Wissenschaftliches Rechnen

Einführung in Python

wr@isg.cs.uni-magdeburg.de

Sommer 2019

1 Grundlagen von Python

Als Programmiersprache werden wir in WR die Scripting-Sprache Python[1] verwenden. Diese ist zur Zeit eine der populärsten Programmiersprachen überhaupt.[2]

Die Wahl von Python ist aber von sekundärer Bedeutung, und wir werden die Sprache sehr pragmatisch verwenden, ohne uns im Detail mit dieser auseinander zu setzen. In den meisten Fällen wird es deshalb auch Ihnen überlassen bleiben, wie Sie ein Problem in Python implementieren. Nichtsdestotrotz sollten Sie den Kurs auch als Möglichkeit betrachten, Ihre Programmierkenntnisse zu vertiefen. Eine Auswahl geeigneter Literatur hierzu finden Sie auf der ISIS Seite des Kurses.

1.1 Einführung in Python

Unserem pragmatischem Ansatz folgend, werden wir die grundlegenden Konstrukte von Python an Beispielen erklären (eine Zusammenfassung von Python-Umgebungen, welche von uns empfohlen werden, finden Sie im Anhang A).

Grundlagen Die einfachste Möglichkeit in Python zu programmieren, ist durch eine interaktive Python Konsole, siehe Abb. 1. Beginnen wir mit dem klassischen, ersten Programm:

```
>>> print("Hello World.")
Hello World.
```

Wie in anderen Programmiersprachen sind Schlüsselwörter im Allgemeinen aus dem Englischen entlehnt. Eine Variablenzuweisung erfolgt ebenfalls wie in den meisten anderen Sprachen:

```
>>> x = 3.5
>>> print(x)
3.5
```

In Python gibt es veränderbare (mutable) und nicht veränderbare (immutable) Objekte die streng typisiert sind. Mit dem -Operator wird einem Objekt ein Name zugewiesen über welchen man im weiteren Programmablauf auf das Objekt zugreifen kann. Die Zuweisung eines Namens kann während der Ausführung des Programms verändert werden. Man spricht daher auch von dynamischer Typisierung. Einem Namen bzw. einer Variable können Objekte unterschiedlichen Typs zugewiesen werden:

```
>>> y = True
>>> print(y)
True
>>> y = 3 + 5
>>> print(y)
8
```

Der Typ eines Objekts kann zur Laufzeit bestimmt werden:

[1] http://www.python.org/
[2] http://spectrum.ieee.org/at-work/tech-careers/the-top-10-programming-languages

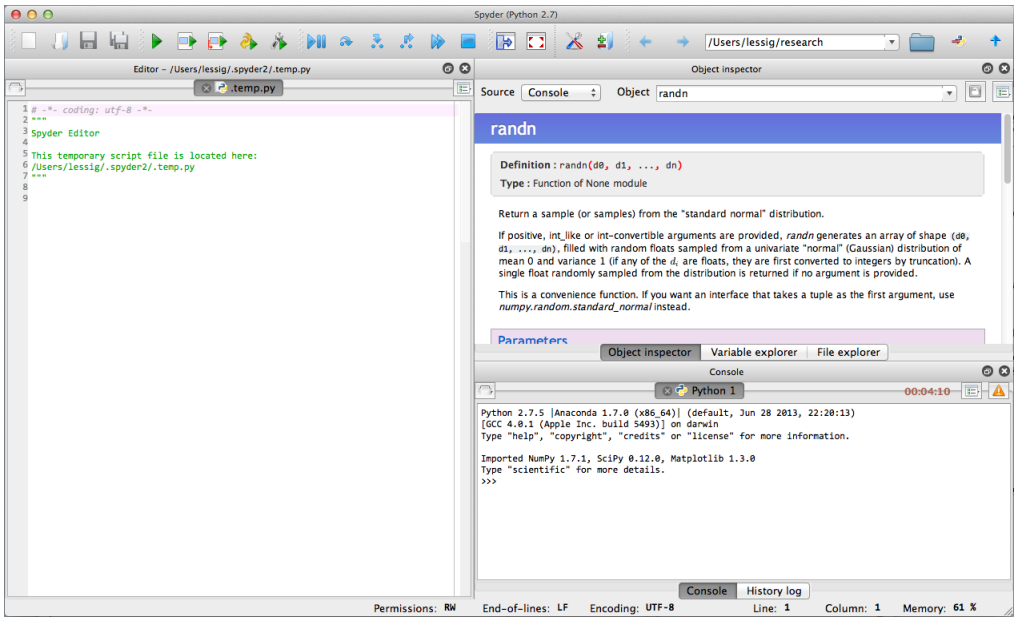

Abbildung 1: Die Spyper Programmierumgebung für Python. Python-Code kann entweder interaktiv im Interpreter eingegeben werden (unten rechts) oder im Editor geschrieben (links) und dann ausgeführt werden. Spyder bietet zusätzlich eine interaktive Hilfe (oben rechts) und einen Inspektor für Variablen.

```
>>> y = 3 + 5
>>> type(y)
<class 'int'>
```

Das letzte Beispiel zeigt eine weitere Eigenschaft von Python: Introspektion und Reflexivität, d.h. Eigenschaften von Objekten können zur Laufzeit festgestellt werden; in den allermeisten Fällen ist auch eine Veränderung möglich. Dies ist in Python auch deshalb möglich, da alle Datentypen durch Objekte realisiert werden.

Entsprechend dem Skriptsprachen-Paradigma wird die Korrektheit von Ausdrücken erst zur Laufzeit überprüft, wenn Code ausgeführt werden soll:

```
>>> y = 3 + 5
>>> z = "House"
>>> z + y
Traceback (most recent call last) :
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'str' and 'int'
```

Python 2 vs. Python 3 Wir verwenden in diesem Modul Python 3.4, das einige Neuerungen gegenüber Python 2.7 brachte[3]. Unterschiede gibt es zum Beispiel beim print-Statement oder der range-Funktion. Falls sie Version 2.7 verwenden möchten, sorgen Sie bitte dafür, dass ihr Code unter Python 3.4 lauffähig ist.

