About INF3331/INF4331

Hans Petter Langtangen[1,2] Joakim Sundnes[1,2]
Ola Skavhaug[3] Simon Funke[1,2]

Center for Biomedical Computing, Simula Research Laboratory[1]

Dept. of Informatics, UiO[2]

Expert Analytics[3]

Aug 24, 2016

About this course

Teachers (1)

- Simon Funke (simon@simula.no)
- Joakim Sundes, Karl-Erik Holter
- Guest lecturers (to be announced)

What we use Python for:

- We use Bash/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!
- 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 and C++
- Creating web interfaces to Python scripts
- Programming tools: version control systems, distributing Python modules, debugging and profiling, documenting code, testing and verification of software

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

- 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: INF3331 vs INF4331

- Same lectures and group sessions.
- INF4331 has more assignments compared to INF3331.
- More points are needed to pass INF4331.

Background 3: 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 do not 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 Langtangen, Sundes, Skavhaug, Funek et al) on the INF3331 web page (here).
- H.P. Langtangen and G. K. Sandve: Illustrating Python via Bioinformatics Examples: PDF
- Associated book (optional): H. P. Langtangen: Python Scripting for Computational Science, 3rd edition, Springer 2008

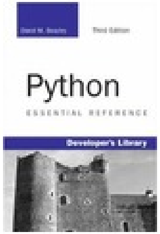
You must find the rest: manuals, textbooks, google

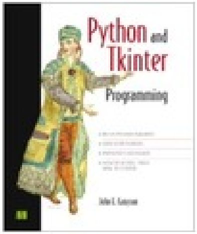
Note that some literature is based to Python 2, while we will use Python 3 in this course (but the differences are minimal for this course).

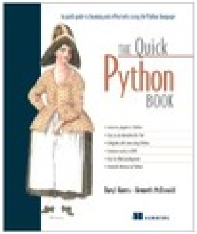
Teaching material (2)

Good Python literature:

- Harms and McDonald: The Quick Python Book, 2nd edition, 2010
- Beazley: Python Essential Reference, 4th edition, 2009
- Mark Pilgrim: Dive into Python 3, 2004







Lectures and groups (1)

- Lectures Wednesday 10:15 – 12:00.
- Lecture slides available on the course website before each lecture.
- Assignments will be available after the lecture.
- Weekly groups sessions (see course website).
- Groups and exercises are the core of the course: problem solving is in focus.
- In addition, we use the Piazza forum to ask and answer questions and discuss problems.

Lectures and groups (2)

- August 24th: Introduction; scripting vs regular programming, version control systems
- August 31th: Basic shell scripting
- September 7th & 14th: Python introduction (not needed if you have INF1100)
- September 12th: Regular expressions
- ...

Group classes and assignments

- Mix of short (1 week) and large (3+1 weeks) assignments.
- Deadlines are published on course website.
- Group classes will be regular group classes, where you work individually and can ask questions about the mandatory assignments.
- Group classes start this week.
- No strict requirement to show up in group classes, but useful to ask questions and discuss solutions.

Assignments and reviews

- Each student solves the assignments individually.
- Large assignments have a peer review system to correct the assignments.
- Assignments and reviews are handed in electronically on github.
- All assignments and reviews are corrected by INF3331 correctors and give points towards passing the course.
- First assignment: today.

Peer-review of large assignments

- After the assignment is handed in, you are given a *marking group* (rettegruppe) with three students in each.
- Each marking group will get the assignments from three other students.
- For each assignment you evaluate, you should write a report, which comments and evaluates the quality of the programming, and provide constructive feedback.
- When, where and how you organize the evaluation is up to you, but the intention is to do the evaluation in groups.
- Deadline for reports: one week after you have been assigned to the marking groups.

Marking

- No exam.
- The assignments give points towards passing the course.
- The peer-review report also gives points.
- Course passed with $\geq$ 80% of total points at the end of the semester.

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 each other’s code.
- Goal: more flexible implementation, but which still allows a high volume of programming exercises.

