hpc Documentation

Release 1.0

Stefan Reiterer

February 28, 2011

CONTENTS

CHAPTER

ONE

A BRIEF INTRODUCTION TO PYTHON (FOR MATHEMATICIANS)

Contents:

1.1 Introduction and personal thoughts

If you want to start with Python right ahead follow go directly to *Python Basics*

1.1.1 Foreword

This short documentation was written by me for the High Performance Computing Seminar in the winter semester 2010/11 of Prof. G. Haase at the University Graz Austria.

In order to learn Sphinx and to make it possible for others to get a quick start with Python in mathematics and scientific computing I started to write this tech report.

I started with Python last summer, after a short introduction to the *Sage* mathematics software. One could say it was love at first sight. I was implementing some test code with krylov methods in Matlab and Octave that time, and was annoyed by the lack of object oriented features like abstracting and capsuling (and no: I don't count structs as objects!). I had the problem, that every time I implement a new numerical scheme I had to rewrite the code of my optimisations algorithms, or at least have to alter it, so that every time I need to retest the new implemtation, which costs time and nerves. And since I'm a lazy person I didn't want to do that.

I'm now implementing my thesis code in Python, and I also work as a Sage developer in my freetime, and try to help improving the numerics, optimisations and symbolics parts of Sage which are my personal research interests.

The Python version used here is Python 2.6. since most of the current packages are not ported to Python 3. But I try to make them compatible with the new version so that this document don't get outdated soon.

This document is aimed towards mathematicians who want learn Python in a quick way to use it. I have to admit that I'm a beginner too, and would be happy about feedback on this document. I plan to extend it frequently with tricks I learned and collected on various newsgroups. I don't know If this is a good guide, but I hope potential readers have the same fun reading, I had writing.

And please don't take they sidehits I make too seriously. (And don't send hatemail on some Java comments!) People have different opinions and experiences, either with food, cars or programming languages. I only tend to say it more directly what I feel than other people do. =)

I distribute this under an open license so people can share it freely. (See Section *Licenses*). You are allowed to use and copy contents as you wish.

I would like to thank Stefan Fürtinger and Andreas Niederl for giving me advice and feedback, which I really needed to write this document.

Stefan Reiterer, Graz Austria 2011

1.1.2 Experiences I want to share with programming beginners

My professors in the basic programming and informatics lectures were all software devolopers, and often cursed programmers from scientific areas, because of their complicated and often weird codes. I didn't understand their words that time, but since I'm working with libraries like BLAS, LAPACK, ATLAS etc. I started to understand...

It's true that the processes of software engeneering for "normal" applications and scientific computation are two different areas, but I realised in the recent years that many people from the latter area seem to simply ignore **nearly all** basic concepts of software design and coding, and I don't know why. Maybe it's ignorance, because many think they don't need that much programming again, or because they are simply lazy. Another reason could be that they are too deep into it, and think everyone else think the same way. Perhaps it has historical reasons, like in the case of BLAS, or it's me because of my friends and education I have a different viewpoint on that things.

Neverteless I want to use this section to give some important lectures to people, who aren't deep into programming. I list here some things I learned during the last 13 years since I'm started "programming" Visual Basic with the age of 13.

The Zen of Python don't apply only to Python

If you type into your Python Interpretor the line

```
import this
```

You will get this:

The Zen of Python, by Tim Peters

1. *Beautiful is better than ugly.*
2. *Explicit is better than implicit.*
3. *Simple is better than complex.*
4. *Complex is better than complicated.*
5. *Flat is better than nested.*
6. *Sparse is better than dense.*
7. *Readability counts.*
8. *Special cases aren't special enough to break the rules.*
9. *Although practicality beats purity.*
10. *Errors should never pass silently.*
11. *Unless explicitly silenced.*
12. *In the face of ambiguity, refuse the temptation to guess.*
13. *There should be one– and preferably only one –obvious way to do it.*
14. *Although that way may not be obvious at first unless you're Dutch.*
15. *Now is better than never.*

16. *Although never is often better than *right* now.*
17. *If the implementation is hard to explain, it's a bad idea.*
18. *If the implementation is easy to explain, it may be a good idea.*
19. *Namespaces are one honking great idea – let's do more of those!*

This is the philosophy of Python and can argue about some the points, e.g. point 13., but I can say without bad feeling, that **every** programmer should especially keep in mind the points 1-7, which apply in every language you will use!

Code is more often read than written

For every time code is written, it is read about 10 times, and five times by yourself! If you write code use good and intuitive names of the variables you use, and make enough comments in your code. One often writes code, and then have to look at it a month later, and if you didn't a good work on naming and commenting, you will spend many ours on trying to understand what you have done that time. And remember: Its **your** time. So don't do it unless you want to assure your employment. And if you want to use short variables like A for a matrix make sure to mention that at the beginning of a function which uses these variables. And rest assured: Using longer variable names don't cost performance.

Program design isn't a waste of time!

Of course you don't need to design every snippet of code you do, but at least take your time to think about the implementation, and how you can eventually reuse it. Sometimes ten minutes of thinking can save yourself ours of programming.

Object oriented programming abstracts away your problems

If one is not familiar with the paradigm of object oriented programming change this! There are tons of books and websites on this topic. OO programming is not a trend of the last decades, it's the way of abstract mathematics itself. Mathematicians don't study special cases all the time. We try to exctract the very essence of a class of problems, and build a theory only using these fundamental properties. This makes it possible to use theorems on huge classes of problem and not only on one.

Carefully done this saves yourself alot of programming time, because now you are able to program your algorithms not only for some special input, but for a whole class of objects in the literal sense.

This semester I gave also an excercise in the optimisation course, where all the linesearch methods we implemented had to be integrated into one steepest descent algorithm. While my students needed ours to implement this in Matlab. I only needed one half in Python, because I simply subdivided the sub problems in classes, and had to write the framework algorithm only once.

Modularity counts

Keep the structure of your programs as modular as possible! **Every function should only do exactly one job, and don't use global variables**. If you have to use global variables, then in 90% of the cases something is wrong with your design of the code! Sounds annoying? I was annoyed by that too in my first programming course. But trust me it will help you a lot. At least if you want to reuse a piece of code, or even worse, someone else wants to use your code, you will run into troubles, if you don't have a good organisation of your code. **Remember:** If you have a lot of parameters, you can always store them in a container or a class.

Premature optimisation is the root of all evil!

This often cited quote of Donald E. Knuth [1] is true in it's very deep essence. In an everage program there are about only 3% of critical code. But many programmers invest their time to optimise the other 97% and wonder why their program isn't getting quicker. The only gain you get is a whole bunch of unreadible code. I remember that I implemented an "optimized" for loop some time ago, and the only gain were 3 ms of more speed. And later when I looked on that function I had no Idea what I did that time...

Choice of the right tools

Since I descend a family of craftsmans, this was taught me very early. You don't want to use a sledgehammer for hitting a tiny nail into a wall, and you don't want to use small axe to cut down a tree. (Well I know people who do...). And this applies for programming as well. You don"t want to write a parser in Fortran, and you don't want to write a program for symbolic manipulation in Java. (I personally would never implement *anything* mathematical in Java, because it lacks some aspects like operator overloading and efficiency, but that's only a biased opinion.) The right choice of used languages and tools, can heavily affect the time you need, and also your success of your projects. It often helps to ask colleagues, teachers and Google to find the right tool. I list some of the tools I use here. Keep always in mind that the choice of your tools, depends also on your personal skills, and preferences. Something that a colleague of yours like, could possible a nuissance for yourself.

Don't use Notepad as your editor!

A good editor is not expensive (often even free), and saves you a whole lot of work! Good editors are for example Emacs [2], (to get your Emacs working with Python I recommend this link [3]) VIM [4]. A good List of editors can be found on Wikipedia. [5]

Use version control

Many, many people simply don't know there are very nice tools to keep record of your changes, and make it possible to redo the changes. Most common are Git [6], Mercurial [7] (which is written in Python), or SVN [8].

Use debugging tools

Very good debugging tools are for example Valgrind [9], GDB [10], and many, many more... [11] Python is shipped with it's own debugger [12].

[1] http://en.wikiquote.org/wiki/Donald_Knuth
[2] http://www.gnu.org/software/emacs/
[3] http://hide1713.wordpress.com/2009/01/30/setup-perfect-python-environment-in-emacs/
[4] http://www.vim.org/
[5] http://en.wikipedia.org/wiki/List_of_text_editors
[6] http://git-scm.com/
[7] http://mercurial.selenic.com/
[8] http://subversion.apache.org/
[9] http://valgrind.org/
[10] http://www.gnu.org/software/gdb/
[11] http://en.wikipedia.org/wiki/Debugger
[12] http://docs.python.org/library/pdb.html

Use Linux

This is of course only a personal recommondation. But Linux is in my opinion better suited as development enviroment, because most things you need for programming are native, or already integrated, and even the standard editors know syntax highlighting of the most programming languages. Even C# is well integrated in Linux nowadays, and many useful programming tools are simply not available in Windows (including many of the things we use here). You don't even need to install a whole Linux distribution. Recently there was a huge development of free Virtual Machines like Virtual Box [13], or projects like Wubi [14]. And thanks to Distributions like Ubuntu [15] and it's many derivatives (I use Kubuntu), or open SUSE [16] using Linux is nowadays possible for normal humans too.

Note: Be aware that I assume in this guide, that you are using Linux!

Not everything from Extreme Programming is that bad

It is shown in many tests that applying the whole concept of XP [17], simply doesn't work in practice. However, done with some moderation the basic concepts of extreme programing can make the life of a programmer much easier. I personally use this modified subset of rules:

- The project is divided into iterations.
- Iteration planning starts each iteration.
- Pair programming (at least sometimes).
- Simplicity.
- Create spike solutions to reduce risk.
- All code must have unit tests.
- All code must pass all unit tests before it can be released/integrated.
- When a bug is found tests are created.

Examples say more than thousend words!

Make heavy use of examples. They are a quick reference, and you can use them for testing your code as well.

If your programs aren't understandable nobody will use them

...including yourself.

Use your brain!

Implicitely used in all points above, this is the most fundamental thing. Never simply apply concepts or techniques without thinking about the consequences, or if they are suited for your problems. And yes I include my guidelines here as well. I met many programmers and software developers, which studied software design, and how to use design tools, but never really think about the basics. Many bad design decisions were decided this way!

I also often hear about totally awesome newly discovered concepts, which I use in my daily basis, because I simply don't want to do unessecary work.

[13] http://www.virtualbox.org/
[14] http://www.ubuntu.com/desktop/get-ubuntu/windows-installer
[15] http://www.ubuntu.com/
[16] http://www.opensuse.org/de/
[17] http://www.extremeprogramming.org/

Links

1.2 About Python

1.2.1 What is Python

Python is a high level interpreted object oriented (OO) language. It's main field of application is in web design and scripting.

It was invented by Guido VanRossum in the end of the 80's and the begin of the 90's [18]. The name was derived of the *Monty Python's Flying Curcus* show.

In the last five years there was a huge development of mathematical tools and libraries for Python. Actually it seems that there is no particular reason for this one could see it as a phenomen, or as a trend. But in the meantime the currently available Python projects reached now dimension that make them vaiable alternatives for the "classical" mathematical languages like Matlab or Mathematica.

Also there are now some very useful tools for code optimisation available like *Cython*, that makes it possible to compile your Python Code to *C* and make it up to 1000x faster, than normal Python code.

There are several versions of Python interpreters. The interpreter I refer here as Python is *CPython*, the first interpreter. The CPython interpreter is written, as the name says, in C. The reason for this choice is, that many numerical tools are using C bindings, and Cython also works currently only on CPython. There are also several other Python Interpreters like Jython (written in Java), PyPy (written in Python), or IronPython (written in C#) available.

1.2.2 Why Python?

- Intuitive Syntax
- Simple
- An easy to learn language.
- Object oriented.
- Multi paradigm (OO, imperative functional)
- Fast (if used with brains).
- Rapid development.
- A common language, so you will find answers to your problem.
- Many nice tools which makes your life easier (like Sphinx, which I use to write this report)

1.2.3 Get Python

The programs and packages used here are all open source, so they can be obtained freely. Most Linux distributions already come with an Python interpreter, because many scripts in the system are using Python. See also the *Python* project page for further information [19] .

For using Python I personally recommend Linux or a virtual machine with Linux, because it's much easier to install and handle (for my taste). But there is currently a .Net Python under development named IronPython [20]. Not all

[18] http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html
[19] http://www.python.org/
[20] http://ironpython.net/

packages from classical CPython are currently working on IronPython (including NumPy), but there exists IronClad [21] which should make it possible to use these CPython modules in IronPython.

An easy way to obtain Python is to install *Sagemath* [22], which contains many useful packages extensions and packages for mathematics.

Another possibility would be *FEMhub* which is a fork of *Sage* [23] . FEMhub is smaller, but more experimental than Sage, and is aimed only for numerics. Some of the packages I introduce here are are currently outdated in Sage/FEMhub or not available yet. Current Versions are available on my Google code project page [24].

The drawback of these distributions is that they are not available as .deb or .rpm packages. They have to be build from source, and currently only work on Linux and other Unix type systems. But there are precompiled binaries available. (I personally recommand to build it from source because then many optimisation options are applied)

Links

1.3 Python Basics

1.3.1 The "Goodbye World" program.

In the old tradition of the "<insert Language here> for Dummies" books, we start with the "Goodbye World" program.

1. Make a file goodbye_world.py (or what name you like).
2. Open your Editor.
3. Write:

   ```
   print("Goodbye World!")
   ```

4. Execute:

   ```
   python goodbye_world.py
   ```

   and you get the output:

   ```
   Goodbye World!
   ```

Thats all!

Remark: If you use Sage as your Python interpreter, simply start the program with

```
sage goodbye_world.py
```

or

```
sage -python goodbye_world.py
```

Alternatively you can do this directly in the interpreter. #. Open a shell #. Type:

```
python
```

[21] http://code.google.com/p/ironclad/
[22] http://www.sagemath.org/
[23] http://www.femhub.org
[24] http://code.google.com/p/computational-sage/

1. Write:

```
>>> print("Goodbye World")
```

and press enter.

1.3.2 Notes on the syntax

Intendation for organising blocks of codes

Codes of blocks are, unlike other programming languages like C++ not organized with parantheses but with indentation. I.e. it looks like the following:

```
Code outside Block

<statement> <identifier(s)> :
    Code in block 1
    Code in block 1
    ...
    <statement2> <id> :
        Code in block 2
        Code in block 2
        ...

    Code in block 1
    Code in block 1

    <statement3 <id3> :
        Code in block 3

    Code in block 1

Code outside Block
```

This sounds for many confusing at the beginning (including myself), but actually it is not. After writing some code (with a good editor!) one get's used to this very quickly. Try it yourself: After a week or even a month writing code in Python, go back to Matlab or C.

