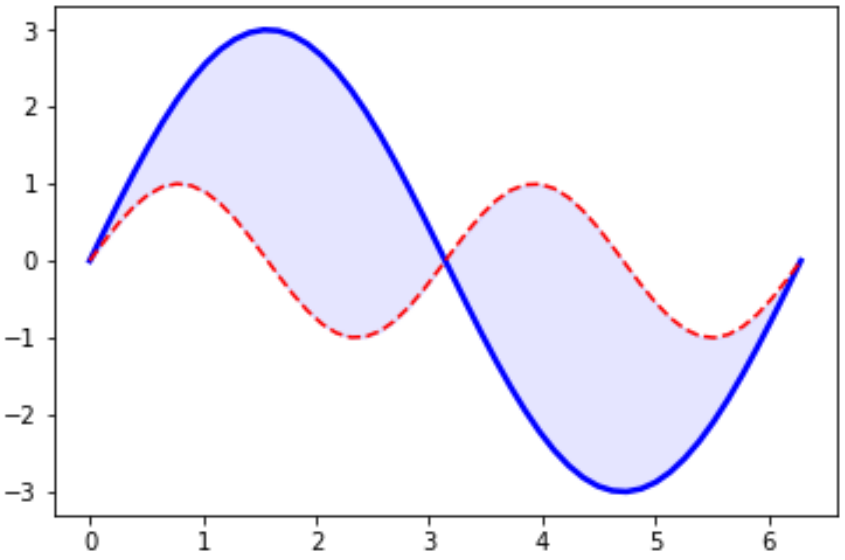
```
# 200 x-Werte zwischen -4 Pi und 4 Pi erstellen
x = np.linspace(-4*np.pi, 4*np.pi, 200)
# y-Werte berechnen
y = np.sin(x)

# Den grundsätzlichen Plot erstellenn
plt.plot(x, y)
# x-Achsen-Beschriftung
plt.xlabel("x")
# y-Achsen-Beschriftung
plt.ylabel("f(x)=sin(x)")
# Titel
plt.title(r"Figure 1: Sinusfunktion im Bereich $x\in(-4\pi, 4\pi)$")
# Plot ausgeben
plt.show()
```



Plotten (Beispiel)

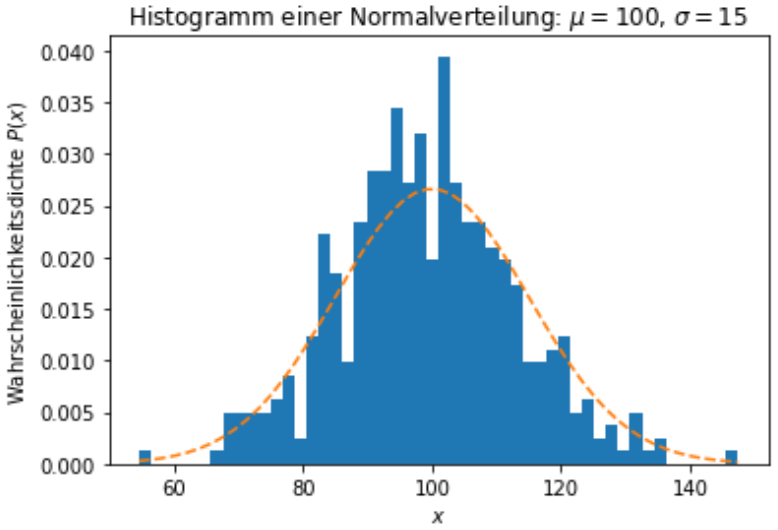
- mehrere Plots und die Fläche dazwischen:

In [3]:
```
X = np.linspace(0, 2 * np.pi, 50)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
plt.plot(X, F1, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, F2, color="red", linewidth=1.5, linestyle="--")
plt.fill_between(X, F1, F2, color='blue', alpha=.1)
plt.show()
```



Plotten (Beispiel)

- Histogramm:




```
# 437 normalverteilte Werte generieren:
mu = 100  # Mittelwert der Verteilung
sigma = 15  # Standardabweichung der Verteilung
x = mu + sigma * np.random.randn(437)

num_bins = 50
# Histogramm der Daten:
n, bins, patches = plt.hist(x, num_bins, normed=True)

# Normalverteilung mit Mittelwert mu und Standardabweichung sigma:
y = ((1 / (np.sqrt(2 * np.pi) * sigma)) *
     np.exp(-0.5 * (1 / sigma * (bins - mu))**2))
plt.plot(bins, y, '--')
plt.xlabel(r'$x$')
plt.ylabel('Wahrscheinlichkeitsdichte $P(x)$')
plt.title(r'Histogramm einer Normalverteilung: $\mu=100$, $\sigma=15$')

plt.show()
```