Kontrollstrukturen Zentral für die Entwicklung komplexer Programme sind Kontrollstrukturen. Im Wesentlichen gibt es dabei in Python dieselben Möglichkeiten, welche auch schon aus Java bekannt sind. Obwohl auch die Syntax sehr ähnlich zu Java ist, können gerade kleine Unterschiede dem beginnenden Python Programmierer Schwierigkeiten bereiten. Aber kein Verzagen: Übung macht den Meister!

Eine if-Anweisung hat in Python die Form:

```
>>> y = 3 + 5
>>> if y < 10 :
...     y += 10
...
>>> print(y)
18
```

Ein wichtige Besonderheit von Python ist die Definition von Blöcken ausschließlich mit Hilfe von Einrückung.[4] Dies sehen wir in der dritten Zeile des obigen Beispiels (die drei Punkte am Zeilenanfang stehen nur für die Eingabe eines längeren Ausdrucks auf der interaktiven Konsole). Die Einrückung kann dabei entweder durch Tabs oder auch durch Leerzeichen erfolgen. Beide Arten dürfen aber nicht gemischt werden. Üblich sind 4 Leerzeichen je Einrückung.[5] Eine ähnliche Syntax wie die if-Anweisung hat auch die for-Schleife:

```
>>> for i in range(1,5) :
...     print(i)
...
1
2
3
4
```

Das Beispiel stellt auch die `range()` Funktion vor, welche in vielen Schleifen nützlich ist. Natürlich besitzt auch Python eine while-Schleife.

```
>>> i = 0
>>> while i < 5 :
...     print(i)
...     i += 1
...
0
1
2
3
4
```

Versuchen Sie ein "Gefühl" für eine Sprache zu entwickeln. Dies erspart, die exakte Syntax für jeden Befehl nachschauen zu müssen, und ermöglicht auch Stil-gerechtes programmieren; dies intuitiv zu ermöglichen, war eines der wichtigsten Designkriterien für Python.[6]

Funktionen und Klassen In den meisten Python-Programmen sind Funktionen und Klassen die zentralen Programmstrukturen. Eine einfache Funktion wird in Python wie folgt definiert:

```
>>> def add_one(x) :
...     x += 1
...     return x
...
>>> y = 8
>>> print(add_one(y))
9
```

Wie aus den vorherigen Syntax-Beispielen für Python zu erwarten ist, wird der Kopf der Funktion durch ":" beendet und der Funktionskörper durch Einrückung gekennzeichnet. Der Rückgabewert wird in Python nicht im Funktionskopf definiert sondern ausschließlich im Körper durch `return` angegeben.

Nicht alle Parameter für eine Funktion müssen immer angegeben werden. Falls ein Parameter nicht angegeben wird, dann wir der in der Funktionsdefinition festgelegte Wert verwendet,

[4] Dies ist eine Besonderheit nur in Bezug aus Sprachen wie Java und C/C++. Die ausschließliche Verwendung von Einrückung um Programmcode-Blöcke zu bilden, geht auf die Programmiersprache ABC zurück; `http://python-history.blogspot.de/2011/07/karin-dewar-indentation-and-colon.html`.

[5] PEP 8 - Style Guide for Python Code; `http://legacy.python.org/dev/peps/pep-0008/`.

[6] `http://python-history.blogspot.de/2009/01/pythons-design-philosophy.html`

```
>>> def add_n(x, n = 10) :
...     x += n
...     return x
...
>>> y = 8
>>> print(add_n(y))
18
>>> print(add_n(y, 5))
13
```

Wenn mehrere Argumente Default-Werte haben, dann müssen nicht alle angegeben werden,

```
>>> def add_n_mult_m(x, n = 10, m = 1) :
...     x = (x + n) * m
...     return x
...
>>> y = 8
>>> print(add_n_mult_m(y))
18
>>> print(add_n_mult_m(y, 1))
9
>>> print(add_n_mult_m(y, 1, 2))
18
>>> print(add_n_mult_m(y, m=2))
36
>>> print(add_n_mult_m(y, n=0, m=2))
16
```

Darüber hinaus können Funktionen in Python auch mehrere Rückgabewerte besitzen:

```
>>> def add_and_mult(x) :
...     x1 = x + 10.0
...     x2 = x * 2.0
...     return x1, x2
...
>>> y = 8
>>> y1, y2 = add_and_mult(y)
>>> print("add =", y1, " and mult =", y2)
add = 18.0 and mult = 16.0
```

Die Rückgabe von Objekten ist der bevorzugte Weg, um in Python Daten durch eine Funktion zu manipulieren.

Dass Objekte, welche in einer Funktion verändert werden, immer als Rückgabewert klar erkenntlich sind steht im Gegensatz zu Java oder C/C++, wo Objekte als Referenz übergeben werden. Allerdings werden wir im Folgenden noch sehen, dass auch Python die Möglichkeit bietet Objekte innerhalb einer Funktion zu verändern, ohne diese zurück zu geben.

Die Definition einer leeren Klasse hat in Python die Form:

```
class Vector :
    """A simple vector class."""

    def __init__(self) :
        self.x = 0.0
        self.y = 0.0
        self.z = 0.0

# Instantiate vector
vec3 = Vector()
```

Auch in Python muss zwischen der Definition einer Klasse, welche wir bis jetzt betrachtet haben, und der Instanzierung unterschieden werden. Eine Instanz wird mit Hilfe des Konstruktors erzeugt, welcher

immer den Namen `__init__` hat, und durch den Klassennamen als Funktionsaufruf aufgeführt wird.

Instanzvariablen werden in Python nicht explizit deklariert, sondern durch die Definition in einer Klassenfunktion, was der Konstruktor oder auch eine beliebige andere Klassenfunktion sein kann, erzeugt. Dass die Variablen Instanzvariablen sind, wird eindeutig bestimmt, indem sie zu `self`, der aktuellen Instanz, hinzugefügt werden. Um ein solches Hinzufügen, aber vor allem aber auch den Zugriff auf Instanzvariablen zu ermöglichen, besitzt jede nicht-statische Klassenfunktion `self` als erstes Argument. Das Objekt wird immer automatisch als Argument an die Funktion übergeben und muss nicht als expliziter Parameter aufgeführt werden.