The benefit of this is, that the code is much more readible, and a good programmer makes indentation nevertheless. It's also helpful for debugging: If you make an indentation error the interpreter knows where it happend, if you forget an **end** or an **}** the compiler often points you to a line number anywere in the code.

Important note: You can choose the type of indentation as you wish. One, two, three, four,... 2011 whitespaces, or tabulators. **But** you should never mix whitespaces with tabulators! This will result in an error.

Recommended by most is to use 4 space indentation (This convention is recomended even by the inventor of Python himself [25])

The semicolon

Generally you don't need a semicolon, and often you don't use it. It's usage is for putting more than one statment in a line.For example:

[25] http://www.python.org/dev/peps/pep-0008/

```
1+1; 2+2
```

Identifiers

Identifier naming follows these rules in Python:

- An identifier starts with a letter (lower- or uppercase) or an underscore (_)
- The rest of the identifier can consist of digits (0-9), letters
- (lower- or uppercase), and underscores.

for example

```
_bla
no
bla_bla_2
```

would be valid,

```
2gether
one space
a-b
```

are non valid identifiers.

Python is case sensitive! The identifiers `a` and `A` are not the same!

1.3.3 Assignment of variables

To assign a value to an identifier, we write =:

```
x = 2
```

There is no need to tell Python the data type, because the interpreter does this for you.

One can also simply change the content:

```
>>> x = 2
>>> x
2
>>> x = 3
>>> x
3
```

Don't worry, Python handle the garbage collection for you.

Like in quite all common programming languges, the value which has to be assigned to is on the left site, this means the statement

```
x = x + 1
```

does *first* add one to x and then overwrite x with the new value:

```
>>> x = 3
>>> x = x + 1
>>> x
4
```

1.3.4 Some basic datatpypes

If you need more information on that topic look in the Python documentation [26].

Remark for Sage users Sage uses it's own integers or reals. Lookup the documentation if you need further information.

Boolean values

In Python the following values are considered as false:

- `None`
- `False`
- Zero of every numeric type, i.e. `0`,``0L``,``0.0``,``0j`` etc.
- Empty containers like ", `()`, `[]`
- instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns the integer zero or `bool` value``False``.

All others are true.

Boolean operations

These are the Boolean operations, ordered by ascending priority:

Operation	Result
`x or y`	if *x* is false, then *y*, else *x*
`x and y`	if *x* is false, then *x*, else *y*
`not x`	if *x* is false, then `True`, else `False`

The return truly means return! Examples:

```
>>> 1 and 2
2
>>> 1 or 2
1
>>> not 1
False
```

Numbers

You can represent numbers in many ways:

```
1
```

[26] http://docs.python.org/library/stdtypes.html

is the **integer** one.

```
1.0
```

is the **float** one.

```
1L
```

represents the **long int** one.

There is also a representation for floats with exponential:

```
1e3
```

which is thousand, or complex numbers:

```
1 + 3j
```

You can also create numbers directly, with their type specified:

```
int(5)
long(3)
float(7)
complex(3,4)
```

Arithmetics

Of course you can use your Python interpreter as a calculator. Simply call

```
python
```

and then try for example:

```
>>> 1+1
2
>>> 2*3
6
>>> 3-2
1
>>> 1+1
2
>>> 1-1
0
>>> 2*3
6
```

Division is a little more tricky in Python:

```
>>> 1/2
0
```

What happened here: A division between two integers return an integer, and Python simply returns the floor. So taking negative numbers it works in the other direction:

```
>>> -5/2
-3
```

If you use the // operator than you force floor division:

```
>>> 1.5//3
0.0
```

More on mathematical operations

Here is short table on basic operations:

Operation	Code
$a+b$	a+b
$a-b$	a-b
$a \cdot b$	a*b
a/b	a/b
a^b	a**b
$\lfloor a/b \rfloor$	a//b
$a \mod b$	a%b
$-a$	-a
$+a$	+a
$\lvert a \rvert$	abs(a)
$\overline{a}$	a.conjugate()

Some operations can be called by functions:

```
>>> 2**3
8
>>> pow(2,3)
8
```

Note: In Python one has also the arithmetic assignemnt operators `+=`, `-=`, `*=`, `/=`, `**=`, `//=`, `%=`, which are shortcuts for performing an operation on the variable, and assign the new value to itself. But there is a little difference: While

```
x = x + 1
```

creates a new variable that get the new value and deletes the old, while the += operator does this *in place*, which means the changes are performed on the object itself. (See the Python pitfalls for more on this [27]) This is done due to performance reasons.

In Python one has also the well known bit operations from C or C++ which can be performed on integers.

Operation	Result
`x | y`	bitwise *or* of *x* and *y*
`x ^ y`	bitwise *exclusive or* of *x* and *y*
`x & y`	bitwise *and* of *x* and *y*
`x << n`	*x* shifted left by *n* bits
`x >> n`	*x* shifted right by *n* bits
`~x`	the bits of *x* inverted

[27] http://zephyrfalcon.org/labs/python_pitfalls.html

Container Types

There are several container types in Python

Lists

Lists are the most common container type in Python. To create a list simply write use the rectangular brackets `[,]`:

```
[1,2,3]
```

The value can be accessed via rectangular brackets again:

```
>>> liste = [1,2,3]
>>> liste[0]
1
```

Note that in Python, like in C, one starts with `0` to count. People who are familiar with Matlab will be happy to here that slicing is supported as well:

```
>>> liste[0:2]
[1, 2]
>>> liste[:]
[1, 2, 3]
```

Note that `[k:n]` goes through the indices k to n-1. Negative indices are also allowed. -1 gives back the last element, -2 the element before the last element and so on:

```
>>> liste[-1]
3
>>> liste[-2]
2
```

One can also declare step sizes to go through the indices:

```
>>> liste[0:3:2]
[1, 3]
>>> liste[::2]
[1, 3]
```

To go backwards through a list use as stepsize -1:

```
>>> liste[::-1]
[3, 2, 1]
```

Lists can also contain elements of various types:

```
>>> liste2 = [1, "two", liste]
>>> liste2[2]
[1, 2, 3]
>>> liste2[0]
1
```

The range function helps to create lists:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1,5)
[1, 2, 3, 4]
>>> range(1,5,2)
[1, 3]
```

One can also create lists from other containers like strings with the list function:

```
>>> list("abc")
['a', 'b', 'c']
```

There are several methods that can be used on lists:

- `append` adds an item to a list:

  ```
  >>> liste = range(5)
  >>> liste
  [0, 1, 2, 3, 4]
  >>> liste.append(5)
  >>> liste
  [0, 1, 2, 3, 4, 5]
  ```

- `extend` appends a complete list:

  ```
  >>> liste2 = range(6,9)
  >>> liste.extend(liste2)
  >>> liste
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 6, 7, 8]
  ```

- `insert` inserts an element at a given position:

  ```
  >>> liste.insert(0,9)
  >>> liste
  [9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 6, 7, 8]
  ```

- `remove` removes the first item from the list, whose value is given:

  ```
  >>> liste.remove(9)
  >>> liste
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 6, 7, 8]
  ```

- `pop` removes the item at the given position:

  ```
  >>> liste
  [0, 1, 2, 3, 4, 5, 6, 6, 7, 8]
  >>> liste.pop(7)
  6
  >>> liste
  [0, 1, 2, 3, 4, 5, 6, 7, 8]
  ```

- `index` gives back the index of the first element with the value given:

```
>>> liste
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> liste[2]
2
```

- `count` returns the number how often the element appears in the list:

```
>>> liste.append(8)
>>> liste.count(8)
2
```

- `reverse` Reverse the elements in place:

```
>>> liste.reverse()
>>> liste
[8, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

- `sort` sort the content of the list in place:

```
>>> liste.sort()
>>> liste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 8]
```

Tuples

Tuples can be created via round brackets:

```
coordinate = (1,2)
```

and they can be accessed like lists:

```
>>> coordinate[0]
1
>>> coordinate[0:1]
(1,)
>>> coordinate[0:2]
(1, 2)
```

There is a tuple function too:

```
>>> tuple([1,2])
(1, 2)
```

The main difference between tuples and lists, is that the former are immutable, that means once created you can't change them on runtime anymore:

```
>>> coordinate[1] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Dictionaries

Dictionaries are special containers that take ketwords for access. They are created with curly brackkets, and each keyword is attached to value with `:`:

```
dic = {'one': 1, 'two': 2}
```

one can it access now like a list, but with the keyword instead the position:

```
>>> dic['one']
1
```

Dictionaries are not immutable:

```
>>> dic['one'] = 3
>>> dic['one']
3
```

Sets

There are also sets in Python. Like the real sets, they are not ordered, and every element is contained only once. They are created with the set function:

```
menge = set([1,2])
```

Of course you can't access an element since there is no ordering. But one can make tests on sets. We come to that right now.

Membership test

One can test the membership of elements within containers.

- `in` tests if an element is in the container and returns True or False:

```
>>> liste = range(5)
>>> 5 in liste
False
>>> 4 in liste
True
>>> liste
[0, 1, 2, 3, 4]
```

- `not in` ... well make an educated guess.

Other operations on containers

- `len` returns the length of an container:

```
>>> liste
[0, 1, 2, 3, 4]
>>> len(liste)
5
>>> tupel = tuple(range(4))
>>> len(tupel)
4
```

- `min`, `max` return the minimal or the maximal value of the container:

  ```
  >>> liste
  [0, 1, 2, 3, 4]
  >>> max(liste)
  4
  >>> tupel
  (0, 1, 2, 3)
  >>> min(tupel)
  0
  ```

  Note thate the output depends on the order relation between the objects!

- The + operator can also be performed to concenate two lists (**Note:** `set` does not support this!):

  ```
  >>> liste
  [0, 1, 2, 3, 4]
  >>> liste + liste
  [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
  ```

- The * operator makes copies of the same container and concenate them (**Note:** `set` does not support this!)

  ```
  >>> liste
  [0, 1, 2, 3, 4]
  >>> liste*2
  [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
  >>> tupel
  (0, 1, 2, 3)
  >>> tupel*2
  (0, 1, 2, 3, 0, 1, 2, 3)
  ```

To be more precise: Those operations are performed on so called sequence types. These are containers, with have an ordered structure, which can be addressed via integers (like normal sequences)

Those types are: * strings * unicode strings * lists * tuples * iterators

For more information I refer here to the Python documentation again.

For all non-german speakers who wonder why I took liste and not list: *Liste* means *list* in German, as *tupel* means *tuple* . The benefit of german names is that they are not reserved, because `list` is a function in Python, and one has to delete the list afterwards:

```
del list
```

Since the german expressions are not that different, I hope people will understand anyway.

Strings

Strings are containers too, but they are quite special, so they get their own section here. There are several ways to create strings in Python:

```
a = 'bla'
b = "bla"
c = """bla"""
d = str('bla')
e = '''bla'''
```

The only one of these, which is slightly different is the triple quote '‘ or “”“, which allows multilines and quotes inside the string:

```
string = """Hi! I'm the "best" sting in this Universe.
            You can believe me, there is no better one."""
```

One can also create strings over more lines using the backslash:

```
>>> a = "First \
... Second"
>>> a
'First Second'
```

Note that writing two strings in one command leads to creating only one string:

```
>>> a = "First" " Second"
>>> a
'First Second'
```

Of course strings are objects to so you can call class methods on them.

Note that Strings are immutable in Python, which means that you can't alter it, after you you created it. Like everything this has benefits and drawbacks.

Another important attribute of strings is that they are containers. You can access every element like a vector in Matlab:

```
>>> "hat"[0]
'h'
>>> "hat"[2]
't'
>>> "hat"[0:]
'hat'
>>> "hat"[0:1]
'h'
>>> "hat"[0:2]
'ha'
```

This somehow logical, because every character is simply an object, in a list of characters, which form the string. People who are coming from the C world, will be familiar with this, because in C a string is also a list of chars.

Special types of strings in Python

You can specify some types of strings in Python:

```
r"Newlines are made with \n"
```

This makes a raw string, on which no formating as applied. Capital R works also for this.

We also can create unicode strings with utf8 support:

```
kebap = "Dürüm"
```

This looks like the following in Python:

```
>>> kebap
'D\xc3\xbcr\xc3\xbcm'
>>> print(kebap)
Dürüm
```

Basic manipulation of strings

Two put two strings together one can use the + operator:

```
>>> a = "First"
>>> b = " Second"
>>> a+b
'First Second'
```

Formating like in C is also allowed:

```
>>> a = "First \nSecond"
>>> print(a)
First
Second
```

Note again the difference to the raw string:

```
>>> b = r"First \n Second"
>>> print(b)
First \n Second
```

We can also make replacement statements:

```
>>> breakfast_everyday = "I had %(SPAM)s pieces of spam, and %(EGGS)s eggs for breakfast"
>>> todays_spam = 2
>>> todays_eggs = 3
>>> breakfast_today = breakfast % {'SPAM': todays_spam, 'EGGS': todays_eggs}
>>> print(breakfast_today)
I had 2 pieces of spam, and 3 eggs for breakfast
```

To use the `%` sign in a string you should use a raw string or simply write `%%` for example:

```
print('%(NR)s %%' % {'NR': 100})
```

else you would get an error!

There are other possibilities to replace placeholders:

```
"There are {0} nuns in this castle!".format(5)
"{1} plus {0} is {2}".format(1,2,1+2)
"{ONE} plus 2 is 3".format(ONE=1)
"{numbers[0]} plus {numbers[1]} is  {numbers[2]}".format(numbers=[1,2,3])
```

For further information see the Python documentation on strings [28]

Iterators

An iterator is an object representing a stream of Data, and returns one element at the time. It is also possible to define infinite iterators.

To create iterators one can use the `iter` function:

```
iterator = iter(range(3))
```

There are several datatypes which support iterators. In fact every sequence type supports iterating (even strings).

Every iterator must support a `next` function:

```
>>> iterator = iter(range(3))
>>> iterator.next()
0
>>> iterator.next()
1
>>> iterator.next()
2
>>> iterator.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterators can be converted to lists or tuples:

```
>>> iterator = iter(range(3))
>>> list(iterator)
[0, 1, 2]
>>> iterator = iter(range(3))
>>> tuple(iterator)
(0, 1, 2)
```

With help of the `itertools` module there are several other ways to create iterators. `itertools.count` for example creates an infinte stream of integers.

1.3.5 The print statement

We used it already some times. Here we give further information.