Wissenschaftliches Arbeiten

SciPy

http://www.scipy.org/

- *SciPy* ist eine Programmbibliothek für verschiedene Problemstellungen im Bereich des computerunterstützten numerischen Rechnens.
  - Numerische Optimierung
  - Lineare Algebra
  - Numerische Integration
  - Interpolation
  - FFT (schnelle Fourier-Transformation)
  - Signalvrarbeitung
  - Bildverarbeitung
  - Numerische Integration gewöhnlicher Differentialgleichungen
  - Symbolische Mathematik („Rechnen mit Buchstaben“)
- enthält als Numerik-Basisbibliothek das Paket NumPy

SciPy: Special Functions

- SciPy bietet eine Vielzahl an gebräuchlichen aber auch weniger üblichen Funktionen in einem auf Performance optimierten Paket an:

  scipy.special

Vollständige Liste unter:
https://docs.scipy.org/doc/scipy/reference/special.html

Available functions¶

Airy functions

airy(z)	Airy functions and their derivatives.
airye(z)	Exponentially scaled Airy functions and their derivatives.
ai_zeros(nt)	Compute *nt* zeros and values of the Airy function Ai and its derivative.
bi_zeros(nt)	Compute *nt* zeros and values of the Airy function Bi and its derivative.
itairy(x)	Integrals of Airy functions

Elliptic Functions and Integrals

ellipj(u, m)	Jacobian elliptic functions
ellipk(m)	Complete elliptic integral of the first kind.
ellipkm1(p)	Complete elliptic integral of the first kind around $m = 1$
ellipkinc(phi, m)	Incomplete elliptic integral of the first kind
ellipe(m)	Complete elliptic integral of the second kind
ellipeinc(phi, m)	Incomplete elliptic integral of the second kind

Bessel Functions

jv(v, z)	Bessel function of the first kind of real order and complex argument.
jve(v, z)	Exponentially scaled Bessel function of order *v*.
yn(n, x)	Bessel function of the second kind of integer order and real argument.
yv(v, z)	Bessel function of the second kind of real order and complex argumen
yve(v, z)	Exponentially scaled Bessel function of the second kind of real order.
kn(n, x)	Modified Bessel function of the second kind of integer order *n*
kv(v, z)	Modified Bessel function of the second kind of real order *v*
kve(v, z)	Exponentially scaled modified Bessel function of the second kind.

SciPy: Special Functions (Beispiel)

Eventuell ist die Bessel'sche Differentialgleichung noch aus der Mechanik oder Quantenmechanik bekannt:

Bessel'sche Differentialgleichung: $x^2 \frac{d^2 f}{dx^2} + x \frac{df}{dx} + (x^2 - \nu^2) f = 0.$

Die Lösungen dieser Differentialgleichung sind die Besselfunktionen.

Es gibt *Besselfunktion 1. Art:* $J_\nu(x) = \sum_{r=0}^{\infty} \frac{(-1)^r (\frac{x}{2})^{2r+\nu}}{\Gamma(\nu+r+1) r!}$

und *Besselfunktion 2. Art:* $Y_\nu(x) = \frac{J_\nu(x) \cos(\nu\pi) - J_{-\nu}(x)}{\sin(\nu\pi)}$,

welche noch den Parameter ν, welcher die Ordnung des Lösung angibt modifiziert wird.