Docstring-Kommentare in Python

Python hat einen empfohlenen Stil für die Dokumentation von Funktionen und Klassen.[7] Dieser ist zwar nicht Teil der Sprachdefinition, er wird jedoch sehr einheitlich befolgt und viel Funktionalität geht verloren, wenn dieser nicht befolgt wird; zum Beispiel basiert die automatische Hilfe der meisten Python-Editoren auf der Einhaltung des kanonischen Dokumentationsstils.

Die Dokumentation von Funktionen erfolgt durch drei Paare von Anführungsstriche, welche zum Öffnen und Schließen des Kommentars verwendet werden. Eine einfache Dokumentation der Funktion `addOne()` ist zum Beispiel:

```
def addOne( x) :
    """Add one to argument and return result."""
    ...
```

Die Funktionalität der Python Funktion sollte in der Dokumentation direkt beschrieben werden (und nicht "'Die Funktion ..."'). Eine vollständige Beschreibung einer Funktion erstreckt sich über mehrere Zeilen:

```
def addN( x, N = 10.0) :
  """Add N to argument x and return result.

  Keyword arguments:
  x -- number to which N is to be added
  N -- increment for x (default: 10.0)

  """
    ...
```

Die Dokumentation von Klassen erfolgt analog zu der von Funktionen, wie wir bereits am Beispiel der Klasse `Vector` gesehen haben.

Natürlich kann auch in Python der Konstruktor Argumente erhalten:

```
class Vector :
    """A simple vector class."""

    def __init__(self, x = 0.0, y = 0.0, z = 0.0) :
        self.x = x
        self.y = y
        self.z = z

    def __str__(self) :
        return "( " + str(self.x) + ", " + str(self.y) + ", " + str(self.z) + ")"

# Instantiate vector
vec3 = Vector( 1.0, 1.0, 0.0)
```

Der Konstruktor `Vector()` ruft dabei die Funktion `__init__()` mit den übergebenen Parametern auf, um die Instanz zu initialisieren. Das gleiche geschieht bei den arithmetischen Operatoren, die durch die Definition der passenden Funktionen, wie zum Beispiel `__add__(self, other)` für die Addition, ebenfalls verwendet werden können. Diese Art von Funktionen werden in Python auch Magic Methods[8]

[8] Ein Überblick über einige Magic Methods `http://www.rafekettler.com/magicmethods.html`

genannt. Weitere magische Funktionen sind `__str__`, welche einen String als lesbare Repräsentation des Objektes zurückgeben soll und z.B. von der `print`funktion verwendet wird, sowie `__cmp__(self, other)` durch welche die Vergleichsoperatoren definiert werden können.

```
>>> print(vec3)
( 1.0, 1.0, 0.0)
>>>
>>> # Change member variable:
>>> vec3.z = 2.0
>>>
>>> print(vec3)
( 1.0, 1.0, 2.0)
```

Da Instanzvariablen in Python immer explizit als solche angegeben werden müssen, zum Beispiel `self.x`, tritt in diesem Beispiel kein Aliasing von Namen auf. Der Zugriff auf Klassenvariablen erfolgt durch `vec3.z`. Im Unterschied zu zahlreichen anderen Objekt-orientierten Programmiersprachen sind in Python keine privaten Datenfelder vorgesehen und alle Variablen einer Klasse können direkt manipuliert werden.

Dies ist wieder eine Manifestierung von Pythons Design-Philosophie: es wird eine "best practice" empfohlen, diese jedoch nicht erzwungen. Nichtsdestotrotz sollten Möglichkeiten wie das nachträgliche Hinzufügen von Datenfeldern oder das Löschen von Datenfeldern in einer existierenden Instanz[9] nur mit größter Vorsicht verwendet werden - nicht jeder in der Sprache gültige Code ist guter Code!

Die Zuweisung von Variablen in Python ist immer nur eine Zuweisung von Referenzen auf ein darunter liegendes Objekts. Zum Beispiel mit der im Vorhergehenden definierte Vektorklasse erhalten wir:

```
v = Vector( 1.0, 1.0, 1.0)
w = v

w.x = 2.0

# Output v.x =  2.0 v.y =  1.0 v.z =  1.0
print("v.x = ", v.x, "v.y = ", v.y, "v.z = ", v.z)
```

Zur Erzeugung "echter" Kopien kann zum Beispiel eine Funktion `copy()` der Klasse hinzugefügt werden.

Python unterstützt auch Vererbung und die Erzeugung von Klassenhierarchien. Wir werden uns mit diesem Thema jedoch nicht beschäftigen und sie sollten dies ggf. selbstständig erkunden.

call by-value vs. by-reference Für die korrekte Benutzung von Funktionen ist es wichtig zu verstehen, ob und wann die Veränderung von Objekten in einer Funktion auch außerhalb sichtbar ist. Zum Beispiel, welchen Wert hat `y` nach dem der folgende Python Code ausgeführt wurde?

```
>>> def assignFive(x) : x = 5
...
>>> y = 2
>>> assignFive(y)
```

Die Ausgabe ist

```
>>> print(y)
2
```

Damit verhält sich Python genau so, wie wir es aus Java gewohnt sind. Bei der Übergabe eines Parameters wird ein Zeiger auf das zugrundeliegende Objekt kopiert. Das heißt man kann diesen Zeiger auf ein anderes Objekt zeigen lassen (hier durch die Zuweisung `x = 5`) ohne das original Objekt zu ändern.

Auf der anderen Seite kann man aber auch mithilfe des Zeigers das Objekt verändern, sodass diese Änderung auch ausserhalb der Funktion sichtbar ist. Im folgenden findest du ein Beispiel für einen solchen sogenannten "Seiteneffekt".

```
>>> def assignFiveX( vec) : vec.x = 5
...
>>> vec3 = Vector( 1, 2, 3)
>>> assignFiveX(vec3)
>>> print(vec3.x)
5
```

Wir sehen, dass Zuweisungen nicht nach dem Funktionsaufruf sichtbar sind, eine Veränderung der Felder eines Objektes aber sehr wohl.