To print a simple string for example write:

```
print("I'm a string!")
```

or without braces:

[28] http://docs.python.org/library/string.html

```
print "I'm a string"
```

(Why did I always write those stupid brackets, when I don't have to? I come back later to that topic.)

We can also print numbers or other datatypes:

```
print(1)
```

In fact every class that holds a __str__, or __repr__ method can be printed. We will come back later to that in the section of :ref: class_ref .

To print more than one thing you can use a comma (`,`)

> print 1, "plus", 2, "is", 1+2

this gives back:

```
1 plus 2 is 3
```

Note that here with use of the brackets we would get:

```
>>> print(1, "plus", 2, "is", 1+2)
(1, 'plus', 2, 'is', 3)
```

To avoid newline, simply add a comma at the end of the statement:

```
print 1, "plus",2, "is",
print 1+2
```

Note: In Python 2.x `print` is a statement, in Python 3 `print` is a function. This is one of the most discussed changes from Python 2 to Python 3 (see for example this famous thread on the Python mailinglist [29] . In order to keep your Code compatible, you can import the print function with:

```
from __future__ import print_function
```

In Python 3 the line

```
print 2
```

would be invalid. One has to use the brackets. This is the reason why I write here all print statements in brackets to make it easier to "port" this document to Python 3.x.

With the print function the statement

```
print(1, "plus", 2, "is", 1+2)
```

would now return

```
1 plus 2 is 3
```

which was to be expected. The trick with the newline, also doesn't work anymore. To get newline at the end you would have to write

```
print(1, end=" ")
```

[29] http://mail.python.org/pipermail/python-list/2010-June/1248174.html

1.3.6 Comparison operators

Here I shortly list the available comparison statments in Python. The syntax should be very familiar to C programmers.

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

Attention: A trap for beginners (including me) is, that the `is` statment, is different from the `==` operator. For example

```
x = 1
x is 1
```

does work correctly, but

```
x = 1.0
x is 1.0
```

does not.

Links

1.4 Programming in Python

In this section I will give an overview of important tools for programming with Python. The sections are ordered by programming paradigms. If you are new to programming, the last paragraph of this section contains an overview of these paradigms. (*Some words on programming paradigms*)

1.4.1 Commenting in Python

To comment out lines of codes use `#`.

Examples:

```
# I'm a comment

x = x + 1 # do some stuff

# bla
# bla
```

1.4.2 Go with the control flow

The `if` statement

The `if` statement in Python is quite the way one would expect from other languages. As mentioned in the section *Intendation for organising blocks of codes* the `if` statement has the following structure:

```
if condition_is_true:
    do_something
```

Note the intendention!

There is also an `else` statement in Python:

```
if condition_is_true:
    do_something
else:
    do_something_else
```

Note that for the `else` statement the intendention rule applies too!

There is also an `elif` clause short for else/if:

```
if condition_is_true:
    do_something
elif another_condition_is_true:
    do_something_different
else:
    do_something_else
```

Here for example we determine the sign of a value:

```
if x > 0:
    sign = 1
elif x < 0:
    sign = -1
else:
    sign = 0
```

`while` loops

`while` loops are also like expected:

```
while condition_is_true:
    do_something
```

In Python while loops know alos an `else` statement. It is executed when the condition is violated:

```
while condition_is_true:
    do_something
else:
    do_something_else
```

Here an example:

```
k = 0
while k < 10:
    print(k)
    k += 1
else:
    # if k >= 10 we come into the else clause
    print("Start")
```

the output of this snippet is:

```
0
1
2
3
4
5
6
7
8
9
Start
```

`for` loops

For loops are a little bit different in Python, because in contrast to other programming languages `for` iterates through an iterator or an type which supports iterating, and not only to integers or numbers, like in C.

A `for` loop looks like this:

```
for x in list:
    do_something_with_x
```

We can use the `range` function (see the section about *Lists*) to create a *normal* `for` loop:

```
for i in range(n):
    do_something_with_x
```

The `for` loop knows also an `else` statement. It is executed when `for` reaches the end of the list/sequence.

Analogous to our `while` example:

```
for k in range(10):
    print(k)
else:
    # When end of list is reached...
    print("Start")
```

Remark: To get out more performance of your Python code use `xrange` instead of range, because `xrange` doesn't need allocate memory for a list. In Python 3, however, `range` returns an iterator and not a list, so this is obsolete then.

See also the Python wiki [30] on this topic.

[30] http://wiki.python.org/moin/PythonSpeed/PerformanceTips

The break and continue statements

The `break` and `continue` statements are borrowed from *C*.

- `continue` continues with the next iteration of the loop. For example:

```
>>> k = 0
>>> for i in range(10):
...     k += i
...     continue # Go on with next iteration
...     print(k) # The interpreter never reaches this line
... else:
...     print(k) # print result
...
45
```

- `break` breaks out of the smallest enclosing `for` or `while` loop. Here a famous example from the official Python tutorial [31]

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

The pass statement

The `pass` statement, in fact, does nothing. It can be used as a placeholder for functions, or classe which have to be implemented yet.

For example the snippet

```
while 1:
    pass
```

results in an endless loop, where nothing happens.

1.4.3 Defining functions

A function is declared with the `def` statement in normal Python manner. The statment has to be followed by an identifier We simply start with a classical example, and give explaination later on.

[31] http://docs.python.org/tutorial/controlflow.html

The factorial would be implemented in Python that way:

```
def my_factorial(n, pochhammer = None):
    """ Your documentation comes here"""

    if pochhammer is None: # Check if evaluate Pochhammer Symbol
        a = n

    k = 1
    for i in xrange(n):
        k *= a - i

    return k # Give back the result
```

The `return` statement

The `return` statement terminate the function and returns the value. To return more values simply use a comma:

```
def f(x,y):
  return 2*x, 3*y
```

Python return them as a tuple:

```
>>> a = f(2,3)
>>> a
(4, 9)
```

If you dont want to store them in a tuple simple use more identifiers seperated by a comma:

```
>>> b,c  = f(2,3)
>>> b
4
>>> c
9
```

`return` without an expression returns `None`

Variables (inside functions)

Variables within a function are all local, except they are defined outside of the code block:

```
>>> x = 1      # declared outside of the function
>>> def f():
...     a = 2    # declared inside of the function
...     print(x) # can be called within the function
...
>>> f()
1
>>> a # not defined outside of the function
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
 But you can't assign values
to a global variable within a function
```

But you can't assign a global varaible a new value within a function:

```
>>> x = 1
>>> def f():
...     x = 2
...     print(x)
...
>>> f()
2
>>> x
1
```

except you use the `global` statement:

```
>>> global Bad    # Declare identifier as global
>>> Bad = 1
>>> def f():
...     global Bad  # Tell the function Bad is global
...     Bad = 2
...     print(Bad)
...
>>> Bad
1
>>> f()
2
>>> Bad
2
```

but I would avoid this as much as possible...

Default values and keyword arguments

Python allows to define functions with default values:

```
>>> def answering(name, mission, answer="I don't know"):
...     print("What iss your name?")
...     print(name)
...     print("What iss your mission?")
...     print(mission)
...     if answer == "I don't know":
...         print(answer + " Ahhhhhhhhh!")
...     else:
...         print(answer)
...         print("You may pass")
...
>>> answering("Gallahad", "The search for the holy grail")
What's your name?
Gallahad
What's your mission?
The search for the holy grail
I dont know Ahhhhhhhhh!
>>> answering("Lancelot", "The search for the holy grail", "Blue")
What's your name?
Lancelot
What's your mission?
The search for the holy grail
```

```
Blue
You may pass
```

You can also call them with keyword arguments:

```
>>> answering("Lancelot", "The search for the holy grail", answer = "Blue")
What's your name?
Lancelot
What's your mission?
The search for the holy grail
Blue
You may pass
```

This can be quite useful. For example you want to define a function, with several options:

```
def f(x,y, offset_x = 0, offset_y = 0):
    return 2*x + 2*y + offset_x - offset_y
```

Now we can call the offset_y variable directly, without setting a value for offset_x:

```
>>> f(0,0,1)
1
>>> f(0,0,offset_y = 1)
-1
```

Important: A non keyword argument cannot follow a keyword argument:

```
>>> f(offset_x = 1,0)
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

This also applies for the definition of the function:

```
>>> def g(y = 1,x):
...     return x + y
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

Calls with lists and dictionaries

A function be called with arbitrary many arguments using the * symbol:

```
>>> def sum_up(offset = 0, *summands):
...     k = offset
...     for x in summands:
...         k += x
...     return k
...
>>> sum_up(1)
1
>>> sum_up(1,2)
3
>>> sum_up(1,2,3)
6
```

```
>>> sum_up(1,2,3,4)
10
```

What happens here? Python wraps all additional arguments into a tuple, which is identified with the keywords after the *. Very often as convention *args is used.

One also can use different types of keywords, and surpass them as dictionary. Here again an example from the Python documentation:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments:
        print arg
    print "-" * 40
    keys = sorted(keywords.keys())
    for kw in keys:
        print kw, ":", keywords[kw]
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Be aware that **name has to come after *name (if there is one).

Remark The * operator can be used to unpack contents of a list and give them to a function as well:

```
>>> def f(x,y):
...     return x+y
...
>>> liste = [1,2]
>>> f(*liste)
3
```

Docstrings

Docstrings are optional, and come right after the definition of the function. A docstring is simply a string. Here is an example:

```
>>> def doubling(x):
...    """I'm doubling stuff!
...       Yes it's true!"""
...    return 2*x
...
>>> print doubling.__doc__
I'm doubling stuff!
     Yes it's true!
```

There are many powerful tools like Sphinx, where you can use your docstrings for creating documentation of your code, or tools for automatic testing, which read take the docstring as input.

Other ways to define functions

There are also some other ways to define functions in Python. One would to be write them in one line, and seperate the different operations with a semicolon:

```
>>> def f(x): y = 2*x; return y
...
>>> f(2)
4
```

Another way is the λ statement:

```
f = lambda x: 2*x
```

One key difference is, that $lambda$ has no return statement, and it can contain only one expression.

In fact lambda returns a function, and is only syntactic sugar, but it often is very handy.

But lambda can take more than one variable:

```
lambda x,y: x + y
```

Note: In older versions of Python 2 lambda can unpack tuples:

```
lambda (x,y): x + y
```

is valid in older versions of Python 2, but not in Python 2.6! In Python 2.6 or above one has to write

```
lambda xy_tuple: xy_tuple[0] + xy_tuple[1]
```

or

```
lambda x,y: x + y
```

instead.

The λ statement is confusing many people. Guido Van Rossum himself wanted to remove the λ statement from Python 3, but didn't succeed to find a good replacement [32] . As one of it's biggest fans I can only say: Hooray for λ !

The reason for the strange naming is that in the early times of Python, many *Lisp* programmers wanted some functional features from *Lisp*, and one of the was λ . But it's true origin comes from the λ calculus [33] .

[32] http://mail.python.org/pipermail/python-dev/2006-February/060415.html
[33] http://en.wikipedia.org/wiki/Lambda_calculus

I prefer lambda for some reasons, especially that I can use it inline. But I wouldn't recommend to use λ every time, sometimes the use of the lambda statement is not good readable.

One may argue that using λ too often creates unreadable code. But on the other hand, it has the benefit, that the actual action is written right there where it is excecuted, and that can be used to avoid unnecessary comments, especially if you defined the action several lines before.

In my work I often have to deal with several mathematical operations. And yes, I prefer it to write:

```
lambda x,y,z: x**2*y**3 + z**4
```

over

```
def square_x_mlt_y_to_pwr_3_add_z_to_pwr_4(x,y,z):
    return x**2*y**3 + z**4
```

Of course one can also use shorter names like `help_func1`, or `help_func2` .. and forget which one does what... but you also can overwrite it again, and again.... and break something in an other part of your code.

Just my 2 cents.

1.4.4 Functional Programming tools in Python (or hooray for λ)

In some sense I'm relaively new to functional programming myself, in somse sense not, since I use it hidden in some mathemtaical languages like Mathematica or Matlab.

Functional programming can be a very powerful tool, and I show here some of the key features for functional programming in Python. I follow here the programming guide for functional programming in Python [34]. For more advanced techniques and more founded Background on that topic I refer to the *Python Functional Programming HOWTO* [35]

The `map` function

The `map function` takes any number on iterables and a function, and apply the function on the iterables. That means:

```
map(f, iter1, iter2,...)
```

returns

```
[f(iter1[0],iter2[0],..), f(iter1[1],iter2[1],...)...]
```

For example:

```
>>> liste = range(3)
>>> def f(x): return 2*x
...
>>> map(f,liste)
[0, 2, 4]
>>> def g(x,y): return x*y
...
>>> map(g,liste,liste)
[0, 1, 4]
```

[34] http://programming-guides.com/python/functional-programming
[35] http://docs.python.org/howto/functional.html

This can be very useful for vectorized operations. Here again the lambda statement comes in handy:

```
map(lambda x: 2*x, liste)
```

returns again

```
[0, 2, 4]
```

The `reduce` function

Reuce takes as input a list, a function and as optional value an initial value. Reduce do now the following: It takes the first elements of the list, and apply the function to it, than it applies the function to the result and the next element in the list, and again and again... and returns an value. If an initial value is given this is taken as the first value. This menas now in expressions that

```
reduce(f, liste)
```

is evaluated as

> ..f(f(f(liste[0],liste[1]),liste[2]),liste[3])..

or

:: reduce (f,liste,init)

to

```
..f(f(f(init,liste[0]).liste[1]),liste[2])..
```

Let's calculate the factorial 10!

```
reduce(lambda x,y: x*y, range(1,10))
```

or the doubled factorial:

```
reduce(lambda x,y: x*y, range(1,10),2)
```

Note In Python 3 reduce was moved to the `functools` module. It can be backimported via:

```
from functools import reduce
```

The `filter` function

An important tool to select elements from a list is the filter function. `filter` takes a function and a list and returns, the list of elements for which the function returned true:

```
def Is_even(x):
  return (x % 2) == 0

filter(Is_even,range(10))
```

returns now:

```
[0, 2, 4, 6, 8]
```

Generators

Generators are like functions, but they give back a sequence of data instead of a single output. They can be used to write iterators.

To create an generator, simply write a function with the `def` statement, but instead of using `return` use `yield`.

For example

```
def generate_even(N):
    for n in range(N):
        yield 2*n
```

gives back an iterator witch contains all even numbers.

If we now create an iterator we can do all things which we know:

```
>>> iterator = generate_even(5)
>>> list(iterator)
[0, 2, 4, 6, 8]
```

One key difference between generators and functions is, that while in a function call, all local variables are created once and are destroyed after `return` was called. The varaibles in an generator stay. You can call `return` within an generator as well, but without output, and after it is called the generator cannot produce further output.