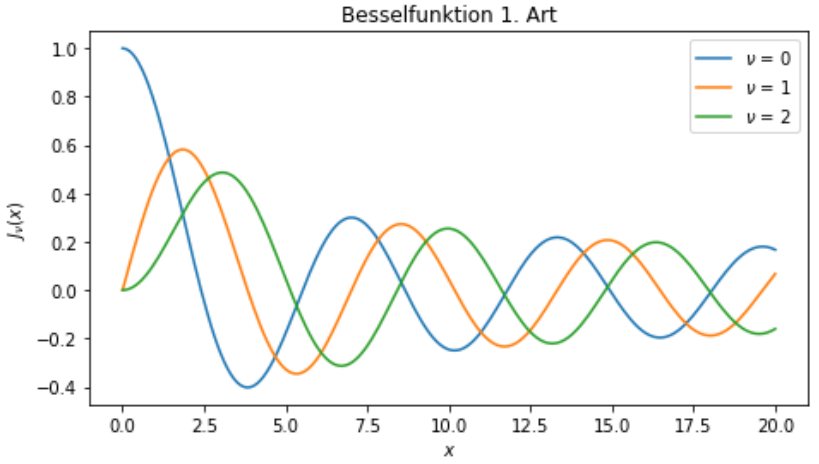
Diese Funktion is in scipy hinterlegt:

Bessel Functions

jv(v, z)	Bessel function of the first kind of real order and complex argument.
jve(v, z)	Exponentially scaled Bessel function of order *v*.
yn(n, x)	Bessel function of the second kind of integer order and real argument.
yv(v, z)	Bessel function of the second kind of real order and complex argument.

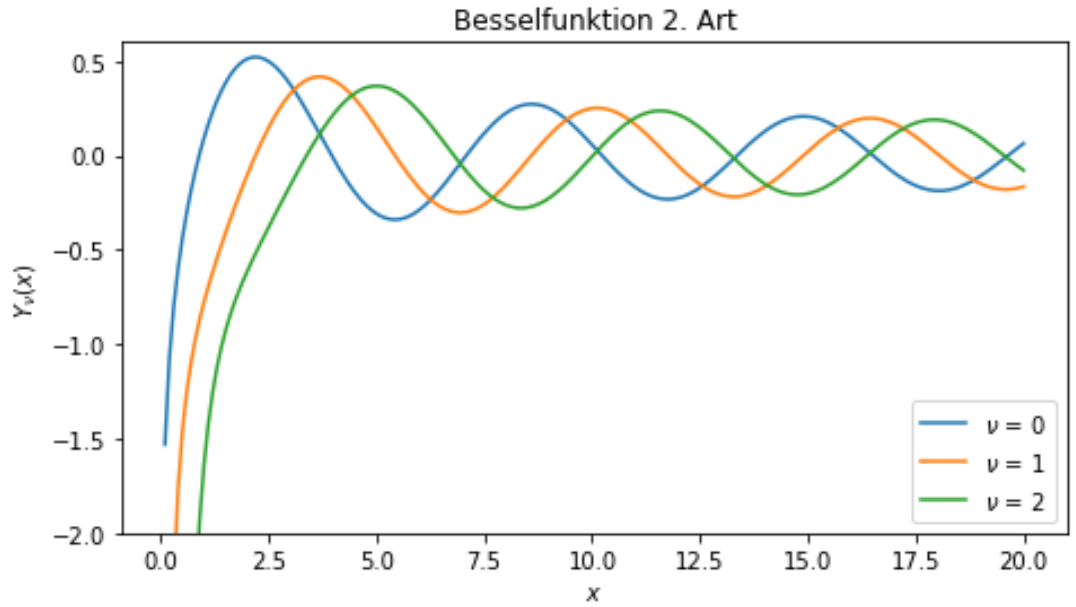
SciPy: Special Functions (Beispiel)

```
# Importieren der Besselfunktionen
from scipy.special import jv, yv
# Definitionsbereich für den Plot festlegen
z = np.linspace(0, 20, 200)
# Es sollen die ersten drei Ordnungen geplottet werden
order = [0,1,2]
# Legt die Größe (in inch) der Abbildung fest.
plt.figure(figsize=(7.5, 4))
for o in order:
    # Berechnen der Besselfunktion
    B = jv(o, z)
    #Plotten und Erstellen eines Labels.
    plt.plot(z, B, label=r'$\nu$ = '+str(o))
    plt.xlabel(r'$x$')
    plt.ylabel(r'$J_\nu(x)$')

plt.title('Besselfunktion 1. Art')
# Die Label können dann in der Legende verarbeitet werden.
plt.legend(loc='best')
plt.show()
```



SciPy: Special Functions (Beispiel)

In [6]:
```
plt.figure(figsize=(7.5, 4))
for o in order:
    D = yv(o, z)
    plt.plot(z, D, label=r'$\nu$ = '+str(o))
    plt.xlabel(r'$x$')
    plt.ylabel(r'$Y_\nu(x)$')

plt.title('Besselfunktion 2. Art')
plt.ylim(-2, .6)
plt.legend(loc='best')
plt.show()
```