Weiterführendes Thema: Analogie zu Zeigern in C

Eine alternative Erklärung ist der folgende C-Pseudocode, welcher das gleiche Verhalten wie unsere Python-Funktion `assignFive()` zeigt:[10]

```
void assignFive(int *n) {
  int newval = 5;
  n = &newval;
}

int y = 2;
assignFive(&i);
```

Weiterführendes Thema: Lambda Kalkül in Python

Funktionen und Klassen sind die wesentlichen Bausteine für moderne imperativer Programmierung. Neben imperativer Programmierung unterstützt Python auch funktionale Programmierung. Ein einfaches Beispiel soll hier genügen:

```
>>> def addN( n) : return lambda x : x + n
...
>>> f = addN(10)
>>> g = addN(20)
>>>
>>> f(10)
20
>>> g(10)
30
>>> addN(3)(10)
13
```

Die Funktion `addN` gibt eine namenlose Funktion zurück! Funktionen können also in gewisser Weise wie Variablen behandelt werden und als Parameter oder Rückgabewerte dienen. Viele Probleme lassen sich so besonders elegant lösen. Lambda Ausdrücke sind in Python insbesondere in Verbindung mit `filter()`, `map()` und `reduce()` nützlich, welche wir in Kürze kennenlernen werden.

Modules and Packages Klassen bieten eine Möglichkeit Daten und die dazugehörige Funktionalität in einer programmtechnischen Einheit zu verbinden. Dies ist insbesondere dann nützlich, wenn viele Objekte gleichen Typs aber mit unterschiedlichen Daten benötigt werden. Oft möchte man jedoch auch nur Funktionalität bündeln, zum Beispiel alle Funktionen, welche zur Auswertung eines gegebenen mathematischen Funktionsklasse notwendig sind. Oder man möchte verschiedene Klassen, zum Beispiel eine Anzahl von geometrischen Formen, welche man implementiert hat, zu einer Einheit zusammenfassen. Für solche und ähnliche Aufgaben stellt Python Module (modules) und Pakete (packages) zur Verfügung.

Ein Modul ist in Python die gesamte Funktionalität in einer Python Datei (mit der Endung *.py). Die Funktionalität kann durch `import` in anderen Python-Skripten, oder im interaktiven Interpreter, zur Verfügung gestellt werden. Zum Beispiel wenn wir unsere Vektor-Klasse in einer Datei `vec.py` speichern,

```
# vec.py
class Vector :
    """A simple vector class."""

    def __init__(self, x = 0.0, y = 0.0, z = 0.0):
        self.x = x
        self.y = y
        self.z = z
```

dann können wir die Funktionalität nach dem Import des Moduls verwenden:

```
>>> import vec
>>> vec3 = vec.Vector( 1.0, 2.0, 3.0)
```

Die Vorteile von Modulen sind die Kapselung der Funktionalität, was zum Beispiel Details der Implementierung vor dem Nutzer verbirgt und äquivalente Änderung ermöglicht, und die Wiederverwendbarkeit. Nach dem Import eines Moduls können wir auf dessen Funktionalität wie auf die Felder einer Klasse zugreifen. Dies vermeidet, dass Namenskonflikte zwischen verschiedenen Modulen entstehen. Wenn man nicht den Dateinamen für ein Modul verwenden möchte, so ist dies auch möglich:

```
>>> import vec as myvec
>>> vec3 = myvec.Vector( 1.0, 2.0, 3.0)
```

Eine dritte Möglichkeit des Imports besteht darin, die Namen des Moduls direkt in den globalen Namensraum zu importieren:

```
>>> from vec import *
>>> vec3 = Vector( 1.0, 2.0, 3.0)
```

Diese Variante sollte, wegen der Möglichkeit von Aliasing und Namenskonflikten, jedoch nur in Ausnahmefällen verwendet werden. Möglich ist auch nur einzelne Funktionen aus einem Modul zu importieren.

Pakete sind eine Erweiterung des Modul Konzepts: es sind Verzeichnisse, welche Python Module enthalten. Zum Beispiel die numpy Bibliothek, welche wir im Detail in Abschnitt 2 betrachten werden, ist ein Paket. Der Zugriff auf die Funktionalität erfolgt wieder durch `import` und Module in einem Paket werden erneut wie Klassenfelder adressiert:

```
>>> import numpy as np
>>> A = np.eye( 3)
>>> B = np.linalg.inv(A)
```

Listen und Wörterbücher Die meistverwendeten Container-Datenstrukturen, welche von Python zur Verfügung gestellt werden, sind Listen und assoziative Dictionaries (oder Wörterbücher; wir werden i.A. den Englischen Begriff verwenden).

Eine Liste ist eine geordnete Menge von Elementen beliebigen Types. Zum Beispiel:

```
>>> A = [1, 'House', 3.2]
```

Auf die Elemente einer Python Liste kann wie auf Arrayelemente in anderen Programmiersprachen zugegriffen werden:

```
>>> A[0]
1
>>> A[1]
'House'
```

Python stellt darüber hinaus Funktionalität zum "Slicing" von Listen zur Verfügung. Zum Beispiel, auf alle Elemente in einer Teilsequenz kann wie folgt zugegriffen werden:

```
>>> A[0:2]
[1, 'House']
```

Zuszätzlich stehen zahlreiche Klassenfunktionen zum Arbeiten auf Listen zur Verfügung. Ein Element kann zum Beispiel wie folgt an eine existierende Liste angefügt werden:

```
>>> A.append( 'dog')
>>> A
[1, 'House', 3.2, 'dog']
```

Listen können auch selbst Listen enthalten. Der Zugriff erfolgt wie bei mehrdimensionalen Arrays.

```
>>> B = ['M', 'P', 'G', 'I', 4]
>>> A.append(B)
>>> A
[1, 'House', 3.2, 'dog', ['M', 'P', 'G', 'I', 4]]
>>> A[4][0]
'M'
```

Die Länge einer Liste kann durch `len()` bestimmt werden.

```
>>> len(A)
5
>>> len(A[4])
5
```

Listen stellen mit `pop()` auch Funktionalität zur Verfügung, welche es ermöglicht diese als Stack zu benutzen. Allerdings ist in den allermeisten Fällen die Verwendung von spezialisierten Datenstrukturen für diesen Zweck zu empfehlen.

