

Python-Einführung

Benjamin Rommel
Proseminar 'Linux für Umsteiger'
Universität Heidelberg

April 20, 2008

Contents

1	Einleitung	3
2	Features	3
3	Programmier-Grundlagen	5
3.1	Ausgabe	5
3.2	Variablen	5
3.2.1	Deklaration	5
3.2.2	Rechenoperationen	6
3.2.3	Strings	7
3.3	Listen	7
3.3.1	Allgemeines	7
3.3.2	Listenoperationen	8
3.4	Schleifen und bedingte Anweisungen	9
3.4.1	if-Abfrage	9
3.4.2	while-Schleife	9
3.4.3	for-Schleife	10
3.5	Funktionen	10
4	Python in High Performance Computing	12
4.1	pyMPI	12
4.1.1	Programmierung	12
4.1.2	Fazit	15
4.2	pyPar	15
4.2.1	Programmierung	16
4.2.2	effizientere Programmierung	17
4.2.3	Fazit	18
4.3	Performance ist nicht alles!	18
4.4	Fazit	20

1 Einleitung

Der niederländische Programmierer Guido van Rossum entwickelte zu Beginn der 1990er Jahre die Programmiersprache Python für das von Andrew Tanenbaum entwickelte verteilte System Amoeba. Grundlage für die Arbeit an Python war die von ihm mitentwickelte Programmiersprache ABC, wobei Python die Kritiken an ABC konstruktiv umsetzen sollte. Die Ziele für Python, welche sich van Rossum setzte, formulierte er 1999 in seinem 'Computer Programming for Everybody' genannten Vorschlag, welchen er bei der DARPA, der Defense Advanced Research Projects Agency, einer amerikanischen Behörde des Verteidigungsministeriums, welche Forschungsarbeiten finanziell unterstützt, einreichte. Zu den Zielen zählten unter anderem, dass die Sprache zwar einfach und intuitiv gehalten werden soll, aber dennoch mächtig ist. Zudem sollte Python vollständig OpenSource bleiben und bei der Entwicklung kurze Entwicklungszeiten ermöglichen.

Vieles davon wurde erreicht. Van Rossum spielt noch heute eine leitende Rolle in der Python Software Foundation, welche die Rechte über Python innehält und mit Hilfe einer großen Community die Entwicklung von Python vorantreibt. Aktuell ist Python in Version 2.5, aber die Version 3 ist bereits für Mitte 2008 angekündigt. Seit seiner Einführung gewinnt Python immer mehr Anhänger und die Entwicklung schreitet, wie bereits zuvor erwähnt, konstant weiter. Trotz seiner Einfachheit wird Python auch bei großen Projekten verwendet und ein Teil dieses Erfolges lässt sich darauf zurückzuführen, dass Programme anderer Sprachen als Module eingebettet werden können und Python selbst als Skriptsprache in anderen Anwendungen wie Blender, Maya oder Gimp verwendet werden kann.

2 Features

Einfachheit Eine der größten Stärken Python ist zweifelsohne, dass es sehr leicht zu lesen ist. Bei C/C++ muss der Benutzer lediglich darauf achten, dass die Inhalte von Schleifen, Funktionen oder bedingten Anweisungen innerhalb zweier geschweiften Klammern stehen. Ihm steht es aber völlig frei, wie er den Code anordnet. Völlig anders ist die Syntax in Python aufgebaut! Python verzichtet zum einen auf die, aus vielen Programmiersprachen bekannten, Semikolons am Ende einer Programmierzeile, was nicht nur das Schreiben des Zeichens in jeder Zeile erspart, sondern auch den Benutzer auch dazu zwingt, jede Programmierzeile in eine eigene Zeile zu schreiben und nicht mehrere in einer zusammenzufassen, was bei C/C++ ja zweifelsohne möglich ist. Maßgeblich an der Übersichtlichkeit beteiligt sind die eingerückten Blöcke. Anstatt Code in geschweiften Klammern zusammenzufassen, werden die Blöcke, welche zu Schleifen oder Funktionen gehören, in einem bestimmten Verhältnis zum linken Bildschirmrand eingerückt, was nicht nur das Schreiben der Klammern

erspart, sondern eine große Übersichtlichkeit des Codes schafft. Der Benutzer wird regelrecht dazu gezwungen übersichtliche Programmcodes zu schreiben, was auch wiederum dazu führt, dass man, sofern man Python beherrscht, problemlos die Programmcodes anderer Autoren lesen kann.

Erweiterbarkeit Eine weitere, sehr bedeutende Stärke von Python ist die Erweiterbarkeit. Der eigene Programmcode lässt sich leicht auf mehrere Dateien und Module verteilen und bietet dazu auch die passenden Funktionalitäten, um untereinander auf Funktionen und Codes der Module zugreifen zu können. Die große Besonderheit ist nun der Umstand, dass man diese Funktionalitäten nicht nur bei Python-Modulen hat, sondern allgemein für Module, d.h. auch für Module, die etwa in C oder Java geschrieben wurden. Ist eine sehr hohe Performance eines bestimmten Moduls unabdingbar für den Erfolg eines Projektes, kann man dieses problemlos in C schreiben und als Modul in das Python-Projekt einbinden und auf die Funktionen zugreifen, als wäre das Modul in Python geschrieben worden.

Portabilität Python wurde komplett in C geschrieben und ist daher auf fast allen gängigen Systemen verfügbar, sofern dort ein ANSI C Compiler vorhanden ist. Es ist gleichgültig, ob ein Programm unter UNIX, Windows oder einem anderen System geschrieben wurde, der kompilierte Code ist unter allen anderen Plattformen ebenso lauffähig. Allerdings kann nicht sichergestellt werden, dass ein Python-Code in einer späteren Version von Python noch vollständig ausführbar ist.

Speichermanagement Alle Variablen in Python sind dynamisch, d.h. es sind keine expliziten Typdefinitionen erforderlich, was das Handhaben der Variablen sehr einfach gestaltet. Darüber hinaus verfügt Python über eine Garbage Collection, d.h. es wird die korrekte Freigabe des reservierten Speichers sichergestellt. So einfach nun die Handhabung der Variablen auch ist, die Verarbeitung ist nicht so effizient wie in C. Im Gegenzug hat man keine Probleme wie bei dynamischen Arrays in C.

