Slides from INF3331 lectures

Hans Petter Langtangen, Ola Skavhaug and Joakim Sundnes

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2014

About this course

Teachers (1)

- Joakim Sundnes (sundnes@simula.no)
- Jonathan Feinberg
- Possible guest lecturers (TBD)
- We use Python to create efficient working (or problem solving) environments
- We also use Python to develop large-scale simulation software (which solves partial differential equations)
- We believe high-level languages such as Python constitute a promising way of making flexible and user-friendly software!
- Some of our research migrates into this course
- There are lots of opportunities for Master projects related to this course

Teachers (2)

- Most examples are from our own research; involves some science and/or mathematics!
- Very little mathematics knowledge is needed to complete the course
- Treating mathematical software as a “black box” without fully understanding the contents is a useful exercise
- Translating simple mathematical expressions to computer code is highly relevant for many applications

Contents

- Scripting in general
- Basic Bash programming
- Quick Python introduction for beginners (two weeks)
- Regular expressions
- Python problem solving
- Efficient Python with vectorization and NumPy arrays
- Combining Python with C, C++ and Fortran
- Useful tools; distributing Python modules, documenting code, version control, testing and verification of software
- Creating web interfaces to Python scripts

What you will learn

- Scripting in general, but with most examples taken from scientific computing
- Jump into useful scripts and dissect the code
- Learning by doing
- Find examples, look up man pages, Web docs and textbooks on demand
- Get the overview
- Customize existing code
- Have fun and work with useful things

Background 1; INF3331 vs INF1100

- In 2011, about 50% of INF3331 students had INF1100, about 33% in 2012 and 2013
- Wide range of backgrounds with respect to Python and general programming experience
- Since INF3331 does not build on INF1100, some overlap is inevitable
- Two weeks of basic Python intro not useful for those with INF1100 background
- INF3331 has more focus on scripting and practical problem solving
- We welcome any feedback on how we can make INF3331 interesting and challenging for students with different backgrounds

Background 2; mathematics

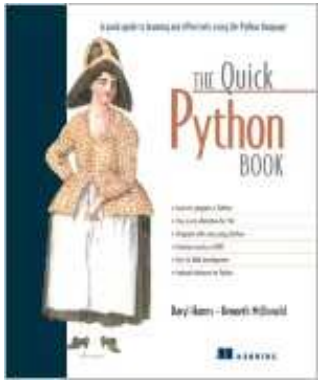
- Very little mathematics is needed to complete the course.
- Basic knowledge will make life easier;
  - General functions, such as $f(x) = ax + b$, and how they are turned into computer code
  - Standard mathematical functions such as $\sin(x), \cos(x)$ and exponential functions
  - Simple matrix-vector operations
- A learn-on-demand strategy should work fine, as long as you don't panic at the sight of a mathematical expression.
- Matlab is commonly cited as code examples, since this is a *de facto* standard for scientific computing.

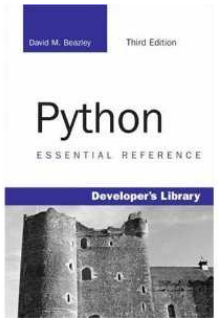
Teaching material (1)

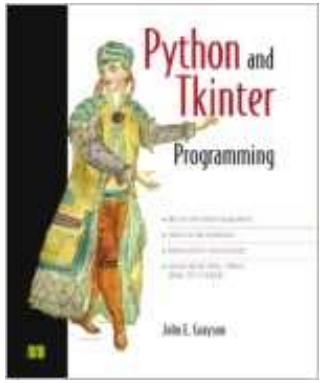
- Slides from lectures (by Sundnes, Skavhaug, Langtangen et al).
  A preliminary version is found here
  `http://www.uio.no/studier/emner/matnat/ifi/INF3331/h14/inf3331_h14.pdf`
  Do not print these slides now! Will be substantially updated through the fall.
- H.P. Langtangen and G. K. Sandve: *Illustrating Python via Bioinformatics Examples*, download from
  `http://hplgit.github.io/bioinf-py/doc/tutorial/bioinf-py.pdf`
- Associated book (optional):
  H. P. Langtangen: *Python Scripting for Computational Science*, 3rd edition, Springer 2008
- You must find the rest: manuals, textbooks, google

Teaching material (2)

- Good Python litterature:
  Harms and McDonald: The Quick Python Book
  (tutorial+advanced)
  Beazley: Python Essential Reference
  Grayson: Python and Tkinter Programming







Lectures and groups (1)

- Lectures Tuesdays 12.15-14.00
- Groups Thursday 12.15-14, Thursday 14.15-16, Friday 10.15-12
- A tentative lecture plan will be online shortly
- Slides will be updated as we go. Printing the entire pdf file in August is not recommended.
- Updated slides will be available before each lecture
- Source code will normally be available after the lecture
- Groups and exercises are the core of the course; problem solving is in focus.