Weiterführendes Thema: Manipulation von Listen

Nützlich für die Manipulation von Listen sind die Funktionen `filter()` und `map()`. Diese Funktionen liefern, genau wie `range()` einen Iterator zurück, den man gut in einer for-Schleife verwenden kann. Um explizit eine Liste zu erhalten muss man `list()` aufrufen.

```
>>> def lessThan3(x) : return x < 3
...
>>> B = 10.0 * np.random.rand( 10)
>>> B
array([ 4.86689676,  8.61164328,  9.6015649 ,  6.40094038,  2.42500922,
        5.75461623,  1.33572532,  5.89913972,  7.21317618,  7.07337869])
>>> list(filter( lessThan3, B))
[2.4250092170890456, 1.3357253168218197]
>>>
```

Funktoren wie `lessThan3()` erlauben es Listen in sehr kompakter Art und Weise zu manipulieren. Ein weiteres Beispiel ist:

```
>>> import math
>>> list(map( math.sqrt, range(1,10)))
[1.0, 1.41, 1.73, 2.0, 2.23, 2.44, 2.64, 2.82, 3.0]
```

Hier haben wir die Funktion `sqrt()` aus der Math-Bibliothek als Funktor verwendet. In der Form von List Comprehensions erlauben for-Schleifen auch eine sehr kompakte Erzeugung von Listen.

```
>>> squares = [x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Anstatt von `x**2` kann natürlich jede unäre Funktion verwendet werden, welche auf Zahlen definiert ist. Das obige Beispiel kann auch durch eine Bedingung `P(x)` erweitert werden:

```
 >>> def f(x) : return math.sqrt(x) + 3.0
...
>>> S = 10.0 * np.random.rand(10)
>>> S
array([ 3.18,  8.90,  3.77 ,  4.98,  9.09,
        1.32,  7.18,  0.17,  6.73,  9.32])
>>> T = [f(x) for x in S if x > 2.0]
>>> T
[4.78, 5.98, 4.94, 5.23, 6.01, 5.68, 5.59, 6.05]
>>> len(T)
8
```

In List Comprehensions sind auch verschachtelte for-Schleifen und somit recht komplexe Strukturen möglich.

Dictionaries sind Datenstrukturen, bei welchen der Zugriff auf Elemente über `keys` anstatt über Indizes wie bei Listen (oder Arrays) erfolgt. Zum Beispiel:

```
>>> Age = { 'John' : 35, 'Carl' : 41 }
>>> Age['John']
35
>>> Age['Carl']
41
```

Die `keys` für ein Dictionary müssen Python immutable Typen sein, wie zum Beispiel string's oder int's. Wie bei Listen kann auch für Dictionaries Comprehension benutzt werden, um diese zu erzeugen:

```
>>> E = { x : f(x) for x in range(1,10) }
>>> E
{1: 4.0, 2: 4.41, 3: 4.73, 4: 5.0, 5: 5.23, 6: 5.44, 7: 5.64, 8: 5.82, 9: 6.0}
```

Schlußbemerkung Wir haben hier nur die elementaren Grundlagen von Python besprechen können. Sie sollten sich bemühen, im Verlauf des Kurses Ihre Python Kenntnisse zu vertiefen und effektive Lösungen für die an Sie gestellten Programmieraufgaben zu finden. Machen Sie sich dazu mit dem Gebrauch Python Documentation[11] vertraut. Im nächsten Kapitel besprechen wir die Numpy Bibliothek. Auch hier wird sich ein Blick in die Dokumentation lohnen.[12]

1.2 Übungsaufgabe: Vektoren und Überladen von Operatoren

Ziel der Übungsaufgabe ist es, sich ein grundlegendes Verständnis der Syntax von Python zu erarbeiten.

Implementieren Sie die Klassen `Vector2` und `Vector3` für die Repräsentation von zwei und dreidimensionalen Vektoren. Die Klassen sollen die natürlichen Operationen (+,-,*,:) auf Vektoren, sowie Skalar- und Kreuzprodukt zur Verfügung stellen. Ausserdem sollen Sie, soweit dies nützlich erscheint, die üblichen arithmetischen Operatoren überladen. Welche Vor- und Nachteile bietet das Überladen von Operatoren? Stellen Sie Ihren Code in einem Modul `vector` zur Verfügung. Kommentare sollen im Python Stil erfolgen, so dass Spyder's Hilfe diese verwenden kann.

2 Lineare Algebra in Python

In den kommenden Wochen werden wir sehr häufig das Python Packet NumPy[13] verwenden, welches speziell für die Umsetzung von linearer Algebra in Python entwickelt wurde. Die Effizienz, welche insbesondere für die Verarbeitung von großen Vektoren und Matrizen notwendig ist, wird dabei durch eine Implementierung der NumPy-Funktionalität in C/C++ realisiert.

2.1 Darstellung von Vektoren und Matrizen in Python

Um NumPy verwenden zu können, müssen wir zunächst das package importieren:

```
>>> import numpy as np
```

Die Benennung als `np` ist dabei nicht zwingend, aber weit verbreitet.

[11] `https://docs.python.org/2.7/`.
[12] `http://docs.scipy.org/doc/numpy-1.8.1/reference/`.
[13] `http://www.NumPy.org/`

Das NumPy Array Objekt Das Herzstück von NumPy ist ein n-dimensionales Array, welches zur Repräsentation von Daten dient. Wir werden hauptsächlich den Fall $n = 1$ und $n = 2$ benötigen, d.h. wenn das Array einem Vektor oder einer Matrix entspricht.

Die einfachste Möglichkeit, ein NumPy Array zu erzeugen, ist durch eine Python Liste:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> a
array([ 1.,  2.,  3.])
```