Robustheit Treten während der Ausführung des Projektes Fehler auf, zeigt Python die als Exceptions bezeichneten Fehler nicht nur detailliert an, sondern bietet darüber hinaus auch die Möglichkeit mit Hilfe des dem Interpreter beigefügten Exceptions Handlers Einfluss auf den weiteren Programmverlauf zu nehmen. So können auftretende Fehler beispielsweise ignoriert oder der weitere Programmverlauf umgelenkt werden, um nur zwei Möglichkeiten zu nennen.

Kompilierung Python zählt zu den interpretierten Sprachen, was bedeutet, dass die Kompilierungszeit kein Faktor für die Programmentwicklung darstellt. Allerdings ist Python keine interpretierte Sprache im klassischen Sinne, da

diese im allgemeinen erheblich langsamer sind als kompilierte Programme. In Python geschriebene Quellcodes werden, ähnlich zu Java, byte-kompiliert und sind damit näher an der Geschwindigkeit von Programmen in Maschinensprache dran ohne die Möglichkeiten einer interpretierten Sprache zu verlieren. Im Punkt der reinen Performance ist Python allerdings Hochsprachen, welche in Maschinensprache kompiliert werden, zweifelsohne unterlegen!

3 Programmier-Grundlagen

3.1 Ausgabe

Die Textausgabe geschieht in Python mit dem Befehl `print`. Im allgemeinen hat man drei Möglichkeiten für die Textausgabe: man möchte entweder einen Text, Variablen oder feste Werte ausgeben. Für die Ausgabe eines festen Wertes schreiben wir einfach den auszugebenden Wert, hier im Beispiel 1000, nach einem Leerzeichen hinter den `print`-Befehl:

```
print 1000
```

Um eine Variable auszugeben ersetzt man einfach den Zahlwert durch den Variablennamen, ein Beispiel hierzu finden im Kapitel über Variablen. Um einen Text auszugeben, muss dieser durch Hochkommata oder Anführungszeichen begrenzt werden. Die Ausgabe des Ausdruckes `Hallo Welt` auf dem Bildschirm würde z.B. so aussehen:

```
print "Hallo Welt"
```

Näheres dazu lässt sich im Kapitel über Strings nachlesen. Man kann aber natürlich nur einen der drei Fälle benutzen, sondern auch eine Verkettung davon, um beispielsweise einen statischen Text mit Variablen auszugeben. Die verketteten Ausgaben werden dabei durch ein Komma von der vorherigen Ausgabe getrennt. Nehmen wir an, wir haben eine Variable `i` mit dem Wert 100, dann würde der Code

```
print "i =", i
```

folgenden Text ausgeben:

```
i = 100
```

3.2 Variablen

3.2.1 Deklaration

Ein großer Vorteil von Python im Gegensatz zu anderen Programmiersprachen wie C++ ist, dass bei den Variablen kein expliziter Variablentyp angegeben werden muss. Die einzige Beschränkung bei der Benennung der Variablen ist,

dass keine reservierten Namen verwendet werden dürfen, wie z.B. `print` oder `def`. Außerdem dürfen die Variablennamen nicht mit Zahlen anfangen. Eine sinnvolle Variablenbenennung ist natürlich empfehlenswert, sodass man aus dem Variablennamen den Zweck der Variable ersieht, aber es ist nicht zwingend erforderlich. Die Wertzuweisung geschieht wie bei anderen bekannten Sprachen durch das Gleichheitszeichen:

```
i = 10
i = "Hallo"
print i
```

Zuerst wird der Variable `i` der Wert 10 zugewiesen und im nächsten Schritt wird der String `Hallo` übergeben. Bei der Ausgabe des Wertes von `i` durch den Aufruf von

```
print i
```

wird der zuletzt gültige Wert, der String `Hallo`, ausgegeben. Auffallend ist außerdem, dass bei der Programmierung mit Python kein Zeichen übergeben werden muss, welches das Ende einer Befehlszeile signalisiert, wie beispielsweise das Semikolon bei C/C++.

3.2.2 Rechenoperationen

Für Python gelten die gleichen Konventionen wie in der Mathematik: Es werden stets Division und Multiplikation vor Subtraktion und Addition ausgeführt und es werden immer erst Ausdrücke in Klammern ausgewertet. Ein kleines Beispiel hierzu:

```
zahl = (10 + 20 / 5) * 2
print zahl
```

Dieser Code würde 28 ausgeben. Zuerst wird die Klammer $(10 + 20 / 5)$ ausgewertet, d.h. man dividiert 20 durch 5 und addiert das Ergebnis, vier, zu 10 hinzu, bevor das Ergebnis des umklammerten Ausdrucks mit zwei multipliziert wird.

Zu beachten ist jedoch, dass das Ergebnis der Division ganzer Zahlen das Ergebnis in die nächstkleinere ganze Zahl umgerechnet wird. Die Rechnung $3/4$ würde folglich 0 ergeben, da das Ergebnis, 0.75 auf die nächstkleinere ganze Zahl gesetzt wird. Um dies zu verhindern, muss eine der an der Division beteiligten Zahlen eine Gleitkommazahl sein. Damit werden automatisch alle an der Division beteiligten Zahlen intern zu Gleitkommazahlen und man erhält das korrekte Ergebnis. Die Operation $3/4.0$ würde demnach zur Ausgabe von 0.75 führen.