SciPy: Numerische Integration

- scipy.integrate stellt in diversen Unterpaketen verschiedene (numerische) Integrations-methoden, sowie Integratoren von gewöhnlichen DGLs bereit:

```
>>> help(integrate)
 Methods for Integrating Functions given function object.

   quad          -- General purpose integration.
   dblquad       -- General purpose double integration.
   tplquad       -- General purpose triple integration.
   fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n.
   quadrature    -- Integrate with given tolerance using Gaussian quadrature.
   romberg       -- Integrate func using Romberg integration.

 Methods for Integrating Functions given fixed samples.

   trapz         -- Use trapezoidal rule to compute integral from samples.
   cumtrapz      -- Use trapezoidal rule to cumulatively compute integral.
   simps         -- Use Simpson's rule to compute integral from samples.
   romb          -- Use Romberg Integration to compute integral from
                    (2**k + 1) evenly-spaced samples.

   See the special module's orthogonal polynomials (special) for Gaussian
      quadrature roots and weights for other weighting factors and regions.

 Interface to numerical integrators of ODE systems.

   odeint        -- General integration of ordinary differential equations.
   ode           -- Integrate ODE using VODE and ZVODE routines.
```

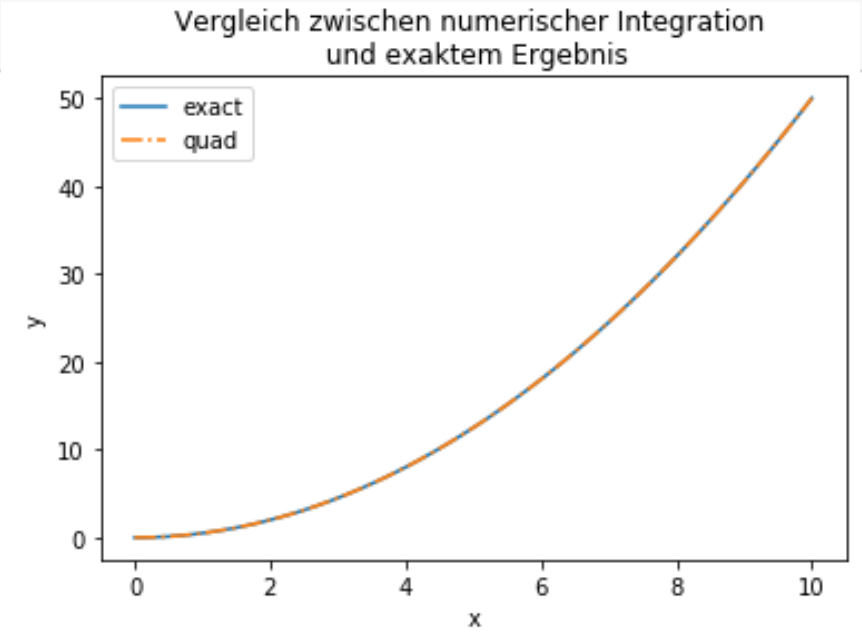
SciPy: Numerische Integration (Beispiel)

```python
# quad ist eine einfache, aber oft ausreichende, Integrationsroutine
from scipy.integrate import quad

def g(x):
    """Funktion, die integriert werden soll"""
    return x

def f(x):
    """Stammfunktion für den Fall, dass x_0=0 ist"""
    return 0.5*x*x

# Berechnung des Integrals
print(r'F(0;5) = {}'.format(quad(g, 0, 5)[0]))

def F(x0, x1, func=f):
    """Eine Funktion, die sich wie die Stammfunktion verhält
    Sie funktioniert sohl für einzelne Werte als auch für Arrays."""
    try:
        F=np.array([quad(func, x0, x1) for x0, x1 in zip(x0, x1)])
        return F[:,0]
    except TypeError:
        F = quad(func, x0, x1)
        return F[0]
```

→ `F(0;5) = 12.5`

SciPy: Numerische Integration (Beispiel)

```
# Untere Grenze
x0 = np.zeros(100)
# Obere Grenze
x1 = np.linspace(0,10,100)

plt.plot(x1, f(x1), label='exact')
plt.plot(x1, F(x0, x1, g), linestyle='-.', label='quad')
plt.legend(loc='best')
plt.title('Vergleich zwischen numerischer Integration \n und exaktem Ergebnis')
plt.xlabel(r'x')
plt.ylabel(r'y')
plt.show()
```