Mehrdimensionale Arrays können durch verschachtelte Listen erzeugt werden:

```
>>> b = np.array( [ [1.0, 2.0, 3.0], [4.0, 5.0, 6.0] ])
>>> b
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

Arrays sind Python-Objekte und haben damit Funktionen definiert, durch welche wir mit diesen arbeiten können. Die Größe eines NumPy Arrays erhalten wir zum Beispiel durch:

```
>>> a.shape
(3,)
>>> b.shape
(2, 3)
```

wohingegen die Anzahl der Element durch `size` bestimmt werden kann:

```
>>> b.size
6
```

Neben der Konstruktion durch Python Listen stellt NumPy auch Funktionen zur Verfügung, welche häufig verwendete Arrays konstruieren. Die Einheits-"Matrix" erhält man mit

```
>>> a_id = np.eye(3)
>>> a_id
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

und ein Array, in welchem alle Elemente einen konstanten Wert haben durch

```
>>> cvals = np.pi * np.ones( (3,3,3))
>>> cvals
array([[[ 3.14159265,  3.14159265,  3.14159265],
        [ 3.14159265,  3.14159265,  3.14159265],
        [ 3.14159265,  3.14159265,  3.14159265]],

       [[ 3.14159265,  3.14159265,  3.14159265],
        [ 3.14159265,  3.14159265,  3.14159265],
        [ 3.14159265,  3.14159265,  3.14159265]],

       [[ 3.14159265,  3.14159265,  3.14159265],
        [ 3.14159265,  3.14159265,  3.14159265],
        [ 3.14159265,  3.14159265,  3.14159265]]])
>>> cvals.shape
(3, 3, 3)
```

Der Zugriff auf die Elemente eines NumPy Arrays erfolgt wie bei anderen Arrays (und Python Listen) durch

```
>>> a[0]
1.0
>>> b[0,1]
2.0
```

Der Zugriff auf nicht existente Elemente erzeugt einen Laufzeitfehler:

```
>>> a[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index out of bounds
```

Mit dem [] Operator können wir auch Teile, sogenannte Slices, eines Arrays erhalten. Das erfolgt ähnlich dem Slicing bei Python Listen. Die Zeilen und Spalten werden hier allerdings durch Komma getrennt.

```
>>> a = np.array( [ [1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0] ])
>>> a
array([ 1.,  2.,  3.],
      [ 4.,  5.,  6.],
      [ 7.,  8.,  9.])
>>> b = a[0]
>>> b
array([ 1.,  2.,  3.])
>>> a[:, :2]                # bedeutet: "alle Zeilen, Spalte 0 und 1"
array([[ 1.,  2.],
       [ 4.,  5.],
       [ 7.,  8.]])
>>> a[1:, ]                 # bedeutet: "ab der 1. Zeile, alle Spalten"
array([[ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
>>> a[[0,2],:]              # bedeutet: "Zeile 0 und 2, alle Spalten"
array([ 1.,  2.,  3.],
      [ 7.,  8.,  9.])
```

So lassen sich auch bequem die Zeilen oder Spalten einer Matrix tauschen:

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

Es ist wichtig zu beachten, dass Slices nur Referenzen erzeugen und kein neues Datenobjekt. In diesem Punkt unterscheiden sich NumPy-Arrays und die Python-Container.

Für die meisten Anwendungen ist ein elementweiser Zugriff jedoch nicht notwendig und auch ineffizient (wir werden dies noch genauer in einer Hausaufgabe betrachten). Soweit es möglich ist, sollten die durch NumPy zur Verfügung gestellten Operationen zum Arbeiten mit Arrays verwendet werden.

Einige Beispiele sind:

```
>>> b = np.array( [ [1.0, 2.0, 3.0], [4.0, 5.0, 6.0] ])
>>> b.max()
6.0
>>> b.mean()
3.5
>>> b.cumsum()
array([  1.,   3.,   6.,  10.,  15.,  21.])
>>> b.fill( 3.0)
>>> b.nonzero()
(array([0, 0, 0, 1, 1, 1]), array([0, 1, 2, 0, 1, 2]))
```

Viele dieser Funktionen sind auch als nicht-Klassenfunktionen verfügbar, zum Beispiel:

```
>>> np.nonzero(b)
(array([0, 0, 0, 1, 1, 1]), array([0, 1, 2, 0, 1, 2]))
```

Zuweisungen von NumPy Arrays erzeugen nur eine neue Referenz auf das bestehende Datenobjekt (dies entspricht einer shallow copy). Man hat also:

```
>>> b = np.array( [ [1.0, 2.0, 3.0], [4.0, 5.0, 6.0] ])
>>> b
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> d = b
>>> d
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> d[0,1] = 99.0
>>> b
array([[  1.,  99.,   3.],
       [  4.,   5.,   6.]])
```

Um tatsächlich ein neues Objekt zu erzeugen, muss man die Funktion `copy()` verwenden:

```
>>> b = np.array( [ [1.0, 2.0, 3.0], [4.0, 5.0, 6.0] ])
>>> d = b.copy()
>>> d[0,1] = 99.0
>>> b
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> d
array([[  1.,  99.,   3.],
       [  4.,   5.,   6.]])
```

Numpy implementiert auch die klassischen arithmetische Operationen für Arrays, zum Beispiel:

```
>>> a = np.array( [ [1, 2], [ 2, 1] ])
>>> b = np.array( [ [3.0, 2.0], [ 2.0, 3.0] ])
>>> a + b
array([[ 4.,  4.],
       [ 4.,  4.]])
```

Analog zur Addition sind auch alle anderen arithmetischen Operationen auf NumPy Arrays elementweise definiert. Für Multiplikation haben wir also:

```
>>> a * b
array([[ 3.,  4.],
       [ 4.,  3.]])
```