3.2.3 Strings

Wie in dem Eingangsbeispiel bei der Variablendeklaration bereits gezeigt, ist auch für die Deklaration von Strings kein expliziter Datentyp anzugeben. Stattdessen übergibt man einfach den Ausdruck an den Variablennamen. Hierbei hat man jedoch zwei Möglichkeiten: man kann die Werte sowohl durch Hochkommata als auch durch Anführungszeichen eingrenzen. Das heißt die beiden folgenden Zeilen führen zum gleichen Ergebnis:

```
string = "Hallo!"  
string = 'Hallo!'
```

Auf den ersten Blick mag das kein großer Vorteil gegenüber anderen Programmiersprachen sein, die eine einzige Syntax hierfür festlegen, aber der Vorteil wird schnell deutlich, wenn man folgenden Text ausgeben möchte:

```
Hallo, "Jack"
```

Würde man bei dieser Ausgabe die Anführungszeichen zum begrenzen des Strings verwenden, so würde Python das gewünschte Ergebnis nicht darstellen können. Den obigen Text könnte man hingegen mit der folgenden Zeile problemlos ausgeben:

```
string = 'Hallo, "Jack"'
```

Findet Python lediglich am Anfang und am Ende der Zeile die begrenzenden Zeichen, d.h. Hochkommata oder Anführungszeichen, so werden alle Zeichen innerhalb des Strings als Zeichenkette ausgewertet und nicht auf die Bedeutung der Zeichen geachtet.

Ebenfalls möglich ist die Addition von mehreren Strings zu einem neuen String, wobei hier gilt, dass die einzelnen Strings von unterschiedlichen Zeichen begrenzt werden können. Der Ausdruck

```
string = 'Hallo, "Jack"' + ", wie geht es dir?"
```

würde die Ausgabe

```
Hallo, "Jack", wie geht es dir?
```

erzeugen.

3.3 Listen

3.3.1 Allgemeines

Listen sind in Python Variablen, die mehrere Werte beinhalten, vergleichbar mit den Arrays in C/C++, wobei die Listen, wie auch die Variablen in Python, nicht auf einen Variablentyp beschränkt sind. Das heißt innerhalb einer Liste können die Werte unterschiedliche Datentypen haben. Als Beispiel hierzu wählen wir folgende Liste:

```
list = [10, 20, 'Hallo', "Test"]
```

Die Elemente der Liste werden durch zwei eckige Klammern begrenzt und die einzelnen Elemente werden durch Kommas von einander getrennt. Eine Speicherreservierung ist nicht nötig, das übernimmt Python für uns, ebenso die Freigabe des Speichers. Geben wir bei der Ausgabe kein Listenindex an, so wird die ganze Liste ausgegeben. Wir können jedoch auch mittels Indexangabe auf ein bestimmtes Element zugreifen. Zu beachten ist, dass die Nummerierung bei 0 beginnt und nicht bei 1, was intuitiver wäre. Haben wir n Elemente, so sind die Nummerierungen von 0 bis n-1. Der Aufruf

```
print list[1]
```

führt zur Ausgabe von 20, dem zweiten Element der Liste. Es kann auch ein bestimmtes Intervall der Liste ausgegeben werden. Hierzu muss man in den eckigen Klammern die linke und rechte Intervallgrenze angeben und sie durch einen Doppelpunkt trennen. Die Ausgabe der Elemente 1 bis 3 würde demnach durch

```
print list[1:3]
```

erfolgen.

3.3.2 Listenoperationen

Listenlänge Die Länge einer Liste erhält man durch den Befehl `len`, wobei man in Klammern die Liste angeben muss, deren Länge ermittelt werden soll. Mit der Liste aus dem letzten Kapitel würde der Aufruf

```
print len(list)
```

zu der Ausgabe 4 führen.

Elemente hinzufügen Um ein Element an das Ende einer Liste hinzuzufügen, sollte man auf `append(x)` zurückgreifen, wobei x das anzuhängende Element ist. Die Funktion `extend(L)` fügt an die Liste eine weitere Liste L an das Ende an und `insert(i, x)` fügt ein Element x an die Stelle i der Liste ein. Beispiel:

```
list = [10, 20, 'Hallo', "Test"]
list.append(30)
list.extend(['40', '50'])
list.insert(1, 0)
print list
```

Diese Zeilen verändern die Ausgangsliste list, sodass sie am Ende wie folgt aussieht:

```
[10, 0, 20, 'Hallo', 'Test', 30, '40', '50']
```

Suchen & Sortieren Um eine Liste zu sortieren, stellt Python den parameterlosen Befehl `sort()` zur Verfügung. `index(x)` sucht das erste Element der Liste, das mit `x` übereinstimmt und gibt seinen Index aus und `count(x)` zählt das Vorkommen des Elements `x` in der Liste.

3.4 Schleifen und bedingte Anweisungen

3.4.1 if-Abfrage

Sollen bestimmte Aktionen nur unter gewissen Bedingungen ausgeführt werden, greift man auf bedingte Anweisungen zurück, die in Python, wie fast jeder anderen Programmiersprache auch, als if-else-Abfrage realisiert ist. Zuerst ein kleines Beispiel:

```
i = 20
if i<30:
    print 'kleiner als 30'
elif i>30:
    print 'groesser als 30'
else:
    print 'exakt 30'
```

Wir definieren uns eine Variable mit einem bestimmten Wert und schauen, ob die Zahl größer, kleiner oder gleich 30 ist. Zu Beginn kommt stets eine if-Abfrage, die eine bestimmte Situation abfragt, in diesem Fall, ob die Variable `i` kleiner als 30 ist. Bei den Vergleichen von Variablen und Werten, bzw. anderen Variablen hat man die Standardvergleichsoperatoren zur Auswahl: `>` (größer), `<` (kleiner), `==` (gleich), `>=` (größer gleich) und `<=` (kleiner gleich). Nach der Bedingung folgt ein Doppelpunkt und in der nächsten Zeile steht die Aktion, welche geschehen soll, wenn die Bedingung erfüllt ist, also `i` kleiner 30 ist. Wichtig ist, dass die Aktionen eingerückt sein müssen, es ist egal ob ein Leerzeichen oder einen Tabulator-Sprung weit, wobei standardmäßig die Einrückungen einem Tabulator-Sprung entspricht. Ist die Bedingung der if-Abfrage nicht erfüllt, geschieht die elif-Abfrage. elif steht für else if, also eine alternative Bedingung, wenn die if-Bedingung und mögliche vorhergehende elif-Bedingungen nicht erfüllt wurden. Ist `i<30`, wird hier der entsprechende String ausgegeben, wenn nicht, d.h. weder die if-Bedingung noch die elif-Bedingung(en) erfüllt wurden, geschieht die Aktion, welche bei else angegeben wurde. Die Aktionen der einzelnen Abfragen können sich über mehrere Zeilen erstrecken, man sollte allerdings beachten, dass alle im gleichen Abstand zum linken Rand stehen, also identisch weit eingerückt sind. Die else- und ggf. elif-Bedingung(en) sind nicht zwingend nötig, wenn eine einfache if-Abfrage ausreicht.