SciPy: optimize

- scipy.optimize hält verschiedene Methoden zur Funktions-Optimierung und -Analyse bereit:
  - Minimierung (oder Maximierung) von Funktionen
  - Lösen von nicht-linearen Problemen
  - Nullstellenbestimmung
  - Regressionsmethoden (non-linear least squares)
  - Kurvenanpassung

SciPy: optimize (Beispiel)

Nullstellensuche:

- Finden von mehreren Nullstellen (oder gar allen) ist sehr aufwendig.
- Ob eine Nullstelle gefunden werden kann und wenn ja welche, hängt auf jeden Fall von der Wahl des Startwertes ab.

In [17]:
```python
# Import der passenden Funktion
from scipy.optimize import root

def f(x):
    """Nach unten verschobene Parabel
    Nullstellen sind x_1=-2**0.5 und x_2=2**0.5"""
    return x*x-2

def jac(x):
    """Jacobi-Matrix der zu untersuchenden Funktion
    Nicht nötig, aber hilfreich. Erhöht die Konvergenzwahrscheinlichkeit."""
    return 2*x

# Ausgabe der Nullstelle (inklusive vieler weiterer Infos, wenn gewünscht)
print("x_1={}".format(root(f, x0=-0.1, jac=jac).x[0]))
# Die gefundene Nullstelle hängt vom Startwert ab
print("x_2=+{}".format(root(f, x0=0.1, jac=jac).x[0]))
```

```
x_1=-1.4142135623730872
x_2=+1.4142135623730872
```

SciPy: optimize (Beispiel)

Kurvenanpassung:

- Anpassung einer (bekannten) Funktion an Messdaten (Vorsicht bei der Parameterauswahl).

SciPy: optimize (Beispiel)

Kurvenanpassung:

- Anpassung einer (bekannten) Funktion an Messdaten (Vorsicht bei der Parameterauswahl).
- Im Beispiel:
  - Exponentiell abklingende Sinus-Funktion, bei der wir die Zerfallszeit τ und die Amplitude A bestimmen wollen.
  - Die Messungenauigkeit wird durch sog. Weißes Rauschen simuliert, welches einer Normalverteilung folgt (Idealisierung des Messprozesses).

```python
# Import von curve_fit
from scipy.optimize import curve_fit

def Expectation(x, A, tau):
    """Funktion, die unseren Prozess beschreibt.
    """
    E = A*np.sin(x)*np.exp(-x/tau)
    return E

def Expectation2(x, A, tau, phi_0):
    """Alternative Funktion, um einen misslungenen Fit zu zeigen"""
    E = A*np.cos(x)*np.exp(-x/tau)
    return E

# Messpunkte
x = np.linspace(0, 20, 50)
# Weißes Rauschen, mit Mittelwert E=0 und Varianz Var=0.8
white_noise = np.random.normal(0.,0.8, 50)
# Messdaten, erzeugt aus dem echten Wert verschmiert mit dem Rauschen.
Data = Expectation(x, 10., 4.)+white_noise
```

SciPy: optimize (Beispiel)

Kurvenanpassung:

- Anpassung einer (bekannten) Funktion an Messdaten (Vorsicht bei der Parameterauswahl).
- Im Beispiel:
  - Exponentiell abklingende Sinus-Funktion, bei der wir die Zerfallszeit τ und die Amplitude A bestimmen wollen.
  - Die Messungenauigkeit wird durch sog. Weißes Rauschen simuliert, welches einer Normalverteilung folgt (Idealisierung des Messprozesses).

```python
# Import von curve_fit
from scipy.optimize import curve_fit

def Expectation(x, A, tau):
    """Funktion, die unseren Prozess beschreibt.
    """
    E = A*np.sin(x)*np.exp(-x/tau)
    return E

def Expectation2(x, A, tau, phi_0):
    """Alternative Funktion, um einen misslungenen Fit zu zeigen"""
    E = A*np.cos(x)*np.exp(-x/tau)
    return E

# Messpunkte
x = np.linspace(0, 20, 50)
# Weißes Rauschen, mit Mittelwert E=0 und Varianz Var=0.8
white_noise = np.random.normal(0.,0.8, 50)
# Messdaten, erzeugt aus dem echten Wert verschmiert mit dem Rauschen.
Data = Expectation(x, 10., 4.)+white_noise
```