Um Matrix Multiplikation durchzuführen, muss die Funktion `dot()` verwendet werden. Um eine Matrix mehrfach mit sich selbst zu multiplizieren eignet sich die Funktion `linalg.matrix_power()`.

```
>>> a.dot(b)
array([[ 7.,  8.],
       [ 8.,  7.]])
>>> np.linalg.matrix_power(a,2)
array([[5, 4],
       [4, 5]])
```

Es existiert in Numpy zwar auch ein Matrix Objekt, für welches der Multiplikationsoperator zur Matrixmultiplikation verwendet werden kann. Dieses werden wir aber im gesamten Semester nicht verwenden. Weitere mathematische Operationen, welche für NumPy Arrays verfügbar sind, sind zum Beispiel:

```
>>> np.exp(a)
array([[ 2.71828183,  7.3890561 ],
       [ 7.3890561 ,  2.71828183]])
>>> np.log(a)
array([[ 0.        ,  0.69314718],
       [ 0.69314718,  0.        ]])
```

Für die meisten arithmetischen Operationen müssen NumPy Array Operanden die gleiche Größe besitzen. NumPy besitzt jedoch auch Regeln, welche es ermöglichen, mit Objekten unterschiedlicher Größe zu arbeiten. Da diese jedoch schnell zu Verwirrung führen, werden wir deshalb auf ihre Verwendung verzichten.

Weiterführendes Thema: Datentypen in Numpy

Bis jetzt haben wir die Frage, welchen Datentyp die Elemente eines NumPy Arrays besitzen, ignoriert, obwohl dies für die Genauigkeit von Berechnungen von entscheidender Bedeutung ist. NumPy stellt im Gegensatz zu Python eine Vielzahl von numerischen Datentypen zur Verfügung: `bool`, `int`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `float`, `float16`, `float32`, `float64`, `complex`, `complex64`, `complex128`.[14]

Der Datentyp der Element eines NumPy Arrays kann zur Laufzeit ermittelt werden:

```
>>> a = array( [ [1.0, 2.0], [ 2.0, 1.0] ])
>>> a.dtype
dtype('float64')
```

In diesem Fall haben wir zum Beispiel einen 64-bit Gleitkommazahl Datentyp. Im Allgemeinen bestimmt NumPy den Datentyp automatisch anhand der Daten. Eine leichte Veränderung des obigen Beispiels führt zum Beispiel zu:

```
>>> a = array( [ [1, 2], [ 2, 1] ])
>>> a.dtype
dtype('int64')
```

Bis auf wenige Ausnahmen werden wir immer Gleitkommazahlen verwenden. Eine Möglichkeit dies sicherzustellen ist die explizite Spezifikation des Datentyps bei der Erzeugung eines Arrays:

```
>>> a = array( [ [1, 2], [ 2, 1] ], dtype='float64')
>>> a.dtype
dtype('float64')
```

Wir werden die Bedeutung verschiedener Datentypen für die Genauigkeit numerischer Berechnungen in der nächsten Woche noch genauer betrachten.

MatPlotLib In vielen Anwendungen im wissenschaftlichen Rechnen ist eine graphische Darstellung von Daten wesentlich effizienter als die Begutachtung der numerischen Werte. Wir können zum Beispiel die folgenden zwei "Zahlenfolgen" betrachten,

...	1.572	1.575	1.578	1.581	1.584	1.588	1.591	1.594	1.597	1.600	...
...	1.	1.	1.	0.9999	0.9999	0.9999	0.9998	0.9997	0.9996	0.9996	...

welche es schwer machen, den Zusammenhang zwischen diesen zu verdeutlichen, oder ihre graphische Darstellung in obiger Abbildung.

Die Zahlen und die Grafik wurden wie folgt erzeugt:

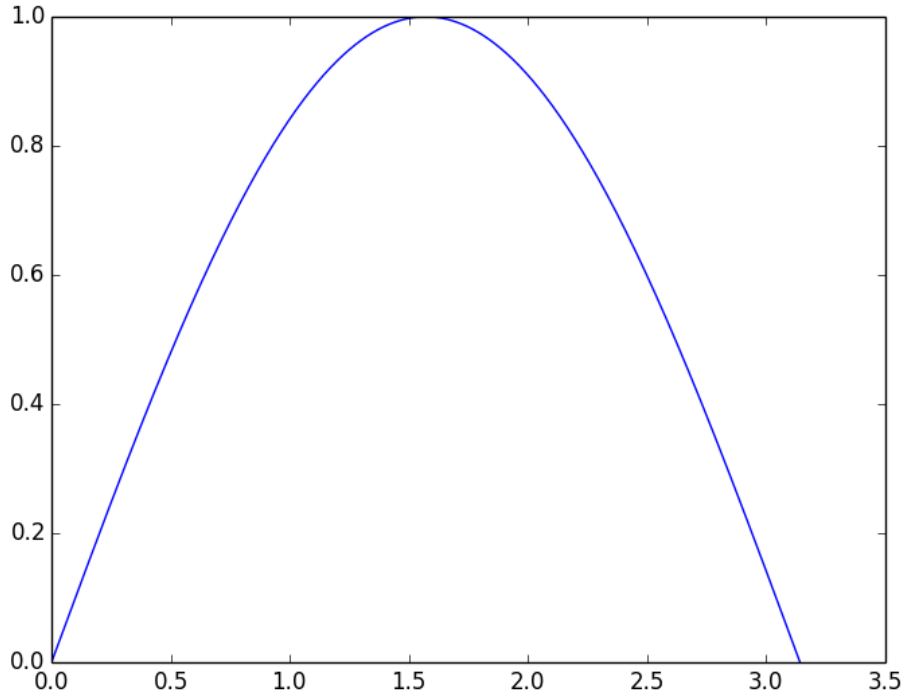

Abbildung 2: Sinus-Schwingung.

```
>>> x = linspace( 0.0, np.pi, 1000)
>>> y = sin(x)
>>> x[500:508]
array([ 1.572,  1.576,  1.579,  1.582,  1.585,  1.588,  1.591,  1.594])
>>> y[500:508]
array([ 1.,  1.,  1.,  0.9999,  0.9999,  0.9999,  0.9998,  0.9997])
>>> plot( x, y)
[<matplotlib.lines.Line2D object at 0x11299f7d0>]
>>> show()
```