Lectures and groups (2)

- August 19th:
  - Intro/motivation; scripting vs regular programming
  - “User survey”
- August 26th:
  - Basic shell scripting
- September 2nd & 9th:
  - Python introduction (not needed if you have INF1100)
- September 16th:
  - Regular expressions

Group classes anno 2013 (1)

- There used to be no regular “group classes” in INF3331
- Groups were for correcting and marking weekly assignments.
- To get a weekly assignment approved;
  - Show up at the group with a print of the script(s)
  - Have the assignment approved by another student
  - Hand in the assignment electronically (in Devilry) by Friday

Group classes anno 2013 (2)

Three alternative course paths:

1. 75% of weekly assignments approved (60 points out of 80)
2. 37.5% of weekly assignments (30 points) + small project (approximately 32 hrs)
3. No weekly assignments, large project (64 hrs)

+ written exam for everyone.

Why has the course been organized like this?

- “Problem solving” is best learnt by solving a large number of problems
- With limited resources, this is the only way we can maintain the large number of mandatory assignments
- You learn from reading and inspecting eachother’s code

Group classes anno 2014

Final details TBD, but here's a rough plan:

- No strict requirement to show up in group classes to get an assignment approved.
- Most likely a reward system, where showing up and correcting assignments gives you extra points.

Goal; more flexible implementation, but which still allows a high volume of programming exercises. Any feedback or suggestions;

`sundnes@simula.no`

Software for this course

- Python runs on Windows, Mac, Linux.
- I have no experience with Windows and very limited experience with Python on Mac
- I recommend Ubuntu Linux, either running natively or in a virtual machine.
- Follow the instructions for INF1100:
  `http://heim.ifi.uio.no/ inf1100/installering.html`

Python 2 vs Python 3

- Python 3.3 is the newest stable version
- Python 2.7 is still widely used
  - Default on Mac OS X
  - Many libraries are still based on Python 2.7
- This course:
  - 2012 - Python 2.7
  - 2013 - Mix of Python 2.7 and 3.3
  - 2014 - Python 3.3 (but look out for bugs in slides!)
- Small difference for the scope of this course, but watch out for widely used functions such as `print`, `open`, `input`, `range`, and integer division.

Scripting vs regular programming

What is a script?

- Very high-level, often short, program
  written in a high-level scripting language
- Scripting languages: Unix shells, Tcl, Perl, Python, Ruby,
  Scheme, Rexx, JavaScript, VisualBasic, ...
- This course: Python
  + a taste of Bash (Unix shell)

Characteristics of a script

- Glue other programs together
- Extensive text processing
- File and directory manipulation
- Often special-purpose code
- Many small interacting scripts may yield a big system
- Perhaps a special-purpose GUI on top
- (Sometimes) portable across Unix, Windows, Mac
- Interpreted program (no compilation+linking)

Why not stick to Java or C/C++?

Features of scripting languages compared with Java, C/C++ and Fortran:

- shorter, more high-level programs
- much faster software development
- more convenient programming
- you feel more productive

Three main reasons:

- no variable declarations,
  but lots of consistency checks at run time
- easy to combine software components and interact with the OS
- lots of standardized libraries and tools

Scripts yield short code

- Consider reading real numbers from a file, where each line can contain an arbitrary number of real numbers:

```
1.1  9   5.2
1.762543E-02
0 0.01 0.001

9 3 7
```

- Python solution:

```
F = open(filename, 'r')
n = F.read().split()
```

Using regular expressions (1)

- Suppose we want to read complex numbers written as text

```
(-3, 1.4)  or  (-1.437625E-9, 7.11)  or  (  4, 2 )
```

- Python solution:

```
import re
m = re.search(r'\(\s*([^,]+)\s*,\s*([^,]+)\s*\)',
              '(-3,1.4)')
re, im = [float(x) for x in m.groups()]
```

(This will only find the first match of the regular expression, use `re.findall` to return a list of all matches.)

Using regular expressions (2)

- Regular expressions like

  ```
  \(\s*([^,]+)\s*,\s*([^,]+)\s*\)
  ```

  constitute a powerful language for specifying text patterns
- Doing the same thing, without regular expressions, in Fortran and C requires quite some low-level code at the character array level
- Remark: we could read pairs (-3, 1.4) without using regular expressions,

  ```
  s = '(-3,  1.4 )'
  re, im = s[1:-1].split(',')
  ```

Script variables are not declared

- Example of a Python function:

```
def debug(leading_text, variable):
    if os.environ.get('MYDEBUG', '0') == '1':
        print leading_text, variable
```

- Dumps any printable variable
  (number, list, hash, heterogeneous structure)
- Printing can be turned on/off by setting the environment variable `MYDEBUG`