As an example we write a little program which factors an integer number with help of functional tools:

```
from __future__ import print_function

def find_factors(num):
  """
  Find prime factors of a number iterativly
  """
  # we take advantage of the fact that (i +1)**2 = i**2 + 2*i +1
  i, sqi = 1, 1
  while sqi <= num+1:
      sqi += 2*i + 1
      i += 1
      k = 0
      while not num % i:
          num /= i
          k += 1

      yield i,k

def print_factors(num_fac):
    if num_fac[1] > 1:
        print(str(num_fac[0]) + "**" + str(num_fac[1]),end = " ")
    else:
        print(num_fac[0],end=" ")

def factorise(num):
    """
    Find prime factors and print them
    """

    factor_list = list(find_factors(num))
    def get_power(pair): return pair[1]
    factor_list = filter(get_power, factor_list)
```

```
    if len(factor_list) is 1 and (factor_list[0])[0] is 1:
        print("PRIME")
    else:
        print(num, end=" ")
        map(print_factors, factor_list)
        print("")
```

List Comprehensions

List comprehensions are often a good alternative to `map`, `filter` and `lambda`. A list comprehension consists of an expression followed by an `for` clause, which are followed by zero or more `for` and/or `if` clauses. The whole thing is surrounded by rectangular brackets.

Examples:

```
>>> vector = range(0,10,2)
>>> [3*x for x in vector]
[0, 6, 12, 18, 24]
>>> [2*x for x in vector if x > 3]
[8, 12, 16]
>>> vector1 = range(3)
>>> vector2 = range(0,6,2)
>>> [x*y for x in vector1 for y in vector2] # Goes through all combinations
[0, 0, 0, 0, 2, 4, 0, 4, 8]
>>> [vector1[i]*vector2[i] for i in range(len(vector1))] # mimic map
[0, 2, 8]
>>> map(lambda x,y: x*y,vector1,vector2) #equivalent statement
[0, 2, 8]
```

List comprehensions can also be applied to much more complex expressions, and nested functions.

1.4.5 Objects and classes

Classes are the basis of every OO language, and Python is no exception.

Definition of classes and basic properties

Classes look quite similar to functions:

```
class class_name:
    <statement1>
    <statement2>
    .
    .
    .
```

We use here complex numbers as an example:

```
class my_complex:
    """ Complex numbers as example"""
    nr_instances = 0 # This belongs to the whole class

    def __init__(self,re,im):
```

```
        """The init method serves as constructor"""

        self.re = re
        self.im = im

        my_complex.nr_instances += 1

    def abs(self):
        """Calculates the absolute value"""
        return self.re**2 + self.im**2
```

What do we have here. First let's look into the `__init__` method, which is the constructor of an object. The first element is the object itself. Every function (method) of the class takes itself as first input parameter. The name `self` is only a convention, one can use every other identifier. **Important**: `self` has to be the first argument in every class method, even when it is not needed!

So what does our constructor here:

- First the object gets it real and imaginary part, simply by setting this class member. In Python the object can be created simply by:

  ```
  >>> a = my_complex(2,3)
  ```

  As seen in the constructor we simply added a new class member to the object, and in fact, one can always add new class members as he/she wishes:

  ```
  >>> a.new = 1
  >>> a.new
  1
  ```

- The last statement simply adds one to the counter, which counts the number of instances. We defined it in the beginning of the class before the `__init__` function. This counter belongs to the whole class, that's the reason why we had to call it with `my_complex.nr_instances`. And indeed the counter is global for our class:

  ```
  >>> a.nr_instances
  1
  >>> b = my_complex(3,4)
  >>> a.nr_instances
  2
  >>> b.nr_instances
  2
  ```

The next thing we defined is a class method, in this case the (squared) absolute value. After creating an instance, we can call it simply like that:

```
>>> a.abs()
13
```

Huh what happened to the `self`? The answer is Python takes the self argument as default, so you don't have to type it anymore.

Deriving classes from other classes and overloading of methods

This is rather easy in Python. To tell the interpreter from which class he should derive the new class, simple put it into round brackets. To to overload a certain function simply add it again.

Let's go back to our complex number example. It annoys us, that the absolute value is squared, but we don't want a new constructor. So we simply derive The old complex class, and overload the absolute value:

```
class my_new_complex(my_complex):

    def abs(self):
        """Calculates the absolute value"""
        return (self.re**2 + self.im**2)**0.5
```

What does Python internally? After it checked which functions are already defined in the new class it adds all members from the old. With that logic one also can inherit from multiple base classes. After checking what's in the new class, it looks what is in the first class given, then in the second, and so on. Consider the for example in the class

```
new_class(base1,base2,base3)
    pass
```

the priority order for looking up new methods is new_class-> base1 -> base2 -> base3 and not new_class -> base3 -> base2 -> base1.

Operator overloading

In my opinion one of the most powerful features in OO languages, and the reason why I think Java isn't worth to look at it.

Especially in mathematics one wants to define new algebras or objects with algebraic operations, to make programs more readable and algorithms reusable.

In order to overload operators in Python classes one has only to add the right methods. Now let's add +, * operations to our complex number class:

```
class my_nice_complex(my_new_complex):

    def __add__(self,other):
        return my_nice_complex(self.re + other.re,self.im + other.im)

    def __mul__(self,other):
        return my_nice_complex(self.re*other.re - self.im*other.im,
                               self.re*other.im + self.im*other.re)
```

The `__add__` and `__mul__` functions return new objects of the complex class. The good thing is we can use the normal + and * operators:

```
>>> a = my_nice_complex(3,4)
>>> b = my_nice_complex(5,7)
>>> c = a + b
>>> c.re
8
>>> c.im
11
```

One can also add additional features like a string representation, that `print` is able to return. Lets add a `__repr__` method to the class:

```
class my_nice_complex(my_new_complex):

    def __add__(self,other):
```

```
        return my_nice_complex(self.re + other.re,self.im + other.im)

    def __mul__(self,other):
        return my_nice_complex(self.re*other.re - self.im*other.im,
                               self.re*other.im + self.im*other.re)

    def __repr__(self):
        return "{0} + {1}i".format(self.re,self.im)
```

Now we can print our complex class:

```
>>> a = my_nice_complex(3,4)
>>> print(a)
3 + 4i
```

There is a whole bunch of features that can be added to a class. I refer here to the Python reference manual for a complete list [36], because listing them all here would be too long.

1.4.6 Exceptions

What is an exception? An exception is a special object (yes exceptions are objects too!) which to tell the interpreter that something happended, which shouldn't have (or sometimes it is expected), then the interpreter tells you that it caught an exception. Of course it is possible to tell the interpreter what to do when an exception arises. This allows many advanced possibilites for the programmer.

Python has many builtin exceptions like out of range exceptions division by zero exceptions and so on.

Exceptions are a powerful tool in programming languages to find errors, and provide a safe workflow. Exceptions can also be used for control flow. In fact handling exceptions can yield better performance, than many `if` statements, because the interpreter checks *many* `if` s but only has to wait for *one* exception.

Handling exceptions

To catch exceptions us the `try` and `except` statements:

```
try:
    1/0
except ZeroDivisionError:
    print("I don't think so, Tim.")
```

What happens here? The try statement excecutes the following codeblock. If an exception of the type ZeroDivisionError arises it executes the code block after the except statement. Of course one can handle sever different exception types. Only add more except statments:

```
try:
    do_something
except exception_type1:
    do_that
except exception_type2, and_exception_type3:
    do_this
.
.
.
```

[36] http://docs.python.org/reference/datamodel.html#special-method-names

With help of the `raise` can also force the program to throw exceptions. For example

```
>>> raise Exception('spam','eggs')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: ('spam', 'eggs')
```

This is important to throw the correct exceptions of certain types, with user defined error messages. This makes debugging a lot easier!

There is another possibility in Python: So called clean up actions, which have to executed at all costs (for example cleaning up allocated memory). Those can be specified via the `finally` statement:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print("I don't think so, Tim.")
...
I don't think so, Tim.
KeyboardInterrupt
```

Here a more advanced example for exception handling: Let's remember our prime factor example from the *Generators* section. We want that the function should only handle integers, so we check this with help of exceptions:

```
from __future__ import print_function

def find_factors(num):
  """
  Find prime factors of a number iterativly
  """

  # we take advantage of the fact that (i +1)**2 = i**2 + 2*i +1
  i, sqi = 1, 1
  while sqi <= num+1:
      sqi += 2*i + 1
      i += 1
      k = 0
      while not num % i:
          num /= i
          k += 1

      yield i,k

def print_factors(num_fac):
    if num_fac[1] > 1:
        print(str(num_fac[0]) + "**" + str(num_fac[1]),end = " ")
    else:
        print(num_fac[0],end=" ")

def factorise(value):
    """
    Find prime factors and print them
    """

    try:                                #check if num is an integer
        num = int(value)                #with exceptions
        if num != float(value):
            raise ValueError
```

```
    except (ValueError, TypeError):
        raise ValueError("Can only factorise an integer")

    factor_list = list(find_factors(num))
    def get_power(pair): return pair[1]
    factor_list = filter(get_power, factor_list)

    if len(factor_list) is 1 and (factor_list[0])[0] is 1:
        print("PRIME")
    else:
        print(num, end=" ")
        map(print_factors, factor_list)
        print("")
```

Compare this to the last programming example of this page [37] , which is an imperative solution.

For further information on Exceptions see the Python documentation [38]

Creating new exceptions

Since Exceptions are classes too, they can be simply created by deriving them from the `Exception` base class:

```
class ToolTimeException(Exception):
    def __init__(self, stupid_comment):
        self.stupid_comment

    def __str__(self):
        print("\n" + self.stupid_comment +  "\nI don't think so, Tim")
```

Then you can normally raise it:

```
>>> raise ToolTimeException("And if you're painting Al's mom, you can \
... get it done in a matter of years." )
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.ToolTimeException
And if you're painting Al's mom, you can get it done in a matter of years.
I don't think so, Tim
```

1.4.7 Modules and Packages

Of course no one wants to type everything into the interpreter all the time, but safe the programs into files and execute them by calling a function. We learn how to do this in Python.

Modules

We alredy dealt indirectly with modules. Modules are several functions, classes etc. stored in a file with a `.py` suffix, like we did in the *Goodbye World* example.

For example lets write a a file with some functions in it:

[37] http://pleac.sourceforge.net/pleac_python/numbers.html
[38] http://docs.python.org/tutorial/errors.html

```
def square(x):
    return x**2

def cube(x):
    return x**3
```

Now save them in a file. Let's say `powers.py`.

Now you can import it into Python with the `import` statement:

```
import powers
```

From that on, you can call at's functions:

```
>>> powers.square(3)
9
```

We called the square function like a class member, and in fact a module is a *class*.

If one don't want to import a part of a module directly, one can use the `from...import` statement:

```
from powers import cube
```

Now cube can be called directly:

```
>>> cube(3)
27
```

This is also a great benefit over Matlab: You can do as many functions as you want into one file.

Packages

To construct trees of modules we can organise them in packages. To make a package do the following: Save all modules that should belong to the package into a directory with the name of the package. Then add an (most times empty) file named `__init__.py` to the folder. For example we want our power module into an math_stuff package which also holds an module for roots of several powers. First we make a directory `math_stuff`

So we write that module:

```
def sqrt(x):
    return x**0.5

def curt(x):
    return x**(1./3.)
```

and save it to a file `roots.py` in the `math_stuff` directory. Then we create an empty file `__init__.py` in that folder.

Important make sure to be in the right working directory! There are several possiblities to do that:

- `cd` to your directory in a shell and call Python there. Then the current directory is also your working directory.
- In Python, you can achieve that by using the `chdir` function from the `os` module:

```
>>> from os import chdir
>>> chdir("/the/folder/math_stuff_is_in/")
```

- In IPython or Sage simply use the command cd in the interpreter.

Now you can normally import the powers module by:

```
>>> import math_stuff.powers
```

and call it's functions:

```
>>> math_stuff.powers.square(4)
16
```

To build subpackages one only has to create a subfolder with the name of the subpackage and put an `__init__.py` file into a that subfolder, and so on.

There are several more things one can do, for example make it possible to import the complete namespace with `*`:

```
>>> from os import *
```

Now you can use every function and submodule of the `os` package, without typing `os.whatever`. I personally don't recommand that because of two reasons:

- If you load to much modules, which have quite similar functions (for example every math packages has it's `sin` function, then you can run into troubles.
- It yields better performance. The more functions and modules are loaded the more load has the interpreter to deal with.

I recommend personally to import explicitly with `from ... import` only the functions you actually need.

For further information see the Python documention [39].

1.4.8 Reading and Writing external files

To read or writing external files you first have to open it. We do this with the open function: `open(filename,mode)`. The modes are `'r'` (**r** ead only), `'w'` (**w** rite only; a file with the same name will be deleted), `'a'` (**a** ppends data to the end of the file), `'r+'` (read and write). Default mode is `'r'`. Open returns a object of the type FileObject.

You find more information at the Python documentation [40], [41]

Examples: We write some text to a file, named test_file.txt and store it in the working directory, containing the following text:

```
I am a file. This is my first line
Second line.
Third line.
```

Now let's print it in Python:

```
>>> file = open("test_file.txt",'r')
>>> file.read()
'I am a file. This is my first line\nSecond line.\nThird line.'
```

`read` prints the content of the file till it reaches it's end, and returns the content as string. You can also tell `read` to read a certain amount of bytes:

[39] http://docs.python.org/tutorial/modules.html
[40] http://docs.python.org/tutorial/inputoutput.html
[41] http://docs.python.org/library/stdtypes.html#bltin-file-objects

```
>>> file = open("test_file.txt",'r')
>>> file.read(12)
'I am a file.'
```

We can also read line for line:

```
>>> file = open("test_file.txt",'r')
>>> file.readline()
'I am a file. This is my first line\n'
>>> file.readline()
'Second line.\n'
>>> file.readline()
'Third line.'
>>> file.readline()
''
```

`readline` prints the line till it finds `\n`. Note that we had to reopen the file, because the we reached the end of the file after the first read call. With help of the seek method:

```
>>> file = open("test_file.txt",'r')
>>> file.read(12)
'I am a file.'
>>> file.tell()
12L
```

To set a different position we use the `seek` method. `seek` goes to number of bytes from the position which is set. As default `seek` uses `0` (beginning of file; is equivalent to `os.SEEK_CUR`). Other values are `1` (the current position; os.SEEK_CUR) or `2` (end of file; os.SEEK_END).

For example let's read the first line of the file, and jump to the next line:

```
>>> file.seek(0)
>>> file.readline()
'I am a file. This is my first line\n'
>>> file.seek(13,1)
>>> file.readline()
'Third line.'
```

Alternativley we can use the system's constants:

```
>>> file.seek(0)
>>> from os import SEEK_CUR
>>> file.readline()
'I am a file. This is my first line\n'
>>> file.seek(13,SEEK_CUR)
>>> file.readline()
'Third line.'
```

The command `readlines` prints the lines from the file in a list:

```
>>> file.seek(0)
>>> file.readlines()
['I am a file. This is my first line\n', 'Second line.\n', 'Third line.']
```

To close the file again, use the close method:

```
>>> file.close()
>>> file
<closed file 'test_file.txt', mode 'r' at 0xb7866b10>
```

Now let's add a new line to the file:

```
>>> file = open("test_file.txt",'a')
>>> file.write("This is the fourth line.\n")
>>> file.close()
>>> file = open("test_file.txt",'r')
>>> print(file.read())
I am a file. This is my first line
Second line.
Third line.
This is the fourth line.
```

We can also do that interactively:

```
>>> file = open("test_file.txt",'r+')
>>> file.seek(0,2)
>>> file.write("This is the fift line.\n")
```

The file will only be changed on the disc if we close it or use the `flush` method:

```
>>> file.flush()
```

Now we can read it again:

```
>>> file.seek(0)
>>> print(file.read())
I am a file. This is my first line
Second line.
Third line.
This is the fourth line.
This is the fift line.
```

Oops we forgot an 'h'. Let's change this:

```
>>> file.seek(-7,2)
>>> file.write("h line.\n")
>>> file.flush()
>>> file.seek(0)
>>> print(file.read())
I am a file. This is my first line
Second line.
Third line.
This is the fourth line.
This is the fifth line.
```

The writelines command make it possible to add a list of strings:

```
>>> file.seek(0,2)
>>> file.writelines(["6th line\n","7th line\n"])
>>> file.seek(0,0)
>>> print(file.read())
I am a file. This is my first line
```

```
Second line.
Third line.
This is the fourth line.
This is the fifth line.
6th line
7th line
```

1.4.9 Some words on programming paradigms

There are several programming paradigms, and the most common in modern programming languages are

- Imperative programming
- Functional programming
- Object oriented programming

Look at for exmaple at Wikipedia for an short overwiev on that topic [42], or a good programming book of your choice, If you want to go deeper into that topic.

In short:

- *Imperative programming*: You define sequences of commands the computer should perform, with help of loops, control statements, and functions. The program has *states* which determine, what the program does, and which action to perform. This is a quite natural approach to programming, because a human works also that way, for example: state "hunger" -> get food). Classical examples for such languages are *Fortran* (the first high level language) or *C*.
- ***Functional programming* is a little bit more artifical,** but often a more elegant approach for programming. In functional programming you define functions and let them operate on objects, lists, or call them recursivly. An example would be the *Lisp* family, which was also the first one. (It's worthwile to look at *Lisp* not only to customize your Emacs. A good reading tip would be: Practical Common Lisp [43]) One important benefit of functional programming is, that is easier to parallize. For example it's easier for the compiler/interpreter to decide, when you operate with a function on a list, because all operations are independent anyway, than within a for loop where the compiler/interpreter doesn't know if there are operations which could be possible connected. Other benefits are listed in the *Python Functional Programming Howto*.
- ***Object oriented programming* is (dear computer scientists, don't** send me hatemail) more a way to organize your data, and program than a real paradigm, and in fact you can program OO even in *C* with the help of structs. I already wrote a little about that (see *Object oriented programming abstracts away your problems*), and at least for everyone who does abstraction in a regular basis this is a very intuitive concept. (And in fact every human does!) OO programming means to collect things, that share specific attributes in certain classes. And every Object that shares those features belongs to that class. A real world example would be wheels: There are big wheels, small wheels, wheels for snow etc. but they all share common properties that makes them wheels (For example they are all round, and break on a regular basis).

The good news are, that in Python you are able to work with all three at least to some extend. (Python is more imperativ than funcional). That means Python is a multi paradigm language.

Even if some say that one of the three is the true answer, I personally think that all three have their benefits and drawbacks, and thats the reason I prefer multiparadigm languages like Python, because sometimes it is easier and more intuitive to program a functionality in one certain way, while it's not so easy in the others.

For example I think it's easier and more elegant to write

[42] http://en.wikipedia.org/wiki/Programming_paradigm
[43] http://www.gigamonkeys.com/book/

```
def f(x): return 2*x
x = range(10)

x = map(f,x)
```

than

```
def f(x): return 2*x
x = range(10)

for i in x:
    x[i] = f(x[i])
```

but it's more intuitive and easier to write

```
def f(x): return 2*x
x = range(10)

for i in range(0,10,2):
    x[i] = f(x[i)])
```

than

```
def f(x): return 2*x
x = range(10)

map(lambda i: f(x[i]), range(0,10,2))
```

Links

1.5 Scientific tools in Python

1.5.1 NumPy

NumPy is based on the old Python Project Numeric, and introduces a Matlab like vector class to Python. Numpy is currently developed by Entought and is distrubuted und the BSD license. Further Information and intall instructions can be found on the official NumPy website [44]. Sage and FemHUB are shipped with current versions of NumPy.

The project is still in an active development process, and release new versions in an regular basis (The last release before I started to writing this report is 2 Months old)

For people who are familiar with Matlab I recommend the online equivalence list between Matlab and Numpy from *Mathesaurus* [45] for the first steps with NumPy.

How to load Numpy

To import numpy into Python simply write:

```
import numpy
```

[44] http://numpy.scipy.org/
[45] http://mathesaurus.sourceforge.net/matlab-numpy.html

You can also import the whole namespace via `*`:

```
from numpy import *
```

Remark I personally don't recommend to load the complete nammespace, or the complete package, except for testing, due to performance reasons. (This is somehow obvious because NumPy is a rather big module)

Numpy arrays

The NumPy *array* class is rather similar to Python *lists*. It's the basic datatype for many numerical tools in Python.

Array Creation

To create an *array* we have to import it from NumPy first:

```
from numpy import array
```

An *array* can be created from different Python sequence types like *lists* or tuples. For example:

```
>>> l = [1,2,3]
>>> t = (1,2,3)
>>> array(l)
array([1, 2, 3])
>>> array(t)
array([1, 2, 3])
```

Remark It is not clerly specified by the documention which other containers may work (I guess the reason for this is that the constructor is written in a quite generic way. The Python way to find out if it work with other tzpes would be testing it out,

The intention of the NumPy developers was to give a Matlab like feeling. So Many ways should be quite familiar for Matlab users:

```
>>> from numpy import zeros, ones, eye
>>> from numpy.random import rand
>>> zeros(3)
array([ 0.,  0.,  0.])
>>> ones(4)
array([ 1.,  1.,  1.,  1.])
>>> eye(2)
array([[ 1.,  0.],
      [ 0.,  1.]])
>>> rand(4)
array([ 0.62475625,  0.97783392,  0.7785848 ,  0.15707817])
>>> zeros((2,2))       #create matrix
array([[ 0.,  0.],
       [ 0.,  0.]])
```

A NumPy array can hold numbers of specific data types. To check which datatype an array holds, one has to simply check the `dtype` member:

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

```
>>> l2 = array([1.,2])
>>> l2.dtype
dtype('float64')
```

As one can see, the default datatpe for integers is `int64`, while for floating point numbers it is `float64` (because it is a 64 bit system I am working on). But there are some more:

```
from numpy import float32  #single precision
from numpy import float64  #double precision
from numpy import float128 #long double

from numpy import int16  #16 Bit integer
from numpy import int32  #32 Bit integer
from numpy import int64  #64 Bit integer
from numpy import int128 #128 Bit Integer
```

To create an array with a specific data type, you only have to specify this:

```
>>> array([2,3],int32)
array([2, 3], dtype=int32)
>>> array([2,3],dtype=int32)  #using keyword argument
array([2, 3], dtype=int32)
```

But these aren't all possible datatypes. NumPy support also other types, and the number is still growing, since it is under development.

There are several other ways to create arrays. See the NumPy documentation [46], [47] for further details.

Artithmetics with NumPy arrays

Since operators can be overloaded (see ::ref::*overload_ref* , NumPy supports also arithmetics with *arrays*. Note that all operations are elementwise.

```
>>> a = array([2.,3]); b = array([5.,6])   # Create vectors
>>> a + b
array([ 7.,  9.])
>>> a - b
array([-3., -3.])
>>> a * b
array([ 10.,  18.])
>>> a / b
array([ 0.4,  0.5])
>>> a ** b
array([  32.,  729.])
```

To calculate the scalar product one has to use the `dot` function:

```
>>> from numpy import dot
>>> dot(a,b)
28.0
```

With the help of dot you can also calculate the matrix vector product:

[46] http://docs.scipy.org/doc/numpy-1.5.x/user/basics.creation.html#arrays-creation
[47] http://docs.scipy.org/doc/numpy-1.5.x/reference/routines.array-creation.html#routines-array-creation

```
>>> A = ones((2,2))
>>> A
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> dot(A,a)
array([ 5.,  5.])
```

Applying functions elementwise

NumPy also holds a lot of standard functions for elementwise operations:

```
>>> from numpy import sin, cos
>>> sin(a)
array([ 0.90929743,  0.14112001])
>>> cos(a)
array([-0.41614684, -0.9899925 ])
```

(see the NumPy reference guide for further information [48])

To create your own customized elementwise functions use the `vectorize` class in NumPy. It takes a Python function for construction of the object, and vectorize it.

Examples:

```
from numpy import vectrorize, array
from numpy.random import randn

def my_sign(x):
    if  x > 0:
        return 1
    elif x < 0:
        return -1
    else:
        return 0

vec_abs = vectorize(my_sign)
```

Then we get:

```
>>> vec = randn(10); vec
array([ 1.2577085 ,  0.71063021,  1.41130699,  1.72412141, -1.18530781,
        0.19527091, -0.20557102, -0.33562998, -1.5370958 , -0.47241905])
>>> vec_abs(vec)
array([ 1,  1,  1,  1, -1,  1, -1, -1, -1, -1])
```

1.5.2 SciPy

SciPy is a module for scientific computing. It is based on NumPy and holds a lot of extensions and algorithms. In fact NumPy is subsumed in SciPy already. It contains a lot of functionality which is contained in Matlab.

I will explain some scientific tools in detail, which are of common interest.

[48] http://docs.scipy.org/doc/numpy/reference/routines.math.html

Linear Algebra

For doing linear algebra with SciPy I would prefer to point at the SciPy documentation, because it is much more detailed [49]

Sparse Linear Algebra

There are several types of sparse matrices. Each of them has several attributes and is used for different tasks. I introduce here the ones I use the most, and some other important features like the LinearOperator class.

LIL (List of Lists)

LIL matrices are made for creating sparse matrices. To create a LIL matrix simply import the class and call the constructor:

```
>>> from scipy.sparse import lil_matrix
>>> A = lil_matrix((1000,1000))
```

Now we can fill the entries like we do it normally with numpy vectors:

```
>>> from scipy import rand
>>> A[0, 0:100] = rand(100); A
<1000x1000 sparse matrix of type '<type 'numpy.float64'>'
      with 100 stored elements in LInked List format>
>>> A[1:21,1:21] = rand(20,20); A
<1000x1000 sparse matrix of type '<type 'numpy.float64'>'
      with 500 stored elements in LInked List format>

and of course call the entries directly::

>>> A[1,1]
0.85312312525865719
```

LIL matrices are not suited for arithmetics or vector operations but for creating other sparse matrices. To convert it into an other sparse type simply call the converting methods. Lets convert it for example to CSC (Compressed Sparse Column matrix) format:

```
>>> A_csc = A.tocsc()
>>> A_csc
<1000x1000 sparse matrix of type '<type 'numpy.float64'>'
      with 500 stored elements in Compressed Sparse Column format>
```

To convert it back to a numpy vector simply call:

```
>>> A.toarray()
array([[ 0.16568301,  0.85841039,  0.58243887, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.85312313,  0.33507849, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.97454761,  0.16457123, ...,  0.        ,
         0.        ,  0.        ],
       ...,
```

[49] http://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html

```
[ 0.        ,  0.        ,  0.        , ...,  0.        ,
  0.        ,  0.        ],
[ 0.        ,  0.        ,  0.        , ...,  0.        ,
  0.        ,  0.        ],
[ 0.        ,  0.        ,  0.        , ...,  0.        ,
  0.        ,  0.        ]])
```

CSC (Compressed Sparse Column) matrix

CSC matrices are quite often used because they can perform matrix vector multiplication quite efficiently. To create a CSC matrix either do it with a LIL matrix like in the LIL matrix section before, or create it with three arrays which contain the necessary data:

```
>>> from scipy.sparse import csc_matrix
>>> from numpy import array
>>> rows = array([0,2,2,1,0])
>>> cols = array([0,2,0,2,1])
>>> data = array([1,2,3,4,5])
>>> B = csc_matrix((data, (rows,cols)), shape = (3,3)); B.toarray()
array([[1, 5, 0],
       [0, 0, 4],
       [3, 0, 2]])
```

Another variant would be the standard CSC representation. There are three arrays: an index_pointer array, an indices array, and a data array. The row indices for the ::math::i th row are stored in `indices[index_pointer[i],index_pointer[i+1]]`, while their corresponding data is stored in `data[index_pointer[i]:index_pointer[i+1]]`. So the `index_pointer` tells where to start and to stop while going throug the indices and data lists. For example:

```
>>> indptr = array([0,2,3,6])
>>> indices = array([0,2,2,0,1,2])
>>> data = array([1,2,3,4,5,6])
>>> csc_matrix( (data,indices,indptr), shape=(3,3) ).toarray()
array([[1, 0, 4],
       [0, 0, 5],
       [2, 3, 6]])
```

Other possible ways would be generating the matrix with another sparse matrix or an dense 2D array with the data as constructing data:

```
>>> csc_matrix(array([[0,1],[1,0]]))
<2x2 sparse matrix of type '<type 'numpy.int32'>'
        with 2 stored elements in Compressed Sparse Column format>
>>> csc_matrix(array([[0,1],[1,0]])).toarray()
array([[0, 1],
       [1, 0]])
```

...and more

To get more information on sparse matrices and their class methods consult the scipy reference guide [50]

[50] http://docs.scipy.org/doc/scipy/reference/sparse.html

The LinearOperator class and iterative solvers

The LinearOperator class allows to define abstract linear mappings, which are not necessarily matrices. A linear operator only consists of a tuple which represents the shape, and a matrix-vector multiplication:

```
>>> def my_matvec(x):
...     a = x[-1]
...     x[-1] = x[0]
...     x[0] = a
...     return x
...
>>> from scipy.sparse.linalg import LinearOperator
>>> lin = LinearOperator((3,3),matvec=my_matvec)
>>> x = array([1,2,3])
>>> lin.matvec(x)
array([3, 2, 1])
```

The matrix vector multiplication can also be called with the * operator:

```
>>> lin*x
array([1, 2, 3])
>>> lin * x
array([3, 2, 1])
```

LinearOperators can be created from arrays, matrices or sparse matrices with the *aslinearoperator* function:

```
>>> A = array([[2,-1,0],[-1,2,-1],[0,-1,2]])
>>> from scipy.sparse.linalg import aslinearoperator
>>> A_lin = aslinearoperator(A)
>>> A_lin.matvec(x)
array([4, 0, 0])
```

The LinearOperator class is mostly used for iterative Kylov solvers. Those methods can be found in the *scipy.sparse.linalg*. For example the CG algorithm:

```
>>> from scipy.sparse.linalg import cg
>>> A_lin*sol[0]
array([ 3.,  2.,  1.])
>>> x
array([3, 2, 1])
```

For more information see again the SciPy reference [51]

1.5.3 Weave

Weave is included in SciPy and a tool for writing inline C++ with weave for speedup your code. I give here a short example how to use Weave.

Consider band-matrix vector multiplication:

```
def band_matvec_py(A,u):

    result = zeros(u.shape[0],dtype=u.dtype)
```

[51] http://docs.scipy.org/doc/scipy/reference/sparse.linalg.html

```
    for i in xrange(A.shape[1]):
        result[i] = A[0,i]*u[i]

    for j in xrange(1,A.shape[0]):
        for i in xrange(A.shape[1]-j):
            result[i] += A[j,i]*u[i+j]
            result[i+j] += A[j,i]*u[i]

    return result
```

This is not very fast:

```
sage: import numpy
sage: datatype = numpy.float64
sage: N = 2**14
sage: B = 2**6
sage: A = rand(B,N).astype(datatype)
sage: %timeit band_matvec_py(A,u)
5 loops, best of 3: 3.48 s per loop
```

The reason for this is that array access is quite costly in Python. A possibility to make that better would be to write C++ code inline with the Weave module. To do that give the Python Interpreter your C++ code as string, and then let compile it. Here an implementation of the band-matrix vector multiplication with weave:

```
from numpy import array, zeros
from scipy.weave import converters
from scipy import weave

def band_matvec_inline(A,u):

    result = zeros(u.shape[0],dtype=u.dtype)

    N = A.shape[1]
    B = A.shape[0]

    code = """
    for(int i=0; i < N;i++)
    {
      result(i) = A(0,i)*u(i);
    }
    for(int j=1;j < B;j++)
    {

        for(int i=0; i < (N-j);i++)
        {
          if((i+j < N))
          {
            result(i) += A(j,i)*u(j+i);
            result(i+j) += A(j,i)*u(i);
          }

        }

    }
    """

    weave.inline(code,['u', 'A', 'result', 'N', 'B'],
```

```
                type_converters=converters.blitz)
    return result
```

As it can be seen the syntax is not that different from Numpy. The reason for this is, that Weave uses here the Blitz library for numerical computation, which has it's own vector class.

If you call this function the first time it will be compiled in runtime:

```
sage: band_matvec_inline(A,u)
creating .../python26_intermediate/compiler_2da6387b1d12110fba46fe47fea9326a
In file included from .../python2.6/site-packages/scipy/weave/blitz/blitz/array-impl.h:37,
                 from .../python2.6/site-packages/scipy/weave/blitz/blitz/array.h:26,
                 from ....python26_compiled/sc_7f8ca882b38e1f398003844545921f4a0.cpp:11:
.../blitz/blitz/range.h: In member function 'bool blitz::Range::isAscendingContiguous() const':
.../blitz/range.h:120: warning: suggest parentheses around '&&' within '||'
array([-7.03708979, -0.53476595, -7.52383126, ...,  1.18391403,
        2.27257052,  0.39116477])
```

The next time you call it, the interpreter will use the compiled program. Let's test the speedup:

```
sage: %timeit band_matvec_inline(A,u)
25 loops, best of 3: 12.7 ms per loop
```

This was now about 270x faster than the original Python version. For more information on using weave see either the documentation of SciPy [52] or the Sage tutorial on that topic [53].

Note: At the time I checked the Sage tutorial the last time it was not updated and contain some mistakes. In the next version of Sage (4.6.2) this should be corrected. See the Sage trac for a corrected version [54]

Links

1.6 Cython

Well Cython isn't a part of Python, it is a different language, but very similar to Python, and in fact it is almost to 90% compatible. (It is stated that Cython is a superset of Python, but it's currently under development so there are some features which are not supported yet!)

It first started with the Pyrex project, which allowed to compile Python to C. The idea was to allow the user to declare C variables and call C functions within Cython, and make it possible for the C compiler to compile the Python like code to fast C code.

Cython has bindings for NumPy, mpi4py and other Python modules to support scientific computation.

Currently Cython only works on the CPython implementation, but there are efforts to get it working in IronPython on .Net as well.

I will here give a short tutorial on Cython and demonstrate on an example how to speed up your NumPy code.

Important Note I assume that you are using Linux as operating system. If you use Windows or an other OS look up the Cython documentation for specific details! [55]

[52] http://www.scipy.org/Weave
[53] http://www.sagemath.org/doc/numerical_sage/weave.html
[54] http://trac.sagemath.org/sage_trac/ticket/9791
[55] http://docs.cython.org/index.html

1.6.1 How to compile your Cython Code

Sage

This is the easiest way. Either write your Cython code in a *.spyx* (Sage Pyrex) file, or in the notebook, with the magic function `%cython`.

To use a *.spyx* file simply load it into Sage with the `load` command:

```
load my_cython_file.spyx
```

For example I write a short code snippet for an self made scalar product:

```
def my_dot(x,y):

    if x.size != y.size
        raise ValueError("Dimension Mismatch")

    result = 0

    for i in range(x.size):
        result += x[i]*y[i]

    return result
```

I save this in the file my_dot.spyx. Now I call Sage, and cd to the directory I saved the file. Now simply call Sage, and type:

```
sage: load my_dot.spyx
Compiling ./my_dot.spyx...
```

Now the function can be called directly like a normal Python function:

```
sage: from numpy import array
sage: x = array([1,2,3.])
sage: y = array([1,0,5])
sage: my_dot(x,y)
16.0
```

A different way would be in the notebook. Simply write in an empty notebook cell:

```
%cython
def my_dot(x,y):

    if x.size != y.size:
        raise ValueError("Dimension Mismatch")

    result = 0

    for i in range(x.size):
        result += x[i]*y[i]

    return result
```

Now if you evalute it, the function will be compiled, and you can call it normally.

Setup files

The direct approach in Python would be to write a setup file. First write your code and save it to a *.pyx* file. I use the same code as before and write it to *my_dot.pyx*.

Now we use disutils and write a setup.py file, which works similar to a make file:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("my_dot", ["my_dot.pyx"])]
)
```

Save this as setup.py in the directory where your code file lies.

Now cd to your working directory where the code and setup file is saved and call it with:

```
python setup.py -build_ext --inplace
```

Then the *.pyx* files will be compiled. Now you can call it normally in Python (after changing to the working directory):

```
>>> from my_dot import my_dot
```

To compile more files, simply put more extensions to the ext_modules list. I created for example a further file with the name *test.pyx*

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("my_dot", ["my_dot.pyx"]),
                   Extension("test", ["test.pyx"])]
)
```

Important: If you import numpy as C library you have to add `include_dirs=[numpy.get_include()])` to the extension. In our example this would look like this:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("my_dot", ["my_dot.pyx"],
    include_dirs=[numpy.get_include()])]
)
```

I state this here, because it is not well documented in the Cython docu, and I had to search it for long in Cython Mailing list. How to import modules as C libraries will we see later.

1.6.2 How to use Cython

Here we look at the advanced syntax in Cython, and other features in Python.

The *cdef* statment

Type declaration

`cdef` is used for C type declaration, and defining C functions. This can be very useful for speeding up your Python programs.

Let's look at our scalar product again:

```
def my_dot(x,y):

    if x.size != y.size
        raise ValueError("Dimension Mismatch")

    result = 0

    for i in range(x.size):
        result += x[i]*y[i]

    return result
```

The counter variables cost a lot of efficiency because the program has to check first, what it recieves, because in Python `i` could be every type of object. To overcome this we tell Cython to take a normal C integer:

```
def my_dot(x,y):

  if x.size != y.size
      raise ValueError("Dimension Mismatch")

  cdef double result = 0

  cdef int i

  for i in range(x.size):
      result += x[i]*y[i]

  return result
```

Now you can compile and use it. Let's measure the difference:

```
sage: x = randn(10**6)
sage: y = randn(10**6)
sage: %timeit my_dot(x,y)
5 loops, best of 3: 1.1 s per loop
sage: load my_dot.spyx
Compiling ./my_dot.spyx...
sage: %timeit my_dot(x,y)
5 loops, best of 3: 653 ms per loop
```

We this was already twice as fast as the old version. (I used a Pentium Dual Core with 1.8 GHz, and 2 GB Ram). This is not that much, but more is possible!

The next step would be to tell the function which data types to use:

```
cimport numpy as cnumpy

ctypedef cnumpy.float64_t reals #typedef_for easier reedding

def my_dot(cnumpy.ndarray[reals,ndim=1] x,
              cnumpy.ndarray[reals,ndim=1] y):

    if x.size != y.size:
        raise ValueError("Dimension Mismatch")

    cdef double result = 0

    cdef int i

    for i in range(x.size):
        result += x[i]*y[i]

    return result
```

In the first line we used the `cimport` statement to load the C version of NumPy. (I explain cimport later) Then we used the `ctypedef` statment to declare the float64 (double) datatype as reals, so that we have to type less (like the typedef statement in C).

The main difference in this example is that we told Cython that the input should be to NumPy arrays. This avoids unecessary overhead. Now we make the timing again:

```
sage: load my_dot.spyx
Compiling ./my_dot.spyx...
sage: %timeit my_dot(x,y)
125 loops, best of 3: 3.54 ms per loop
```

This was now about 300x faster than the original version.

The drawback is that the Cython function only take numpy arrays:

```
sage: x = range(5)
sage: y = randn(5)
sage: my_dot(x,y)
...
TypeError: Argument 'x' has incorrect type (expected numpy.ndarray, got list)
```

Declaring functions

The `cdef` statement can also be used for defining functions. A function that is defined by a `cdef` statment doesn't appear in the namespace of the Python interpreter and can only be called within other functions.

For example let's define a `cdef` function `f`:

```
cdef double f(double x):
    return x**2 - x
```

If yould try now to call it Python won't find it:

```
NameError: name 'f' is not defined
```

But you can call it within an other function defined in a *.pyx*

```
def call_f(double x):
    return f(x)
```

Another possibility would be the cpdef statement:

```
cdef double f(double x):
    return x**2 - x
```

This function can now be called both ways.

Note: If you don't declare it, `cdef` functions can't handly exceptions right. For example

```
cdef double f(double x):

    if x == 0:
        raise ValueError("Division by Zero!")

    return x**(-2) - x
```

would not raise a Python exception. To do this use the except statement:

```
cdef double f(double x) except *:

    if x == 0:
        raise ValueError("Division by Zero!")

    return x**(-2) - x
```

The `*` means that the function should propagate arbitrary exceptions. To be more specific you can also handle specific output:

```
cdef double f(double x) except? 0:

  if x == 0:
      raise ValueError("Division by Zero!")

  return x**(-2) - x
```

The `?` here means that `0` is accepted as output too (or else you would recieve an error if `0` is returned)

`cdef` classes

Classes can also be defined with `cdef` also. Let's take the example from the Cython documentation (see [56]):

```
cdef class Function:
    cpdef double evaluate(self, double x) except *:
        return 0
```

A `cdef` class is also called Extension Type.

This class can be derived like a normal Python class:

[56] http://docs.cython.org/src/tutorial/cdef_classes.html

```
cdef class SinOfSquareFunction(Function):
    cpdef double evaluate(self, double x) except *:
        return sin(x**2)
```

`cdef` classes are very limited in comparison to Python classes, because C don't know classes, but only structs. (Since Cython 0.13 it is possible to wrap C++ classes. See the Cython documentation for further details [57])

We can use this new class like a new datatype. See again an example from the Cython documentation:

```
def integrate(Function f, double a, double b, int N):
    cdef int i
    cdef double s, dx
    if f is None:
        raise ValueError("f cannot be None")
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f.evaluate(a+i*dx)
    return s * dx

print(integrate(SinOfSquareFunction(), 0, 1, 10000))
```

Calling extern C functions

In Cython it is possible to call functions from other C programs defined in a header file. For example we want to wrap the sinus from the math.h in a Python function. Then we would write for example in a *.pyx* file:

```
cdef extern from "math.h":
  double sin(double)

def c_sin(double x):
    return sin(x)
```

The `cdef extern` statement help us to call `sin` from C. The `c_sin` function only serves as a wrapper for us, because we can't call a `cdef` function directly. If you want to call your Python function with sin, you can rename the extern C function with a custom made identifier:

```
cdef extern from "math.h":
    double c_sin "sin"(double)

def sin(double x):
    return c_sin(x)
```

The `c_sin` is the name of the `cdef` function.