3.4.2 while-Schleife

Zuerst das Beispiel:

```

i = 15
while i>10:
    print i
    i = i-1

```

Die while-Schleife führt so lange ihre Aktionen aus, bis ein Abbruchkriterium erreicht wird, welches in der ersten Zeile nach dem while steht. Die Vergleichsoperatoren sind hier identisch zu denen der if-Abfragen. Wir schauen in unserem Beispiel, ob eine Zahl i größer als 10 ist, und falls ja, werden die zur Schleife gehörenden Aktionen ausgeführt. Diese sind, wieder wie bei if-Abfragen, im gleichen Abstand zum linken Bildschirmrand eingerückt.

3.4.3 for-Schleife

Anders als in vielen anderen Programmiersprachen ist die for-Schleife in Python realisiert:

```

a = ['Hund', 'Katze', 'Maus']
for x in a:
    print x

```

Während man in C/C++ beispielsweise das Start- und Abbruchkriterium, sowie den Iterationsschritt angeben kann bzw. muss, iteriert die for-Schleife in Python über ein Intervall, meist Listen.

3.5 Funktionen

Um öfter verwendete Codefragmente schnell und bequem zur Hand zu haben, sollte man den Code in Funktionen zusammenfassen. Eine typische Funktion in Python hätte folgenden Aufbau:

```

def function(n):
    """Dies ist eine Funktion."""
    n = n*2000
    print n

function(5)

```

Um eine Funktion zu definieren, verwendet man den Befehl def, gefolgt vom Namen der Funktion, der, wie auch bei Variablen, Aufschluss über Sinn und Zweck der Funktion geben sollte. Möchte man der Funktion Parameter übergeben, so werden diese in runden Klammern dem Funktionsnamen nachgestellt. Der Doppelpunkt am Zeilenende signalisiert nun, dass die eigentliche Funktion beginnt. Wie auch bei bedingten Anweisungen oder Schleifen müssen die Ausdrücke, die zur Funktion gehören, im gleichen Abstand zum linken Rand eingerückt werden.

Docstrings Die erste Zeile ist ein sogenannter Docstring. Docstrings sind Strings, welche den Sinn und Zweck der Funktion beschreiben und zum guten Programmierstil unbedingt dazugehören. Der Anfang und das Ende der Docstrings wird jeweils durch drei Anführungszeichen gekennzeichnet. Da Funktion beliebig lang und kompliziert sein können, sollte man die wichtigsten Informationen zur Funktion hier erwähnen. Die Docstrings können dabei auch über mehrere Zeilen gehen, wobei hier folgendes beachtet werden muss: gehen Docstrings über mehrere Zeilen, so wird bei der Ausgabe die erste Zeile direkt am linken Bildschirmrand ausgegeben, aber alle weitere Zeilen werden entsprechend ihres Abstandes zum Bildschirmrand eingerückt. Beispiel:

```
def function(n):
    """Dies ist docstring
    \nuber mehrere Zeilen."""
    n = n*2000
    print n

print function.func_doc
```

Dieser Code würde die folgende Ausgabe erzeugen:

```
Dies ist docstring
    \nuber mehrere Zeilen.
```

Möchte man, dass alle Zeilen am linken Bildschirmrand angrenzen, so muss entsprechend der Docstring, ab der zweiten Zeile (!), zurückgerückt werden:

```
def function(n):
    """Dies ist docstring
    \nuber mehrere Zeilen."""
    n = n*2000
    print n
```

Es gilt aber zu beachten, dass die eigentlichen Funktionsausdrücke nach wie vor im vorgegebenen Abstand zum linken Rand stehen müssen, es werden lediglich die Docstring-Zeilen zurückgerückt.

Rückgabewert Die Funktion kann, wie in unserem Beispiel, direkt eine Ausgabe ihres Ergebnis erledigen, oder aber, was ebenfalls häufig erwünscht ist, das Ergebnis ihrer Berechnung zurückgeben. Hierfür gibt es den Befehl `return`, der den nachgestellten Wert bei beenden der Funktion zurückgibt:

```
def function(n):
    """Dies ist docstring"""
    n = n*2000
    return n
```

```
i = function(1)

print i
```

4 Python in High Performance Computing

In der Einleitung habe ich geschrieben, dass Python mit dem Ziel entwickelt wurde, leicht programmier- und erlernbar zu sein und trotz einer einfachen Syntax in der Mächtigkeit anderen Hochsprachen in nichts nachstehen soll. Im zweiten Kapitel habe ich die Grundlagen der Programmiersprache erklärt und gezeigt, dass Python in der Tat sehr einfach zu lesen und programmieren ist und wer bereits über Programmiererfahrung in anderen Hochsprachen verfügt und mit dem Grundwissen vertraut ist, sollte mühelos auf Python umsteigen können. Nun steht der letzte Punkt noch aus, bei dem ich zeigen werde, dass Python auch die angesprochene Mächtigkeit besitzt, in dem ich auf die Möglichkeiten mit Python im Hochleistungsrechensektor verweise.

4.1 pyMPI

Als erstes möchte ich pyMPI vorstellen, ein plattformunabhängiges OpenSource-Projekt von Forschern des Lawrence Livermore National Laboratory, welches MPI¹ in den Python-Interpreter integriert hat und damit parallele Verarbeitung mit Python ermöglicht. pyMPI stellt ein Modul für den Interpreter dar, d.h. um auf die Funktionalitäten, welche pyMPI bereitstellt, zugreifen zu können, muss das Modul vorher importiert werden (`import mpi`). Gestartet wird eine parallele Anwendung durch den Aufruf

```
mpirun -np N pyMPI
```

wobei N die Anzahl der Knoten darstellt. Alle pyMPI-eigenen Variablen sowie die Funktionen beginnen stets mit 'mpi.' gefolgt vom Variablennamen oder dem Funktionsnamen.