The same function in C++

- Templates can be used to mimic dynamically typed languages
- Not as quick and convenient programming:

```
template <class T>
void debug(std::ostream& o,
           const std::string& leading_text,
           const T& variable)
{
  char* c = getenv("MYDEBUG");
  bool defined = false;
  if (c != NULL) {  // if MYDEBUG is defined ...
    if (std::string(c) == "1") {  // if MYDEBUG is true ...
      defined = true;
    }
  }
  if (defined) {
    o <<  leading_text << " " << variable << std::endl;
  }
}
```

The relation to OOP

- Object-oriented programming can also be used to parameterize types
- Introduce base class `A` and a range of subclasses, all with a (virtual) print function
- Let `debug` work with `var` as an `A` reference
- Now `debug` works for all subclasses of `A`
- Advantage: complete control of the legal variable types that `debug` are allowed to print (may be important in big systems to ensure that a function can only make transactions with certain objects)
- Disadvantage: much more work, much more code, less reuse of `debug` in new occasions

Flexible function interfaces (1)

- User-friendly environments (Matlab, Maple, Mathematica, S-Plus, ...) allow flexible function interfaces
- Novice user:

```
# f is some data
plot(f)
```

- More control of the plot:

```
plot(f, label='f', xrange=[0,10])
```

- More fine-tuning:

```
plot(f, label='f', xrange=[0,10], title='f demo',
     linetype='dashed', linecolor='red')
```

Flexible function interfaces (2)

- In C++, some flexibility is obtained using default argument values, e.g.,

```
void plot(const double[]& data, const char[] label='',
const char[] title = '', const char[] linecolor='black')
```

  Limited flexibility, since the order of arguments is significant.

- Python uses keyword arguments = function arguments with keywords and default values, e.g.,

```
def plot(data, label='', xrange=None, title='',
         linetype='solid', linecolor='black', ...)
```

- The sequence and number of arguments in the call can be chosen by the user

Classification of languages (1)

- Many criteria can be used to classify computer languages
- Dynamically vs statically typed languages
  Python (dynamic):

```
c = 1              # c is an integer
c = [1,2,3]        # c is a list
```

  C (static):

```
double c; c = 5.2;    # c can only hold doubles
c = "a string..."     # compiler error
```

Classification of languages (2)

- Weakly vs strongly typed languages
  Perl (weak):

```
$b = '1.2'
$c = 5*$b;   # implicit type conversion: '1.2' -> 1.2
```

  Python (strong):

```
import math
b = '1.2'
c = 5*b      #legal, but probably not the result you want
c = math.exp(b)  #illegal, no implicit type conversion
c = math.exp(float(b)) #legal
```

Classification of languages (3)

- Interpreted vs compiled languages
- Dynamically vs statically typed (or type-safe) languages
- High-level vs low-level languages (Python-C)
- Very high-level vs high-level languages (Python-C)
- Scripting vs system languages

Turning files into code (1)

- Code can be constructed and executed at run-time
- Consider an input file with the syntax

```
a = 1.2
no of iterations = 100
solution strategy = 'implicit'
c1 = 0
c2 = 0.1
A = 4
```

- How can we read this file and define variables `a`, `no_of_iterations`, `solution_strategi`, `c1`, `c2`, `A` with the specified values?

Turning files into code (2)

- The answer lies in this short and generic code:

```
file = open('inputfile.dat', 'r')
for line in file:
    # first replace blanks on the left-hand side of = by _
    variable, value = line.split('=').strip()
    variable = re.sub(' ', '_', variable)
    exec(variable + '=' + value)   # magic...
```

- This cannot be done in Fortran, C, C++ or Java!

Scripts can be slow

- Perl and Python scripts are first compiled to byte-code
- The byte-code is then *interpreted*
- Text processing is usually as fast as in C
- Loops over large data structures might be very slow

```
for i in range(len(A)):
    A[i] = ...
```

- Fortran, C and C++ compilers are good at optimizing such loops at compile time and produce very efficient assembly code (e.g. 100 times faster)
- Fortunately, long loops in scripts can easily be migrated to Fortran or C

Scripts may be fast enough

Read 100 000 (x,y) data from file and
write (x,f(y)) out again

- Pure Python: 4s
- Pure Perl: 3s
- Pure Tcl: 11s
- Pure C (fscanf/fprintf): 1s
- Pure C++ (iostream): 3.6s
- Pure C++ (buffered streams): 2.5s
- Numerical Python modules: 2.2s (!)
- Remark: in practice, 100 000 data points are written and read in binary format, resulting in much smaller differences

When scripting is convenient (1)

- The application's main task is to connect together existing components
- The application includes a graphical user interface
- The application performs extensive string/text manipulation
- The design of the application code is expected to change significantly
- CPU-time intensive parts can be migrated to C/C++ or Fortran

When scripting is convenient (2)

- The application can be made short if it operates heavily on list or hash structures
- The application is supposed to communicate with Web servers
- The application should run without modifications on Unix, Windows, and Macintosh computers, also when a GUI is included

When to use C, C++, Java, Fortran

- Does the application implement complicated algorithms and data structures?
- Does the application manipulate large datasets so that execution speed is critical?
- Are the application's functions well-defined and changing slowly?
- Will type-safe languages be an advantage, e.g., in large development teams?