MatPlotLib stellt zahlreiche Funktionen zur professionellen Erstellung und Bearbeitung von Graphen zur Verfügung. Sie sollten die von Ihnen benötigten Befehle selbstständig in den einschlägigen Referenzen nachschlagen.[15]

SciPy SciPy ist eine Erweiterung von NumPy, welche Algorithmen zur Integration, Optimierung, Interpolation, Fourier Transform, Signalverarbeitung und vielen anderen Themen implementiert.[16] In der Tat sind viele der Verfahren und Algorithmen, welche wir in diesem Kurs kennenlernen und implementieren wollen, bereits in SciPy vorhanden. Auch nur einen Überblick über die verschiedenen Module zu geben, würde den Rahmen dieses Tutoriums sprengen. Deshalb betrachten wir hier beispielhaft Integration. Die einfachste Möglichkeit ist die Verwendung der Funktion `quad()` mit deren Hilfe numerisch Integriert werden kann:

```
>>> import scipy.integrate
>>> scipy.integrate.quad( lambda x : sin(x), 0.0, 2.0 * np.pi)
(2.221501482512777e-16, 4.3998892617845996e-14)
```

Der erste Rückgabewert ist die numerische Abschätzung für das Integral, und der zweite Werte eine Schätzung des Fehlers (der korrekte Wert für das Integral ist natürlich 0.0). Wie wir hier bereits sehen, müssen wir uns bei numerischen Berechnungen im Allgemeinen mit Abschätzungen zufrieden geben, obwohl diese hoffentlich mit genügend Aufwand auch beliebig genau gemacht werden können.

[15] Ein guter Startpunkt ist `http://matplotlib.org/`.
[16] `http://www.scipy.org/`

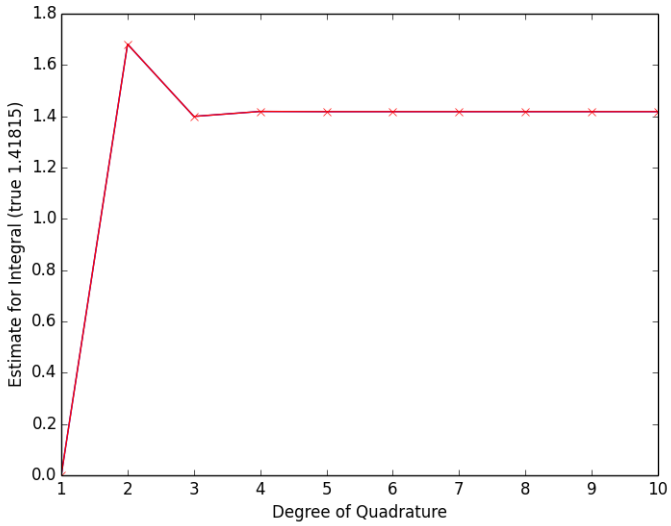

Abbildung 3: Fehler für Gauss-Legendre Quadratur mit Grad N für $f(x) = y = sin(x)/x$.

SciPy stellt verschiedene Quadraturalgorithmen zur Berechnung bzw. Abschätung von Integralen zur Verfügung. Dass das Integrationsproblem im Allgemeinen schwieriger ist, als es vielleicht auf den ersten Blick erscheint, wird am Beispiel von $f(x) = y = sin(x)/x$ deutlich:

```
>>> for i in range(1,6):
...   scipy.integrate.fixed_quad(lambda x : sin(x)/x, 0.0, 2.0 * np.pi, (), i)
(2.4492935982947064e-16, None)
(1.681162749828647, None)
(1.3994617320654141, None)
(1.4188623502594748, None)
(1.4181345524085365, None)
```

Der Wert des Integrals ist hier

$$\int_0^{2\pi} \frac{\sin(x)}{x}\, dx \approx 1,41815$$

Wir sehen, dass das Quadraturverfahren und die gewählten Parameter einen großen Einfluß auf die erzielte Genauigkeit haben können, siehe auch Abbildung 3. Ihnen ein Verständnis zu vermitteln, unter welchen Umständen welches Quadraturverfahren zu bevorzugen ist, und wie man Verfahren für spezielle Anwendungen entwickelt, ist eines der Ziele dieser Veranstaltung.

Anhang A: Python Programmierumgebungen

Hinweis: Es gibt keine Python-Implementierung oder Programmierumgebung, welche von uns gefordert wird. Im Folgenden beschreiben wir die Umgebung, welche wir verwenden und welche in unserer Erfahrung für die Ziele der Veranstaltung geeignet ist.

Numpy, Scipy, und MatPlotLib sind nicht Teil der Python Standardbibliothek. Eine Installation der einzelnen Pakete ist möglich, und insbesondere unter Linux einfach umzusetzen, kann jedoch auch schnell zu Inkompatibilitäten und anderen Problemen führen. Eine Python Distribution, welche alle von uns verwendeten Pakete bereits enthält und für Linux, MacOS und Windows verfügbar ist, ist Anaconda.[17] Neben Python installiert Anaconda auch den iPython Interpreter[18] und die Spyder Entwicklungsumgebung.[19] Letztere werden wir auch in der Vorlesung und den Übungen verwenden.

[17] http://continuum.io/downloads
[18] http://ipython.org/
[19] https://code.google.com/p/spyderlib/

Spyder bietet neben einem interaktiven Interpreter und einem Variablen Inspektor auch einen Editor. Dieser lässt jedoch einige Funktionalitäten, insbesondere beim Debuggen und in Bezug auf Codevervollständigung, vermissen, welche in professionellen Entwicklungsumgebung heute selbstverständlich sind. Für das Schreiben größerer Programme empfehlen wir daher PyCharm,[20] welches mit einer akademischen Lizenz kostenfrei verwendet werden kann.

[20] `http://www.jetbrains.com/pycharm/`