4.1.1 Programmierung

rank, **size** size gibt die Gesamtanzahl der an der parallelen Verarbeitung beteiligten Knoten an und rank ist der jeweilige Rang des Knotens. Sofern es nicht anders angegeben wird, ist der Knoten mit Rang 0 automatisch der Root-Knoten. Nachdem mpirun gestartet wurde, können sie durch `mpi.size` und `mpi.rank` die gewünschten Werte auslesen. Beispiel unter der Annahme, dass wir auf drei Knoten arbeiten:

¹Message Passing Interface

```

>>> mpi.size
3
>>> mpi.rank
0
2
1

```

Wie auch bei allen anderen Implementierungen schickt jeder Knoten die Informationen standardmäßig asynchron, d.h. die Rangnummern der drei Beispieldknoten sind nicht zwangsläufig in der richtigen Reihenfolge angegeben.

Broadcast, Barrieren und Reduction Die ersten Elemente zum pyMPI-Nachrichtenaustausch, die ich vorstellen möchte, sind Broadcast, Reduction und Barriere, welche den Gegenstücken aus anderen MPI-Implementierungen völlig entsprechen. Mit Broadcast kann der Benutzer einen bestimmten Datentyp vom root-Knoten an alle anderen Knoten schicken. Der Datentyp kann hierbei einer der Standard-Python-Typen sein, z.B. einfache Variablen, bzw. Werte, Listen oder Dictionaries. Wie bereits weiter oben angesprochen ist standardmäßig der Knoten mit Rang 0 der Root-Knoten, allerdings kann man beim Aufruf des Broadcasts auch einen alternativen root-Knoten angeben. Beispiel:

```

w = mpi.bcast(v)
w = mpi.bcast(v, 2)

```

Die erste Zeile wäre ein Broadcast des Wertes v vom Standardrootknoten an alle anderen Knoten, welche den ankommenden Wert in w speichern. Die zweite Zeile wäre der alternative Aufruf des Broadcasts, wenn der Knoten mit Rang 2 der sendende root-Knoten sein soll.

Das genaue Gegenteil von Broadcast ist Reduction, bei der alle Knoten ihre Werte an den Root-Knoten schicken, welcher die Daten in geeigneter Weise verarbeitet. Auf welche Weise die ankommenden Werte verarbeitet werden, kann mittels eines Operators angegeben werden. Mögliche Operatoren sind:

Name	Operation
BAND	Boolsches AND
BOR	Boolsches OR
BXOR	Boolsches XOR
LAND	Logisches AND
LOR	Logisches OR
LXOR	Logisches XOR
MAX	Maximum
MAXLOC	erstes Maximum
MIN	Minimum
MINLOC	erstes Minimum
PROD	Produkt
SUM	Summe

Als Beispiel nehmen wir hier an, dass alle Knoten ihren Wert, welchen sie in `v` gespeichert haben, an den root-Knoten senden, welcher alle empfangenen Daten aufaddiert und in `w` speichert:

```
w = mpi.reduce(v, mpi.SUM)
w = mpi.reduce(v, mpi.SUM, 2)
```

Die zweite Zeile ist wieder für den Fall, dass der Knoten mit Rang 2 root sein soll und nicht der Knoten mit Rang 0.

Broadcast und Reduction sind zwei Möglichkeiten ein Programm zu synchronisieren. In beiden Fällen geht die Berechnung bzw. die Bearbeitung des Programmes auf jedem Knoten erst dann weiter, wenn alle den Broadcast bzw. die Reduction durchgeführt haben. Eine Möglichkeit zur Synchronisierung ohne Nachrichtenaustausch bilden die Barrieren. Der parameterlose Aufruf der Barriere sieht wie folgt aus:

```
mpi.barrier()
```

Trifft ein Knoten auf die Barriere, so setzt er seine Bearbeitung erst dann fort, wenn alle anderen Knoten ebenfalls auf die Barriere getroffen sind.

blockierendes Senden und Empfangen Das Senden von einem Knoten zum anderen erfolgt durch `send`, wobei die Funktion mindestens zwei Parameter erwartet: die Nachricht und das Ziel. Als dritten und optionalen Parameter kann man einen bestimmten Tag der Nachricht anhängen. Ein Beispiel für das Senden des Inhaltes von `'msg'` an den Knoten 3, einmal ohne Tag und einmal mit Tag 22.

```
mpi.send(msg, 3)
mpi.send(msg, 3, tag=22)
```

Das Empfangen einer Nachricht erfolgt durch die Funktion `recv`, welche bis zu zwei Parameter erhalten kann. Beispiel:

```
msg, status = mpi.recv()
msg, status = mpi.recv(0)
msg, status = mpi.recv(0, 22)
```

Im ersten Fall würde der Knoten jede ankommende Nachricht mit offenen Armen empfangen und die erhaltene Nachricht in `msg` speichern. In `status` wird der Rang des sendenden Knotens sowie der Tag der Nachricht gespeichert. Wird beim Senden kein Tag angegeben, so ist er standardmäßig 0. Im zweiten Fall würde jede Nachricht akzeptiert werden, die vom Knoten 0 kommt und im dritten würde die Nachricht nur empfangen werden, wenn die Nachricht vom Knoten 0 kommt und den Tag 22 hat.

