hpc Documentation

Release 1.0

Stefan Reiterer

January 27, 2011

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION AND PERSONAL THOUGHTS

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.

Actually to learn Sphinx and to make it possible for other to get a quick and dirty reference for working 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. I had the problem, that every time I implement a new numerical scheme I have 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 But I try to make them compatible with the new version so that this document don't get outdated soon.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.

Stefan Reiterer, Graz Austria 2011

1.2 The Zen of 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 some words are really true, especially the statements of simpleness. Simply keep this rules in mind.

1.3 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 learnt during the last 13 years since I'm started "programming" Visual Basic with 13.

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.3.4 Premature optimisation is the rule 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...

1.3.5 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 like Git [2], Mercurial [3] (which is written in Python), or SVN [4].

1.3.6 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 [5], or projects like Wubi [6]. And thanks to Distributions like Ubuntu [7] and it's many derivatives (I use Kubuntu), or open SUSE [8] using Linux is nowadays possible for normal humans too.

[1] http://en.wikiquote.org/wiki/Donald_Knuth
[2] http://git-scm.com/
[3] http://mercurial.selenic.com/
[4] http://en.wikipedia.org/wiki/Apache_Subversion
[5] http://www.virtualbox.org/
[6] http://www.ubuntu.com/desktop/get-ubuntu/windows-installer
[7] http://www.ubuntu.com/
[8] http://www.opensuse.org/de/

1.3.7 Not everything from Extreme Programming is that bad

It is shown in many tests that applying the whole concept of XP [9], 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.

1.3.8 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.

1.3.9 If your programs aren't understandable nobody will use them

...including yourself.

1.3.10 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.

Links

[9] http://www.extremeprogramming.org/

CHAPTER

TWO

ABOUT PYTHON

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 [1]. 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 optimization 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.

2.2 Why Python?

- Intuitive Syntax
- Simple
- An easy to learn language.
- Object orientated.
- Fast (if used with brains).
- Rapidly developing.
- 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)

2.3 Get Python

The programs and packages used here are all open source, so they can be obtained freely. Most Linux distributions already ship Python, because many scripts are written in Python. See also the *Python* project page for further information [2].

[1] http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html
[2] http://www.python.org/

An easy way to obtain Python is to install *Sagemath* [3], which contains many useful packages extensions and packages for mathematics.

Another possibility would be *FEMhub* which is a fork of *Sage* [4] . 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 [5].

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

[3] http://www.sagemath.org/
[4] http://www.femhub.org
[5] http://code.google.com/p/computational-sage/

CHAPTER

THREE

HOW TO USE PYTHON

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
```

CHAPTER

FOUR

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*