SciPy: optimize (Beispiel)

Kurvenanpassung:

```
# Plotten der Messwerte.
plt.plot(x, Data, linewidth=0., marker='o', label='Data')

# Plotten der echten Lösung
plt.plot(x, Expectation(x, 10., 4.), label="True solution")

# Curve Fit, die zu fittenden Parameter werden auomatisch ermittelt.
# popt enhält die best-fit Parameter.
# pcov enthält die Kovarianzmatrix, aus der sich der Fehler berechnen lässt.
popt, pcov = curve_fit(Expectation, x, Data)

# Plot der best-fit Funktion
plt.plot(x, Expectation(x, *popt), linestyle='-.', label='Fit')
print("Fit1: A={}({}), tau={}({})".format(np.round(popt[0], 1), np.round(np.sqrt(np.diag(pcov))[0],1),
                                          np.round(popt[1], 1), np.round(np.sqrt(np.diag(pcov))[1], 1)))

# Das gleiche nochmal für dem anderen Fit.
popt, pcov = curve_fit(Expectation2, x, Data)
plt.plot(x, Expectation2(x, *popt), linestyle='--', label='Fit2')
print("Fit2: A={}({}), tau={}({})".format(np.round(popt[0], 1), np.round(np.sqrt(np.diag(pcov))[0],1),
                                          np.round(popt[1], 1), np.round(np.sqrt(np.diag(pcov))[1], 1)))
plt.legend(loc='best')
plt.xlabel(r'$x$')
plt.ylabel(r'$f(x)$')
plt.show()
```

SciPy: optimize (Beispiel)

Kurvenanpassung:

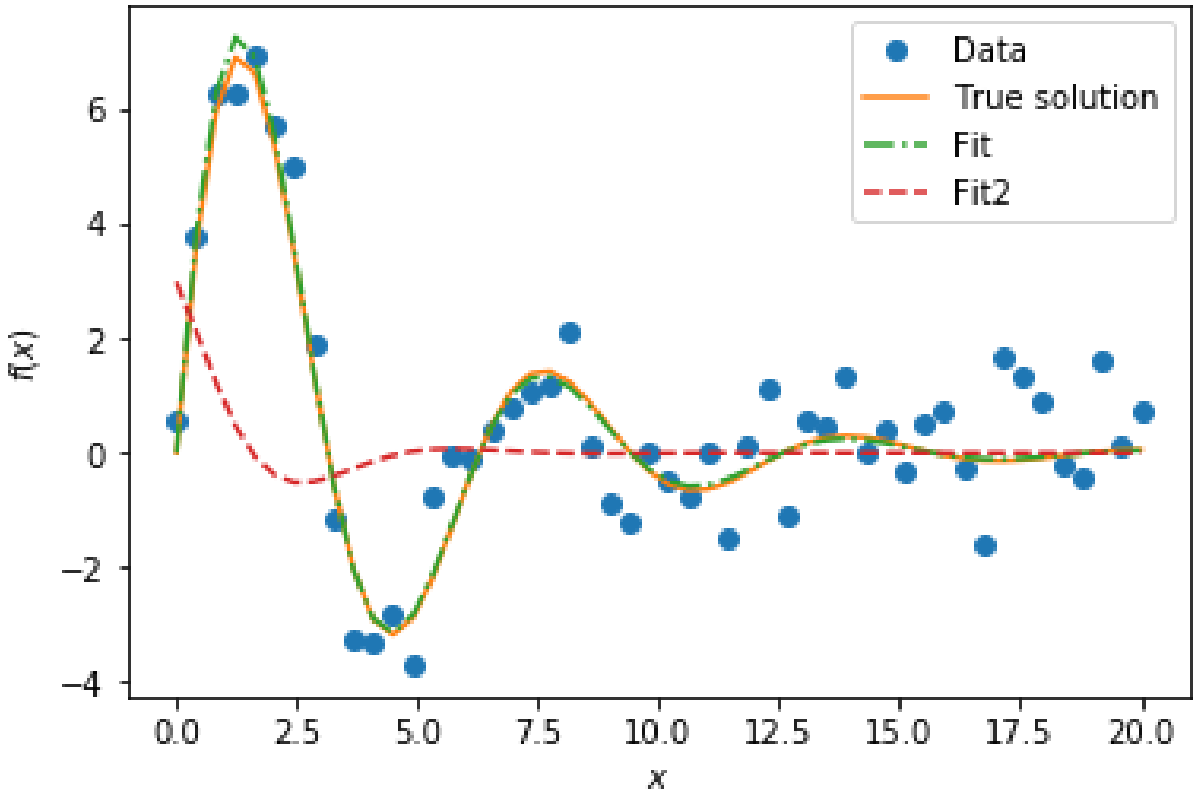
```
# Plotten der Messwerte.
plt.plot(x, Data, linewidth=0., marker='o', label='Data')

# Plotten der echten Lösung
plt.plot(x, Expectation(x, 10., 4.), label="True solution")

# Curve Fit, die zu fittenden Parameter werden auomatisch ermittelt.
# popt enhält die best-fit Parameter.
# pcov enthält die Kovarianzmatrix, aus der sich der Fehler berechnen lässt.
popt, pcov = curve_fit(Expectation, x, Data)

# Plot der best-fit Funktion
plt.plot(x, Expectation(x, *popt), linestyle='-.', label='Fit')
print("Fit1: A={}({}), tau={}({})".format(np.round(popt[0], 1), np.round(np.sqrt(np.diag(pcov))[0],1),
                                          np.round(popt[1], 1), np.round(np.sqrt(np.diag(pcov))[1], 1)))

# Das gleiche nochmal für dem anderen Fit.
popt, pcov = curve_fit(Expectation2, x, Data)
plt.plot(x, Expectation2(x, *popt), linestyle='--', label='Fit2')
print("Fit2: A={}({}), tau={}({})".format(np.round(popt[0], 1), np.round(np.sqrt(np.diag(pcov))[0],1),
                                          np.round(popt[1], 1), np.round(np.sqrt(np.diag(pcov))[1], 1)))
plt.legend(loc='best')
plt.xlabel(r'$x$')
plt.ylabel(r'$f(x)$')
plt.show()
```

```
Fit1: A=10.7(0.9), tau=3.7(0.4)
Fit2: A=3.0(inf), tau=1.6(inf)
```