Scatter und Gather Beim Gather sendet jeder Knoten, inklusive dem root-Knoten, den Inhalt seines Sendebuffers an den root-Knoten, welcher die ankommenden Fragmente in der richtigen Reihenfolge zusammensetzt. Scatter ist wieder das genaue Gegenteil zu Gather: hier sendet der root-Knoten die einzelnen Teile des Buffers an die anderen Knoten. Unter der Annahme, dass bei Gather der Inhalt von `v` an den root-Knoten gesendet werden soll, welcher ihn in `w` zusammensetzt, bzw. bei Scatter, wo der Inhalt von `v` zerlegt und in den lokalen Variablen `w` gespeichert werden soll, sehen sie Funktionsaufrufe wie folgt aus:

```
w = mpi.gather(v)
w = mpi.scatter(v)
```

Ausgabe pyMPI stellt eine Funktion namens `synchronizedWrite` zur Verfügung, mit deren Hilfe man eine synchronisierte Ausgabe im Konsolenfenster tätigen kann. Der Clou an der Funktion: die Ausgaben geschehen in der richtigen Rangreihenfolge und nicht zufällig! Beispiel zur Ausgabe der Rang-Nummer bei 3 Knoten:

```
>>> mpi.synchronizedWrite(mpi.rank)
0
1
2
```

4.1.2 Fazit

pyMPI ist derzeit noch in der Entwicklung und befindet seit längerer Zeit in der Beta-Phase. Es sind noch nicht alle Features von MPI-1 und MPI-2 implementiert und es fehlen auch Funktionen von MPI-IO. Auch wenn pyMPI für die Lehre geeignet wäre, da Python an sich sehr leicht zu lesen und zu schreiben ist und viele Unannehmlichkeiten, wie richtige Speicherallokierung in C, entfallen, so wäre pyMPI derzeit für professionelles Hochleistungsrechnen ungeeignet, da es im Schnitt beim Senden um einen Faktor 2 langsamer ist, da viele Nachrichten in zwei Nachrichten zerlegt werden, bevor sie versendet werden. Bei hinreichend großem Nachrichtenaustausch kann dies enorme Leistungseinbußen bedeuten.

4.2 pyPar

pyPar ist auf vielen Clustern erfolgreich getestet worden, darunter MPICH-fähige Maschinen mit Linux, Windows, Darwin oder LAM. Als Voraussetzung, um mit pyPar arbeiten zu können, geben die Entwickler das Vorhandensein der MPI C-Bibliothek, eines C-Compilers, sowie Python ab Version 2.0 an. Um ein pyPar-Programm auf einem Cluster zu starten, sollte man den von den Entwicklern vorgeschlagenen Befehl

```
mpirun -np N python demo.py
```

nutzen, wobei N hier wieder die Anzahl der verwendeten Knoten angibt und demo.py das auszuführende Skript ist.

Die Entwickler geben in ihrer Dokumentation auch die interne Funktionsweise von pyPar an. Danach startet mpirun N (siehe oben) Python-Prozesse, wobei jeder dieser Prozesse pyPar importiert, welches wiederum eine Shared Library importiert, die zu den C-MPI-Bibliotheken verlinkt ist. pyPar setzt nun die angegebenen Befehle für den Nachrichtenaustausch in MPI um, sofern die Aufrufe unterstützt werden.

4.2.1 Programmierung

rank, size und host name Wie bei pyMPI setzt auch die Syntax von pyPar voraus, dass man vor jedem Befehl das Prefix 'pypar.' schreibt. Um die Werte size, die Gesamtanzahl der Knoten, rank, dem Rang des lokalen Knotens und den host-Namen des lokalen Knotens zu erhalten gibt es die folgenden Befehle:

```
>>> pypar.size()
>>> pypar.rank()
>>> pypar.Get_processor_name()
```

Im Gegensatz zu pyMPI sind size und rank hier Funktionen und keine Variablen!

send und recieve Der Befehl für das Senden ist wenig überraschend pypar.send, wobei die Funktion mehrere Parameter erhalten kann:

```
pypar.send(x, d)
pypar.send(x, d, tag=t)
```

Im ersten Fall wird der Datensatz x an den Zielknoten d gesendet, wobei der Standard-Tag verwendet wird. Soll ein anderer Tag verwendet werden, so muss dieser explizit angegeben werden, wie in der zweiten Zeile zu sehen ist. Für das Empfangen der Daten steht der Befehl recieve zur Verfügung:

```
x = receive(s)
x = receive(s, tag=t)
```

Im ersten Fall wird wieder jede Nachricht vom Knoten s empfangen und im zweiten Fall werden lediglich die Daten empfangen, die den richtigen Tag t haben.

Gather, Scatter, Broadcast, Reduce Da ich bereits bei pyMPI die einzelnen Befehle kurz erläutert habe, will ich mich hier lediglich auf die pypar-Syntax der angesprochenen Befehle beschränken:

```
pypar.broadcast(x, root)
pypar.reduce(x, op, root)
pypar.gather(x, root)
pypar.scatter(x, root)
```

Die Funktionsweise der Befehle sind identisch zu denen in C-MPI oder pyMPI. `x` gibt jeweils den entsprechenden Datensatz an, `op` bei `reduce` gibt es den Operator an, mit welchem die erhaltenen Datensätze verknüpft werden sollen und `root` gibt den root-Knoten an. Im Gegensatz zu pyMPI ist ein explizites Angeben des root-Knotens erforderlich, auch wenn der Knoten mit Rang 0 als root eingetragen ist.

weitere Funktionen Zum Schluss möchte ich noch kurz fünf Befehle vorstellen:

```
pypar.time()
pypar.barrier()
pypar.abort()
pypar.finalize()
pypar.initialized()
```

Zur kurzen Erläuterung: `time()` gibt die MPI Wall time zurück, `barrier()` ist identisch zu der im pyMPI, `abort()` bricht alle laufenden Prozesse ab, nach `finalize()` kann keine Nachrichtenübertragung mehr stattfinden und `initialized()` gibt einen bool'schen Wert zurück: `true`, falls MPI initialisiert wurde und `false`, wenn nicht.