If you want to compile this file, you have to tell your compiler which libraries you linked, because they are not linked automatically! In this case it is the math library with abbreviation "m". You have to specify this in your setup file (I saved the sinus to *math_stuff.pyx*):

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
```

[57] http://docs.cython.org/src/userguide/wrapping_CPlusPlus.html

```
    name = "Math Stuff",
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("math_stuff", ["math_stuff.pyx"],
                   libraries=["m"])]
    #m for compiler flag -lm (math library)
)
```

If you use Sage you have to specify this directly in the *.spyx* file with:

```
#clib m
```

in our example this would look like this:

```
#clib m

cdef extern from "math.h":
    double c_sin "sin"(double)

def sin(double x):
    return c_sin(x)
```

Another example: Let's link the scalar product from the BLAS library:

```
cimport numpy

ctypedef numpy.float64_t reals #typedef_for easier reedding

cdef extern from "cblas.h":
    double ddot "cblas_ddot"(int N, double *X, int incX,double *Y, int incY)

def blas_dot(numpy.ndarray[reals,ndim = 1] x, numpy.ndarray[reals,ndim = 1] y):
    return ddot(x.shape[0],<reals*>x.data,x.strides[0] // sizeof(reals), <reals*>y.data,y.strides[0]
```

The blas implementation gives only a small improvement here (which is not completely unexpected, because the algorithm is rather simple):

```
sage: x = randn(10**6)
sage: y = randn(10**6)
sage: %timeit my_dot(x,y)
125 loops, best of 3: 3.55 ms per loop
sage: %timeit blas_dot(x,y)
125 loops, best of 3: 3.05 ms per loop
```

`cimport` and .pxd files

.pxd are like *.h* files in C. They can be used for sharing external C declarations, or functions that are suited for inlining by the C compiler.

Functions that are declared inline in *.pxd* files can be imported with the `cimport` statement.

For example let's add a function which calculates the square root of a number to the *math_stuff.pyx* from earlier, where the operation itelf is called as inline function from C. We write the inline function to the file *math_stuff.pxd*:

```
cdef inline double inl_sqrt(double x):
    return x**(0.5)
```

We can now load this function from a *.pyx* file:

```
def sqrt(double x):
    return inl_sqrt(x)
```

You can also save the extern definition of the BLAS scalar product to a *.pxd* file and can `cimport` it from there.

Here the *blas.pxd* file:

```
cdef extern from "cblas.h":
    double ddot "cblas_ddot"(int N, double *X, int incX,double *Y, int incY)
```

and here the addition to the math_stuff.pyx:

```
cimport numpy

from blas cimport ddot
ctypedef numpy.float64_t reals #typedef_for easier reedding

cpdef dot(numpy.ndarray[reals,ndim = 1] x, numpy.ndarray[reals,ndim = 1] y):
    return ddot(x.shape[0],<reals*>x.data,x.strides[0] //
    sizeof(reals), <reals*>y.data,y.strides[0] // sizeof(reals))
```

What is also possible is to declare prototypes in a *.pxd* file like in a C header, which can be linked more efficiently.

For example let's make a prototype of a function and a class:

```
cpdef dot(numpy.ndarray[reals,ndim = 1] x, numpy.ndarray[reals,ndim = 1] y)

cdef class Function:
    cpdef double evaluate(self, double x)
```

Profiling

Profiling is a way to analyse and optimize your Cython programs. I only give the reference to a tutorial in the Cython documentation here [58]

Links

1.7 MPI4Py

MPI4Py is a Python module for calling the MPI API. For more information and detailed documentation I refer to the official MPI4Py documentation [59] Let's start with the MPI Hello World program in Python:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
print("hello world")
print("my rank is: %d"%comm.rank)
```

As it can be seen the API is quite similar to the normal MPI API in C. First we save this file as *mpi.py*. To call now our parallized version of the Hello World program simply call the Python Interpreter with MPI:

[58] http://docs.cython.org/src/tutorial/profiling_tutorial.html
[59] http://mpi4py.scipy.org/docs/usrman/index.html

```
$ ./where/mpi/is/installed/mpirun -n <nr_processes> python mpi.py
```

(If you use Sage, you have to install the openmpi package, and then you can find mpirun in `SAGE_LOCAL/bin/`) I for example use Sage, and this would look like this:

```
$ $SAGE_ROOT/local/bin/mpirun -n 4 sage -python mpi.py
hello world
my rank is: 2
hello world
my rank is: 0
hello world
my rank is: 1
hello world
my rank is: 3
```

Here another example: We generate an array with a thread which is currently our main thread. Then we distribute it over all threads we called:

```
from mpi4py import MPI
import numpy
import time

comm = MPI.COMM_WORLD
rank = comm.rank

sendbuf=[]
root=0
if rank==0:
    m=numpy.random.randn(comm.size,comm.size)
    print(m)
    sendbuf=m
    t1 = time.time()

v=MPI.COMM_WORLD.scatter(sendbuf,root)

print(rank,"I got this array:")
print(rank,v)

v=v*2

recvbuf=comm.gather(v,root)

if rank==0:
  t2 = time.time()
  print numpy.array(recvbuf)
  print "time:", (t2-t1)*1000, " ms "
```

This snippet produces this output:

```
$ $SAGE_ROOT/local/bin/mpirun -n 3 sage -python mpi_scatter.py
[[ -5.90596754e-04   4.21504158e-02   2.11213337e-01]
 [  9.67314022e-01  -2.16766512e+00   1.00552694e+00]
 [  1.37283086e+00  -2.29582623e-01   2.88653028e-01]]
(0, 'I got this array:')
(0, array([-0.0005906 ,  0.04215042,  0.21121334]))
(1, 'I got this array:')
```

```
(1, array([ 0.96731402, -2.16766512,  1.00552694]))
(2, 'I got this array:')
(2, array([ 1.37283086, -0.22958262,  0.28865303]))
[[ -1.18119351e-03   8.43008316e-02   4.22426674e-01]
 [  1.93462804e+00  -4.33533025e+00   2.01105389e+00]
 [  2.74566171e+00  -4.59165246e-01   5.77306055e-01]]
time: 3.59892845154  ms
```

For further examples I refer to the Sage tutorial for scientific computing. [60] **Note** The last time I checked the tutorial, it was outdated. If you need a corrected version, I posted one on Sage trac [61].

Links

1.8 Python+CUDA = PyCUDA

PyCUDA is a Python Interface for CUDA [62]. It is currently in Alpha Version, and was developed by Andreas Klöckner [63]

To use PyCUDA you have to install CUDA on your machine

Note: For using PyCUDA in Sage or FEMHub I created a PyCUDA package [64].

I will give here a short introduction how to use it. For more detailed Information I refer to the documentation [65] or the Wiki [66].

1.8.1 Initialize PyCUDA

There are two ways to initialize the PyCUDA driver. The first one is to use the autoinit module:

```
import pycuda.autoinit
```

This makes the first device ready for use. Another possibility is to manually initialize the device and create a context on this device to use it:

```
import pycuda.driver as cuda
cuda.init() #init pycuda driver
current_dev = cuda.Device(device_nr) #device we are working on
ctx = current_dev.make_context() #make a working context
ctx.push() #let context make the lead

#Code

ctx.pop() #deactivate again
ctx.detach() #delete it
```

This is useful if you are working on different devices. I will give a more detailed example combined with MPI4Py lateron. (See *Using MPI4Py and PyCUDA together*)

[60] http://www.sagemath.org/doc/numerical_sage/mpi4py.html
[61] http://trac.sagemath.org/sage_trac/attachment/ticket/10566/mpi4py.rst
[62] http://www.nvidia.com/object/cuda_home_new.html
[63] http://mathema.tician.de/software/pycuda
[64] http://trac.sagemath.org/sage_trac/ticket/10010
[65] http://documen.tician.de/pycuda/
[66] http://wiki.tiker.net/PyCuda

1.8.2 Get your CUDA code working in Python

Similar to ::ref::*weave_ref* we can write CUDA code as string in Python and then compile it with the NVCC. Here a short example:

First we initialize the driver, and import the needed modules:

```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy
from pycuda.compiler import SourceModule
```

Then we write our Source code:

```
code = """
__global__ void double_array_new(float *b, float *a, int *info)
{
  int datalen = info[0];

  for(int idx = threadIdx.x; idx < datalen; idx += blockDim.x)
  {
    b[idx] = a[idx]*2;
  }
}
"""
```

And then write it to a source module:

```
mod = SourceModule(code)
```

The NVCC will now compile this code snippet. Now we can load the new function to the Python namespace:

```
func = mod.get_function("double_array_new")
```

Let's create some arrays for the functions, and load them on the card:

```
N = 128

a = numpy.array(range(N)).astype(numpy.float32)
info = numpy.array([N]).astype(numpy.int32)
b = numpy.zeros_like(a)

a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)

b_gpu = cuda.mem_alloc(b.nbytes)
cuda.memcpy_htod(b_gpu, b)

info_gpu = cuda.mem_alloc(info.nbytes)
cuda.memcpy_htod(info_gpu, info)
```

Now we can call the function:

```
func(b_gpu, a_gpu,info_gpu, block = (32,1,1), grid = (4,1))
```

Note: The keyword `grid` is optional. If no grid is assigned, it consists only of one block.

Now get the data back to the host, and print it:

```
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, b_gpu)

print "result:", a_doubled
```

Note: To free the memory on the card use the free method:

```
a_gpu.free()
b_gpu.free()
info_gpu.free()
```

PyCUDA has Garbage Collection, but it's still under developement. I Therefore recommend it to free data after usage, just to be sure.

To create a Texture reference, to bind data to a texture on the Graphic card. you have first to create one your source code:

```
code_snippet = """
texture<float, 2> MyTexture;
// Rest of Code
"""
```

Then compile it:

```
>>> texture_mode = SourceModule(code_snippet)
```

and get it:

```
>>> MyTexture = texture_mode.get_texref("MyTexture")
```

1.8.3 The `gpuarray` class

The `gpuarray` class provides a high level interface for doing calculations with CUDA. First import the gpuarray class:

```
>>> import pycuda.driver as cuda
>>> import pycuda.autoinit
>>> from pycuda import gpuarray
```

Creation of gpuarrays is quite easy. One way is to create a NumPy array and convert it:

```
>>> from numpy.random import randn
>>> from numpy import float32, int32, array
>>> x = randn(5).astype(float32)
>>> x_gpu = gpuarray.to_gpu(x)
```

You can print gpuarrays like you normally do:

```
>>> x
array([-0.24655211,  0.00344609,  1.45805557,  0.22002029,  1.28438667])
>>> x_gpu
array([-0.24655211,  0.00344609,  1.45805557,  0.22002029,  1.28438667])
```

You can do normal calculations with the gpuarray:

```
>>> 2*x_gpu
array([-1.09917879,  0.56061697, -0.19573164, -4.29430866, -2.519032  ], dtype=float32)

>>> x_gpu + x_gpu
array([-1.09917879,  0.56061697, -0.19573164, -4.29430866, -2.519032  ], dtype=float32)
```

or check attributes like with normal arrays:

```
>>> len(x_gpu)
5
```

`gpuarrays` also support slicing:

```
>>> x_gpu[0:3]
array([-0.5495894 ,  0.28030849, -0.09786582], dtype=float32)
```

Unfortunatly they don't support indexing (yet):

```
>>> x_gpu[1]
...
ValueError: non-slice indexing not supported: 1
```

Be aware that a function which was created with a SourceModule, takes an instance of `pycuda.driver.DeviceAllocation` and not a gpuarray. But the content of the `gpuarray` is a `DeviceAllocation`. You can get it with the attribute `gpudata`:

```
>>> x_gpu.gpudata
<pycuda._driver.DeviceAllocation object at 0x8c0d454>
```

Let's for example call the function from the section before:

```
>>> y_gpu = gpuarray.zeros(5,float32)
>>> info = array([5]).astype(int32)
>>> info_gpu = gpuarray.to_gpu(info)
>>> func(y_gpu.gpudata,x_gpu.gpudata,info_gpu.gpudata, block = (32,1,1), grid = (4,1))
>>> y_gpu
array([-1.09917879,  0.56061697, -0.19573164, -4.29430866, -2.519032  ], dtype=float32)
>>> 2*x_gpu
array([-1.09917879,  0.56061697, -0.19573164, -4.29430866, -2.519032
>>> ], dtype=float32)
```

`gpuarrays` can be bound to textures too:

```
>>> x_gpu.bind_to_texref_ext(MyTexture)
```

1.8.4 Using MPI4Py and PyCUDA together

I give here a short example how to use this, to get PyCUDA working with MPI4Py. We initialize as many threads, as graphic cards available (in this case 4) and do something on that devices. Every thread is working on one device.

```
from mpi4py import MPI
import pycuda.driver as cuda

cuda.init() #init pycuda driver

from pycuda import gpuarray
from numpy import float32, array
from numpy.random import randn as rand
import time

comm = MPI.COMM_WORLD
rank = comm.rank
root = 0

nr_gpus = 4

sendbuf = []

N = 2**20*nr_gpus
K = 1000

if rank == 0:
    x = rand(N).astype(float32)*10**16
    print "x:", x

    t1 = time.time()

    sendbuf = x.reshape(nr_gpus,N/nr_gpus)

if rank > nr_gpus-1:
    raise ValueError("To few gpus!")


current_dev = cuda.Device(rank) #device we are working on
ctx = current_dev.make_context() #make a working context
ctx.push() #let context make the lead

#recieve data and port it to gpu:
x_gpu_part = gpuarray.to_gpu(comm.scatter(sendbuf,root))

#do something...
for k in xrange(K):
  x_gpu_part = 0.9*x_gpu_part

#get data back:
x_part = (x_gpu_part).get()

ctx.pop() #deactivate again
ctx.detach() #delete it

recvbuf=comm.gather(x_part,root) #recieve data

if rank == 0:
    x_doubled = array(recvbuf).reshape(N)
    t2 = time.time()-t1

    print "doubled x:", x_doubled
    print "time nedded:", t2*1000, " ms "
```

Links

1.9 An Example: Band-matrix vector multiplication

We want to implement a band-matrix class for a symmetric band-matrix. The constructor takes an array as input, which holds the matrix entries, of the lower part of the band matrix. See Wikipedia for an Idea [67], and the IBM ESSL for precise details [68].