Some personal applications of scripting

- Get the power of Unix also in non-Unix environments
- Automate manual interaction with the computer
- Customize your own working environment and become more efficient
- Increase the reliability of your work
  (what you did is documented in the script)
- Have more fun!

Some business applications of scripting

- Python and Perl are very popular in the open source movement and Linux environments
- Python, Perl and PHP are widely used for creating Web services (Django, SOAP, Plone)
- Python and Perl (and Tcl) replace 'home-made' (application-specific) scripting interfaces
- Many companies want candidates with Python experience

What about mission-critical operations?

- Scripting languages are free
- What about companies that do mission-critical operations?
- Can we use Python when sending a man to Mars?
- Who is responsible for the quality of products?

The reliability of scripting tools

- Scripting languages are developed as a world-wide collaboration of volunteers (open source model)
- The open source community as a whole is responsible for the quality
- There is a single repository for the source codes (plus mirror sites)
- This source is read, tested and controlled by a very large number of people (and experts)
- The reliability of *large* open source projects like Linux, Python, and Perl appears to be very good - at least as good as commercial software

Practical problem solving

- Problem: you are not an expert (yet)
- Where to find detailed info, and how to understand it?
- The efficient programmer navigates quickly in the jungle of textbooks, man pages, README files, source code examples, Web sites, news groups, ... and has a gut feeling for what to look for
- The aim of the course is to improve your practical problem-solving abilities
- *You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program* (Alan Perlis)

Group classes and assignments 2014

- Group classes start September 1st
- For 2014, weekly assignments are replaced by a smaller number of larger mandatory assignments
- Help with mandatory assignments is offered in group classes and some of the lectures

Mandatory assignments

- Each assigment contains multiple steps
- A Latex-written report is to handed in with each assignment
- First deadline: Friday 19 Sept
- All material is handed in via github
- Everyone needs to obtain a github account

Marking and approval of mandatory assignements

- After the assignment is handed in, you are given a *marking group* (rettegruppe) with three students in each
- Each marking group will get the assigments from one other group for evaluation
- For each assignment you evaluate, you should write a short report (10-20) lines, which comments on the work done
- When, where and how you organize the evaluation is up to you
- Group teachers approve assigments based on the reports
- Deadline for reports; one week after you have been assigned to the marking groups and have received the assignments.

Group classes and lectures

- Group classes for 2014 will be regular group classes, where you work individually and can ask questions about the mandatory assignments
- A number of the lectures will also be used for programming labs:
  - More suitable for the topics of the course (i.e. problem solving)
  - “Flipped classroom”
  - Based on feedback from previous students
- Some lectures are used for programming lab; you will need to bring a laptop.

Using github

- Everyone needs to obtain an account on github. This will be used for handing in marking of assignments
- We will establish a *classroom* on github. For this we need from everyone:
  - Full name
  - Email
  - Github user name
- A web form will be emailed to everyone, for handing in this information.

For more information, see the course web page

- Available info:
  - Lecture plan until 16/9
  - General info on assignments
  - Template for latex report
- Later this week:
  - More details on assignments
  - Info on the github classroom

Questions: sundnes@simula.no

Using git

Why use git or other version control systems

- Can retrieve old versions of files
- Can print history of incremental changes
- Very useful for programming or writing teams
- Contains an official repository
- Programmers work on *copies* of repository files
- Conflicting modifications by different team members are detected
- Can serve as a backup tool as well
- So simple to use that there are no arguments against using version control systems!

Some git commands

- git: a modern version control system, similar to mercurial, bazaar, svn, cvs etc.
- See `http://git-scm.com, http://github.com`
- `git clone URL`: clone a (remote) repository
- `git init`: create a (local) repository
- `git commit -a`: check files into the repository
- `git rm`: remove a file
- `git mv`: move/rename a file
- `git pull`: update file tree from (remote) repository
- `git push`: push changes to central repository
- And much more, see `git help`

git example 1

```
git clone git://github.com/git/hello-world.git
cd hello-world
(edit files)
git commit -a -m 'Explain what I changed'
git format-patch origin/master
(update from central repository:)
git pull
```

git example 2

```
cd src
git init
git add .
(edit files)
git commit -a -m 'Explain what I changed'
(accidentally remove/edit file.tmp)
git checkout file.tmp
```