4.2.2 effizientere Programmierung

Die im ersten Subkapitel vorgestellten Methoden ähneln sehr den Befehlen und der Syntax von pyMPI, aber leider auch deren Effizienz. Um den Nachrichtenaustausch mit niedriger Latenz betreiben zu können, und damit die Effizienz zu steigern, werden numerische Arrays als vordefinierte Buffer verwendet. Bei pyMPI war der Vorteil ja gerade die einfachere Programmierweise (im Vergleich zu C), weil auf die korrekte Allokierung jedes Buffer-Speichers verzichtet werden konnte. Der Nachteil hingegen war nun, dass pyMPI fast jedes mal eine Nachricht vor der eigentlichen Nachricht schicken musste, in welcher mitgeteilt wurde, wie groß das ankommende Paket ist. Bei pyPar hat man nun die Möglichkeit mit Numerischen Arrays den Zielbuffer vorzugeben und damit wird die vorausseilende Nachricht über die Größe des Nachrichteninhalts überflüssig und man spart sich die Hälfte der Sende-Kommunikation.

Befehle mit Buffern Nehmen wir an wir haben geeignet gewählte numerische Arrays `X` und `A` gegeben, welche, wie auch bei der Programmierung mit C, vorher definiert sein müssen, dann sehen die Befehle für das Senden und empfangen wie folgt aus:

```

pypar.raw_send(A, p, use_buffer=True)
X = pypar.receive(q, buffer=X)
gather(A, root, buffer=X)
scatter(A, root, buffer=X)
reduce(A, op, root, buffer=X)

```

Die Befehle haben die gleiche Bedeutung, wie sie oben erläutert wurden, allerdings verwenden sie nun Buffer. Im Grunde müssen bei Empfangsfunktionen nur das zusätzliche Array mit angegeben werden und im Falle der `raw_send`-Funktion, welche das normale Senden ersetzt, wird gesagt, dass ein Buffer im Zielknoten verwendet wird und somit werden die 'rohen Daten' (raw send) übermittelt ohne vorher zu sagen, wie die zu sendenden Daten aussehen. Dies halbiert, wie oben bereits angesprochen, die Anzahl der Sendeoperationen und steigert die Effizienz der Kommunikation maßgeblich.

4.2.3 Fazit

Mein Fazit für pyPar ist, dass es, wie auch pyMPI, einen leichten Einstieg in parallele Programmierung bietet, dank der Übersichtlichkeit von Python und den einfach gehaltenen Befehlen. Der enorme Vorteil von pyPar im Gegensatz zu pyMPI ist die Möglichkeit zur Verwendung von Buffern, da hierdurch die Effizienz bei parallelen Programmen mit einem großen Nachrichtenaustausch um einiges steigern kann. Allerdings muss hier sichergestellt werden, dass die Buffer korrekt allokiert wurden und dem Programmierer fällt hier die gleiche Verantwortung zu, wie bei MPI in C. Laut der offiziellen Dokumentation ist die Leistungsfähigkeit von pyPar-Programmen sehr nahe an der von MPI-Programmen, welche in C geschrieben wurden. Allerdings bleibt offen, wie klein 'sehr nahe' nun genau ist, da auch für pyPar keine Vergleichsmessungen zwischen MPI in C und pyPar vorliegen.

4.3 Performance ist nicht alles!

In den beiden Subkapiteln zuvor habe ich pyMPI und pyPar unter reinen Performance-Gesichtspunkten betrachtet und kam wie die Entwickler zu dem Schluss, dass die Leistung von Programmen in pyMPI und pyPar der von vergleichbaren Punkten in C unterlegen ist. Aber ungeachtet der Tatsache, dass die Leistung der Python-Varianten der von MPI in C hinterherhinkt finden Skriptsprachen wie Python, Octave und MATLAB immer mehr Einzug in das Hochleistungsrechnen. Der Grund hierfür ist, dass die reine Geschwindigkeit von Programmen nicht mehr das alleinige Kriterium für die Güte eines Programmes ist. Die Mitglieder der High Performance Computing Modernization Program (HPCMP) Community, zu denen auch die Autoren des für diese Seminararbeit verwendeten Papers 'Octave and Python: High-Level Scripting Languages Productivity and Performance Evaluation' stammt, zählen, führen heute vor allem die Produktivität als eine der wichtigsten Kriterien für die Güte eines parallelen

Programmes an. Entscheidend für die Produktivität ist hierbei vor allem die time-to-solution, d.h. die Zeit, welche benötigt wird, um das physische Problem in einen passenden Algorithmus umzuwandeln, das Programm zu debuggen, optimieren und die Ergebnisse zu analysieren und visualisieren.

Und die Produktivität ist genau das, was Entwickler von Skriptsprachen im allgemeinen als große Stärke ihrer Produkte angeben. Obwohl oftmals die Anpreisungen von Entwicklern von der Realität abweisen, scheint sich die Produktivität von Python im Hochleistungsrechnen zu bestätigen. Ein guter Beweis hierfür ist der Sieg der Kombination von Python und Star-P bei der HPC Challenge auf der Supercomputing Conference im vergangenen Jahr in der Kategorie 'Most Productivity'. In dieser Kategorie sind die reine Performance und die Eleganz und Lesbarkeit des Codes zu gleichen Teilen gewichtet.

vom seriellen zum parallelen Programm Python alleine bietet keine Möglichkeit für die Bearbeitung von Datentypen, Matrizen und Linearer Algebra. Doch mit Hilfe des Zusatzmodules numPy werden diese fehlenden Lücken gefüllt und numPy gilt derzeit als Standarderweiterung für numerische Aufgaben. Ist man nun im Besitz eines seriellen Python-Programmes, welches die gewünschte Aufgabe löst und möchte dieses parallelisieren, so kann auf die Erweiterung Star-P zurückgegriffen werden. Star-P bietet parallele Versionen der Funktionen von numPy und kann diese ersetzen, ohne die Funktionalität zu ändern. Dies führt dazu, dass der Entwickler problemlos sein serielles Python-Programm als parallele Version umsetzen kann, ohne sich zu große Gedanken über die parallele Verarbeitung machen zu müssen - dies erledigt Star-P für ihn.