First we implement our class with several class methods, like addition and a matvec method:

```
def add_band_mat(A,M,beta = 1,alpha = 1):
    (d1,d2) = A.shape
    (d3,d4) = M.shape

    d1 = A.band_width
    d3 = M.band_width

    if d2 != d4:
        raise ValueError("From _rational_krylov_trigo_band:\
                          Dimension Missmatch!")

    if (d1 < d3):
        SYST = py_band_matrix(zeros((d3,d4)));
        SYST.data[0:d1,0:d2] = beta*A.data;
        SYST.data = alpha*M.data + SYST.data;
    elif (d1 > d3):
        SYST = py_band_matrix(zeros((d1,d2)));
        SYST.data[0:d3,0:d4] = alpha*M.data;
        SYST.data = SYST.data + beta*A.data;
    else:
        SYST =  py_band_matrix(alpha*M.data + beta*A.data);

    return SYST

class band_matrix:
    def __init__(self,ab):
        self.shape = (ab.shape[1],ab.shape[1])
        self.data = ab
        self.band_width = ab.shape[0]
        self.dtype = ab.dtype

    def matvec(self,u):
        pass

    def __add__(self,other):
        return add_band_mat(self,other)
```

First we implement our matrix vector multiplication in Python:

```
def band_matvec_py(A,u):

    result = zeros(u.shape[0],dtype=u.dtype)
```

[67] http://en.wikipedia.org/wiki/Band_matrix

[68] http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=%2Fcom.ibm.cluster.essl43.guideref.doc%2Fam501_upbsm.html

```
    for i in xrange(A.shape[1]):
        result[i] = A[0,i]*u[i]

    for j in xrange(1,A.shape[0]):
        for i in xrange(A.shape[1]-j):
            result[i] += A[j,i]*u[i+j]
            result[i+j] += A[j,i]*u[i]

    return result
```

Then we derive our Python base class with the Python matrix vector multiplication:

```
class py_band_matrix(band_matrix):
    def matvec(self,u):
        if self.shape[0] != u.shape[0]:
            raise ValueError("Dimension Missmatch!")

        return band_matvec_py(self.data,u)
```

But this is quite slow. We can alternativly implement this Inline with weave:

```
from numpy import array, zeros
from scipy.weave import converters
from scipy import weave

def band_matvec_inline(A,u):

    result = zeros(u.shape[0],dtype=u.dtype)

    N = A.shape[1]
    B = A.shape[0]

    code = """
    for(int i=0; i < N;i++)
    {
      result(i) = A(0,i)*u(i);
    }
    for(int j=1;j < B;j++)
    {

        for(int i=0; i < (N-j);i++)
        {
          if((i+j < N))
          {
            result(i) += A(j,i)*u(j+i);
            result(i+j) += A(j,i)*u(i);
          }

        }

    }
    """

    weave.inline(code,['u', 'A', 'result', 'N', 'B'],
                 type_converters=converters.blitz)
    return result
```

and create a new band matrix class:

```
class inline_band_matrix(band_matrix):
    def matvec(self,u):
        if self.shape[0] != u.shape[0]:
            raise ValueError("Dimension Missmatch!")

        return band_matvec_inline(self.data,u)
```

or we implement the matrix vector product with Cython:

```
cimport numpy as cnumpy
ctypedef cnumpy.float64_t reals #typedef_for easier reedding

def band_matvec_c(cnumpy.ndarray[reals,ndim=2] A,cnumpy.ndarray[reals,ndim=1] u):
    cdef Py_ssize_t i,j
    cdef cnumpy.ndarray[reals,ndim=1] result = numpy.zeros(A.shape[1],dtype=A.dtype)
    for i in xrange(A.shape[1]):
        result[i] = A[0,i]*u[i]

    for j in xrange(1,A.shape[0]):
        for i in xrange(A.shape[1]-j):
            result[i] = result[i] + A[j,i]*u[i+j]
            result[i+j] = result[i+j]+A[j,i]*u[i]

    return result
```

and make the new band-matrix class analogously:

```
class c_band_matrix(band_matrix):
   def matvec(self,u):
       if self.shape[0] != u.shape[0]:
           raise ValueError("Dimension Missmatch!")

       return band_matvec_c(self.data,u)
```

You can either import the band matrix base class to the *.pyx* file and define the derived Python class in the *.pyx* file, or `cimport` the function to a Python file with the new matrix class defined.

For a comparison, the algorithm implemented in C:

```
void band_matvec(float *result, float *A, float *x)
{
  unsigned int i,j;
  for(i = 0; i < cols; i++)
  {
    result[i] = A[i]*x[i];
  }

  for(j = 1; j < rows; j++)
  {
    for(i = 0; i < cols - j; i++)
    {
        result[i] += A[j + i*rows]*x[i+j];
        result[i+j] += A[j + i*rows]*x[i];

    }
  }
}
```

and a (direct) Matlab version:

```
function result = band_matvec_ma(A,u)

    [B N] = size(A);

    result = zeros(N,1);


    for i = 1:N
        result(i) = A(1,i)*u(i);
    end

    for j = 2:B
        for i =1:N
            if ((i+j-1) <= N)
              result(i) = result(i) + A(j,i)*u(i+j-1);
              result(i+j-1) = result(i+j-1) + A(j,i)*u(i);
            end
        end
    end
end
```

I made a time comparison on a Intel Core Duo with 1800MHz and 2 GB Ram. The dimension was ::math::`2^14`, and a with a band of the size of ::math::`2^6` under the diagonal. The reason for this choice was the limited texture memory on the card to compare it with the PyCUDA implementation lateron.

Language	time	Speedup
Python	2.96 s	1x
Matlab	90 ms	30x
C	30 ms	60x
Cython	12 ms	220x
Inline C++	10 ms	290x
C (-O2 compiler flag)	10 ms	290x
Cython (w. Boundscheck false)	6 ms	440x

Remarks:

- @Matlab: the for loops are bad rule is no longer true! The Java JIT (Just in Time) compilation vectorizes simple for loops like mine very efficiently, and that's also the reason why the Matlab version is 30 times faster than the Python implementation. There is a JIT compiler for Python too named Psyco. But Psyco doesn't optimize the numpy array access like Cython.
- @C The C implementation is rather straight forward and not well optimised. Of course C cannot be slower than Cython, because Cython is compiled to C.

There is a saying: "There are lies, there are big lies, and there are benchmarks", and my intention here is not to show that Python/Cython is the fastest language, but it is very easy to write fast Code in that languages, which can be faster than other implementations, if you don't optimize them well. That is a big Plus for Python/Cython, because when you write a prototype you want to get your code fast enough that you can work with it, and don't have to invest much time with optimizing it. Although well written Cython code is nearly as fast as a direct C implementation. If you don't need the last bit of speed out of your programs it is a good and simple alternative.

A more advanced example is an implementation in PyCUDA:

```
import pycuda.driver as cuda
import pycuda.gpuarray as gpuarray

import pycuda.autoinit
```

```
import numpy
from pycuda.compiler import SourceModule
from pycuda.driver import matrix_to_texref

from numpy import array, zeros, int32, float32, intp


sourceCodeTemplate = """
texture<float, 2> matrixTexture;

__global__ void gpu_band_matvec( //%(REALS)s *matrix,
                                %(REALS)s *vector,
                                %(REALS)s *result,
                                int *info
                                )
{
  // Infos about matrix dimension
  const int dim = info[0];
  const int band_with = info[1];

  // Infos about Blocks

  //unsigned int position = 0;
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  __shared__ %(REALS)s vector_help[%(DOUBLE_BLOCK_SIZE)s];
  %(REALS)s result_helper;

  //__syncthreads(); //Memory has to be loaded

  while(idx < dim +  %(BLOCK_SIZE)s * %(NUMBER_BLOCKS)s) //While loop over all blocks
  {

      __syncthreads(); //Memory has to be loaded

      if(idx < dim)
      {
      vector_help[threadIdx.x] = vector[idx];


      if(idx + %(BLOCK_SIZE)s < dim)
      {
        vector_help[threadIdx.x+%(BLOCK_SIZE)s] =  vector[idx+%(BLOCK_SIZE)s];
      }
      __syncthreads(); //Memory has to be loaded

      result_helper = tex2D(matrixTexture, 0, idx) * vector_help[threadIdx.x];

      for(int i = 1; i < band_with; i++)
      {
         result_helper += tex2D(matrixTexture, i, idx)*vector_help[threadIdx.x+i];

      }


     __syncthreads(); //Memory has to be loaded

     vector_help[threadIdx.x + %(BLOCK_SIZE)s] = vector_help[threadIdx.x];
```

```
      }

      if((idx - %(BLOCK_SIZE)s >= 0) && (idx - %(BLOCK_SIZE)s) < dim)
      {
        vector_help[threadIdx.x] = vector[idx - %(BLOCK_SIZE)s];
      }

      __syncthreads(); //Memory has to be loaded

      for(int i = 1; i < band_with; i++)
      {
         if((idx - i >= 0) && (idx - i < dim))
         {
           result_helper += tex2D(matrixTexture, i, idx - i)*
           vector_help[threadIdx.x + %(BLOCK_SIZE)s - i];
         }
      }

      if(idx < dim)
      {
        result[idx] = result_helper;
      }


     idx += %(BLOCK_SIZE)s * %(NUMBER_BLOCKS)s;

   }
}

"""

REALS = "float"
BLOCK_SIZE = 256
NUMBER_BLOCKS = 8

sourceCode = sourceCodeTemplate % {
'REALS': REALS,
'BLOCK_SIZE': BLOCK_SIZE,
'DOUBLE_BLOCK_SIZE': 2*BLOCK_SIZE,
'NUMBER_BLOCKS': NUMBER_BLOCKS,
}

matvec_module = SourceModule(sourceCode)

matvec_func = matvec_module.get_function("gpu_band_matvec")
matrixTexture = matvec_module.get_texref("matrixTexture")

from band_matrix import add_band_mat

class gpu_band_matrix:
  """variables for information about which matrix is
     currently on the texture
  """

  nr_matrices = 0
  active_matrix = 0
```

```
    """Takes a numpy array"""
    def __init__(self,data,set_right = False):
        self.data = data

        if set_right:
            for j in xrange(1,B):
                self.data[j:,N-j] = 0

        #self.data = gpuarray.to_gpu(data)
        self.shape = (data.shape[1],data.shape[1])
        self.band_with = data.shape[0]
        self.dtype = data.dtype

        info = array([data.shape[1],data.shape[0]]).astype(int32)

        self.gpu_info = gpuarray.to_gpu(info)

        cuda.matrix_to_texref(data, matrixTexture, order="F")

        gpu_band_matrix.nr_matrices += 1

        self.identity_nr = gpu_band_matrix.nr_matrices
        active_matrix = gpu_band_matrix.nr_matrices

    def matvec(self,x_gpu):
        if x_gpu.size != self.shape[0]:
            raise ValueError("Dimension mismatch!")

        #self.data.bind_to_texref_ext(matrixTexture,channels = 2)
        #cuda.matrix_to_texref(self.cpu_data, matrixTexture, order="F")

        if gpu_band_matrix.active_matrix != self.identity_nr:
            cuda.matrix_to_texref(self.data, matrixTexture, order="F")
            gpu_band_matrix.active_matrix = self.identity_nr

        y_gpu  = gpuarray.empty(x_gpu.size, x_gpu.dtype)
        matvec_func(intp(x_gpu.gpudata),intp(y_gpu.gpudata),
        self.gpu_info.gpudata, block = (BLOCK_SIZE,1,1), grid= (NUMBER_BLOCKS,1))
        return y_gpu

    def __add__(self,other):
        return add_band_mat(self,other)
```

I tested it on the GPU4U at the University of Graz and had the following times:

Language	time	Speedup
Python	2.87 s	1x
Cython	9.87 ms	270x
Cython (w. Boundscheck false)	5.56 ms	500x
PyCUDA (on GTX 280)	585 µs	5000x

Remark: PyCUDA is very fast, but has a breakeven point! For small dimensions Python or Cython are much faster.

At last I represent here an implementation of the CG algorithm. It was built in such a way that it accepts different forms of matrix vector products. For readability the gpu calculation was assumed to be a special case. But it could be avoided if the import statements had been named correctly.

```
# -*- coding: utf-8 -*-

import numpy, scipy
from scipy.sparse.linalg import LinearOperator
from scipy.sparse.linalg import cg
from numpy import dtype, array, reshape, vstack, hstack
from numpy import sqrt

def abstract_cg(A,b,x0 = None, inner_product = None, tol = 10e-5, maxiter = None,
                xtype = None, preconditioner = None, implicit_preconditioner = None,
                callback = None, nr_iterations = False, gpu = False):
    """
    This is an implementation of the CG algorithm on different inner product spaces
    It is assumed that the scalar product is positive definite.
    The preconditioner could either be a LinerOperator or a solver, depending on what
    is prefered.

    """
    if gpu:
        import pycuda.autoinit
        from pycuda.gpuarray import zeros
        from pycuda.gpuarray import dot as cu_dot
        dot = lambda x,y: cu_dot(x,y).get()*1.
    else:
        from numpy import zeros, dot

    #Prepare input
    #dim = A.shape[1]
    dim = b.size

    if maxiter is None:
        maxiter = dim*10

    if xtype is None:
        if x0 is None:
            xtype = A.dtype
        else:
            xtype = x0.dtype

    if x0 is None:
        x0 = zeros(dim,dtype = xtype)

    if inner_product is None:
        inner_product = dot

    x = zeros(x0.size,x0.dtype)
    x += x0

    if preconditioner is None and implicit_preconditioner is None:
        r = b - A.matvec(x0)
        p = r
        residuum_old = inner_product(r,r)

        for k in xrange(maxiter):
            Ap = A.matvec(p)

            checker = inner_product(p,Ap)
```

```
            if checker <= 0:
                raise ValueError("A not p.d.")

            alpha = residuum_old/inner_product(p,Ap)

            x = x + alpha*p

            if callback is not None:
                callback(x)

            r =  r - alpha*Ap
            residuum_new = inner_product(r,r)

            if sqrt(residuum_new) < tol:
                return x, k

            p = r + residuum_new/residuum_old*p
            residuum_old = residuum_new
        else:
            print "Maxiter reached!"
            return x, (k+1)

    if preconditioner is not None:

        r = b - A.matvec(x0)
        v = preconditioner.matvec(r)
        p = v

        residuum_old = inner_product(v,r)


        for k in xrange(maxiter):


            s = A.matvec(p)

            sigma = inner_product(s,p)
            if sigma <= 0:
                raise ValueError("A not p.d.")

            alpha = residuum_old/sigma

            x += alpha*p
            r -= alpha*s
            v = preconditioner.matvec(r)

            residuum_new = inner_product(v,r)

            if sqrt(residuum_new) < tol:
                return x, k

            beta = residuum_new/residuum_old
            p = v + beta*p

            residuum_old = residuum_new

        else:
```

```
            print "Maxiter reached!"
            return x, (k+1)

    if implicit_preconditioner is not None:
        raise NotImplementedError("preconditioning not implemented yet!")
```

Links

1.10 Licenses

To avoid the risk to get sued for nothing... but since everything is open, there are no costs except copying them.

1.10.1 License

(e) "License Elements" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

(f) "Licensor" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

(g) "Original Author" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

(h) "Work" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

(i) "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

(j) "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

(k) "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

   (a) to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

   (b) to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

(c) to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

(d) to Distribute and Publicly Perform Adaptations.

(e) **For the avoidance of doubt:**

i. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

iii. Voluntary License Schemes. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

1. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

(a) You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

(b) You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

(c) If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

(d) Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

2. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

1. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

2. Termination

   (a) This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

(b) Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

3. Miscellaneous

   (a) Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   (b) Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   (c) If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   (d) No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

   (e) This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

   (f) The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

   Creative Commons Notice

   Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

   Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of the License.

   Creative Commons may be contacted at http://creativecommons.org/.

PSF LICENSE AGREEMENT FOR PYTHON 1.0

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.0 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.0 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright Â© 2001-2010 Python Software Foundation; All Rights Reserved" are retained in Python 1.0 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.0 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.0.
4. PSF is making Python 1.0 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.0 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.0, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 1.0, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CHAPTER

TWO

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*