SciPy: optimize (Beispiel)

Kurvenanpassung:

```
# Plotten der Messwerte.
plt.plot(x, Data, linewidth

# Plotten der echten Lösung
plt.plot(x, Expectation(x,

# Curve Fit, die zu fitten
# popt enhält die best-fit
# pcov enthält die Kovarian
popt, pcov = curve_fit(Expe

# Plot der best-fit Funktio
plt.plot(x, Expectation(x,
print("Fit1: A={}({}), tau=

# Das gleiche nochmal für
popt, pcov = curve_fit(Expe
plt.plot(x, Expectation2(x,
print("Fit2: A={}({}), tau=

plt.legend(loc='best')
plt.xlabel(r'$x$')
plt.ylabel(r'$f(x)$')
plt.show()
```



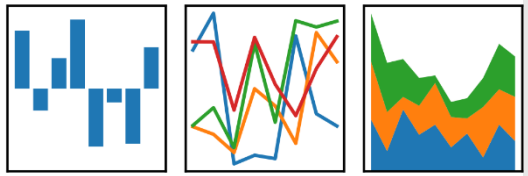

```
Fit1: A=10.7(0.9), tau=3.7(0.4)
Fit2: A=3.0(inf), tau=1.6(inf)
```

Python *fits your needs*

- ***Pandas:*** Zur Datenverarbeitung und Analyse, insbesondere bei großen Datenmengen

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



- **SymPy**: Für symbolisch-mathematische Berechnungen (einfache symbolische Arithmetik bis hin zu Differential-und Integralrechnung sowie Algebra)

- ***Healpy:*** Zur Erzeugung und Bearbeitung von HEALPix (Hierachical Equal Area isoLatitude Pixelation) Karten
- ***Seaborn:*** Zur „high-level“ Visualisierung von statistischen Daten (basierend auf matplotlib)
- … und vieles mehr – was ist das Problem?

Ausblick

Anwesenheitsübung

(Donnerstag, 04.04.19, 14:15-15:45 Uhr, **NB 6/99**):

Besprechung der Blätter 4-6

Moodle-Abschlusstest:

Freitag zwischen 9:15-10:15 Uhr (Gruppe 1)

und 10:30-11:30 Uhr (Gruppe 2)

im SÜDPOL