Parallelisierung in Zahlen In einem aktuellen Beitrag im InteractiveSupercomputing-Blog (siehe References) wird die Frage gestellt, wie man am effektivsten ein serielles Skriptsprachenprogramm parallelisieren kann und dabei werden im wesentlichen zwei Möglichkeiten betrachtet: man schreibt das Programm in eine Sprache wie C++ oder FORTRAN um und fügt manuell MPI hinzu oder man nutzt Star-P um das bestehende Programm zu parallelisieren. In dem Beitrag wurde allgemein eine serielle MATLAB-Anwendung angenommen. Im ersten Fall, des Schreibens des Programmes in C++ und MPI wurde ein finanzieller Aufwand von 1.000.000 Dollar geschätzt, wobei 100.000 Dollar auf einen 128-Core-Server entfallen und 900.000 Dollar auf Personalkosten. Zudem wurde eine Entwicklungszeit von 12-18 Monaten veranschlagt. Laut der Schätzung sollte man mit dieser Option eine Leistungssteigerung von einem Faktor von etwa 60 erwarten können, im Vergleich zum seriellen Programm. Die zweite Möglichkeit, das Verwenden von Star-P, wird mit einem Gesamtkostenaufwand von 270.000 Dollar geschätzt, wobei hier 20.000 Dollar für Personal entfallen, 50.000 Dollar für die Star-P-Lizenz und 200.000 Dollar für einen 256-Core-Server aufgewendet werden. Hierbei soll ebenfalls eine Leistungssteigerung von einem Faktor 60 möglich

sein.

Im zuvor kurz erwähnten Paper 'Octave and Python: High-Level Scripting Languages Productivity and Performance Evaluation' einer Arbeitsgruppe des Ohio Supercomputer Centers wird ein Vergleich von Python und Octave im Vergleich zu MATLAB in der Performance auf verschiedenen Systemen durchgeführt und sie kommen in einem verallgemeinerten Benchmark zu dem Ergebnis, dass Python etwas langsamer als MATLAB ist. Dennoch müsste eine vergleichbare Leistungssteigerung zu verzeichnen sein, wenn man ein vorhandenes serielles Python-Programm mit Star-P parallelisiert, anstatt es in C umzusetzen und mit MPI manuell zu parallelisieren, abgesehen davon, dass man viel Personalkosten spart.

4.4 Fazit

In meinem Fazit über die Verwendung von Python im Hochleistungsrechnen komme zu dem Schluss, dass es keine eindeutige Aussage derzeit geben kann, ob man Python nutzen sollte oder doch bei C++ bleiben sollte.

Im wesentlichen hängt die Entscheidung, welche der Möglichkeiten man nutzt, von den eigenen Ressourcen ab. Ist ein serielles Programm in C++ gegeben, welches parallelisiert werden muss, sollte man eindeutig auf MPI zurückgreifen, da C von Haus aus schneller ist. Ist das gegebene serielle Programm hingegen in Python gegeben, so bleibt dem für die Parallisierung Zuständigen die Wahl der Parallelisierung mit pyPar, Star-P oder das Umschreiben in C und manuelle Parallelisieren mit MPI. Bei Projekten mit sehr geringem Budget sollte auf pyPar zugegriffen werden, da dieses kostenlos verfügbar ist und aufwendige Umschreibungen in C und MPI erspart bleiben. Wird eine höhere Leistung erwartet bleibt einem die Wahl zwischen Star-P und dem Umschreiben in C. Bei einem großen Cluster und/oder wenig fachkundigem Personal sollte auf Star-P zurückgegriffen werden. Hier ist zwar der Erwerb einer Lizenz erforderlich, aber die Entwicklungszeit ist minimal. Bei Clustern von geringer Größe erhält man mit Star-P keine hinreichend große Leistungssteigerung und verfügt man über hinreichend viel fachkundiges Personal sollte man eher die höhere Entwicklungskosten in Kauf nehmen, anstatt das bestehende Cluster zu erweitern, zumal die Kosten für zusätzliche Rechner auch zusätzlich höhere Kosten für Strom und Kühlen bedeuten!

Schwer wird die Entscheidung hingegen, wenn vor der Entwicklung eines parallelen Programmes kein serielles Programm zu Grunde liegt und man sich entscheiden muss, ob man sein Projekt in C++ oder Python realisiert. Python bietet zwar eine sehr hohe Produktivität, hat allerdings Einbußen in der Performance, dagegen ist C++ schneller, kann bei komplexen Projekten aber an Übersichtlichkeit verlieren, was bei Python aufgrund der Syntax nicht im gle-

ichen Maße möglich ist. Letztlich kann keine allgemeingültige Aussage getroffen werden, da es auch von dem zur Verfügung stehenden Cluster und dem Personal abhängt. Ist das Cluster sehr stark ausgelastet und Rechenzeit wertvoll und/oder Personal vorhanden, welches hinreichend Erfahrung in C++ hat, sollte man bei C++ bleiben. Beginnt man erst mit der parallelen Arbeit und hat einen großen Cluster zur Verfügung kann an den neuen Trend von parallelem Python angeknüpft werden.

References

- [1] ore Python programming, Wesley J. Chun
- [2] [ttp://pympi.sourceforge.net](http://pympi.sourceforge.net)
- [3] [ttp://datamining.anu.edu.au/~ole/pypar/](http://datamining.anu.edu.au/~ole/pypar/)
- [4] ctave and Python: High-Level Scripting Languages Productivity and Performance Evaluation, Ohio Supercomputer Center
- [5] [ttp://blog.interactivesupercomputing.com/StarP_Blog/tabid/8706/Tag/Python/Default.aspx](http://blog.interactivesupercomputing.com/StarP_Blog/tabid/8706/Tag/Python/Default.aspx)
- [6] [ttp://blog.interactivesupercomputing.com/starp_blog/tabid/8706/bid/3526/What-is-the-Cost-of-Performance.aspx](http://blog.interactivesupercomputing.com/starp_blog/tabid/8706/bid/3526/What-is-the-Cost-of-Performance.aspx)
- [7] [ttp://pytut.infogami.com/node1.html](http://pytut.infogami.com/node1.html)
- [8] [ttp://de.wikipedia.org/wiki/Python_%28Programmiersprache%29](http://de.wikipedia.org/wiki/Python_%28Programmiersprache%29)
- [9] [ttp://de.wikipedia.org/wiki/Guido_van_Rossum](http://de.wikipedia.org/wiki/Guido_van_Rossum)