Software for this course

- Bash runs on Windows (via cygwin), Mac, Linux.
- Python runs on Windows (e.g. Enthought), Mac, Linux.
- C (GCC) runs on Windows, Mac, Linux.
- I recommend Ubuntu Linux, either running natively or in a virtual machine.
- I have limited experience with Python/Bash/C on Windows.
- Follow the instructions for INF1100 (but install Python 3)

Python 2 vs Python 3

- Python 3.5 is the newest stable version.
- Most libraries have now Python 3 support.
- Python 2.7 is still around, and the default on Mac OS X and some Linux distributions.
- This course:
    - 2016 - Python 3 (3.4/3.5)
    - 2015 - Mostly 2.7, some 3.3/3.4
    - 2014 - Mostly 2.7, some 3.3/3.4
    - 2013 - Mix of Python 2.7 and 3.3
- 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 and automate tasks
- 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 to Java, C/C++ and Fortran:

- shorter, more high-level programs
- much faster software development
- more convenient programming
- you feel more productive. The three main reasons are:
  - no variable declarations, but lots of consistency checks at run time
  - technical details are hidden: no pointers, automatic garbage collection, ...
  - 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:

```python
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):

```python
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)
- 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_strategy`, `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('=')
    variable = variable.strip()  # strip leading and trailing blanks
    variable = re.sub(' ', '_', variable)
    exec(variable + '=' + value)    # magic...
```

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

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, Ruby and Javascript are widely used for creating Web services (Django, 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)

Version control systems

.

Version control systems

What are version control systems good for?

- Manage changes of files such as scripts, source code, documents, ...
- Can retrieve old versions of files.
- Can print history of incremental changes.
- Allows collaborative programming.
- Can serve as a backup tool.

Core idea:

- A version controlled system (typically) contains one official repository.
- Programmers work on *copies* of repository files and upload the changes to the official repository.
- Non-conflicting modifications by different team members are merged automatically.
- Conflicting modifications are detected and must be resolved manually.

The git version control system

- git: a modern version control system, similar to mercurial, bazaar, svn, cvs etc.
- Developed by Linus Torvalds to improve the development of the Linux kernel and now one of the most succesfull version control system.
- Git is complex and powerful - start with the basics and learn as you go.
- There exist a good ecosystem for storing git repositories online (e.g. github and bitbucket).
- Official git webpage `http://git-scm.com`

Creating your first git repository

A git repository is a folder in which files can be tracked by git. A git repository is created with:

```
$ cd src
$ git init .  # The src folder is now also a git repository
```

Files need to be added to the repository in order to track their changes:

```
# create myfile.py and some text files
$ git add myfile.py *.txt
```

Commit all tracked files into a new version:

```
$ git commit -a -m 'This is my first commit'
```

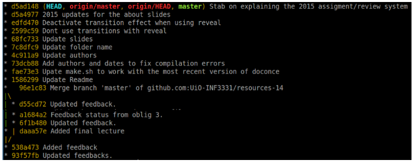
Make some changes and create a new commit.

```
# edit myfile.py
$ git status # list files that have been changed since last commit
$ git diff   # show the detailed changes since the last commit
$ git commit -a -m 'Explain what I changed'
```

Reading history and going backwards in time

View history with commit messages with:

```
$ git log
$ git log --graph --oneline --all --decorate
```



```
* d5ad148 (HEAD, origin/master, origin/HEAD, master) Stab on explaining the 2015 assigment/review system
* d5a4977 2015 updates for the about slides
* edfd470 Deactivate transition effect when using reveal
* 2599c59 Dont use transitions with reveal
* 68fc733 Update slides
* 7c8dfc9 Update folder name
* 4c911a9 Update authors
* 73dcb88 Add authors and dates to fix compilation errors
* fae73e3 Upate make.sh to work with the most recent version of doconce
* 1586299 Update Readme
*   96e1c83 Merge branch 'master' of github.com:UiO-INF3331/resources-14
|\
| * d55cd72 Updated feedback.
| * a1684a2 Feedback status from oblig 3.
| * 6f1b480 Updated feedback.
* | daaa57e Added final lecture
|/
* 538a473 Added feedback
* 93f57fb Updated feedbacks.
```

Each version has a unique hash key. This hash can be used to obtain an older version of a file:

```
$ git checkout 664250addc1a23c9e8db2c53e203ea2ef9c7c9fc myfile.py
```

Or to go back to the latest version of a file:

```
rm myfile.py              # "accidentaly" remove myfile.py
git checkout myfile.py    # restore latest version
```

Remote repositories

We can work on git repositories that live on a remote location (for collaboration and backup).

Clone a remote repository to a local repository:

```
git clone git@github.com:UiO-INF3331/INF3331-simonwf.git
cd INF3331-simonwf
```

Create a new commit and push it to the remote repository.

```
(edit files)
git commit -a -m 'Explain what I changed'
git push origin master    # Requires write permission
                          # on the remote repository
```

Download changes from remote repository:

```
git pull # This might result merge conflicts
         # which need to be resolved manually (see exercise).
```

Most important git commands

- `git clone URL`: clone a (remote) repository
- `git init`: create a new (local) repository
- `git add`: add a file
- `git rm`: remove a file
- `git mv`: move/rename a file
- `git status`: View status of commited/uncommited files
- `git commit -a`: commit all changed files
- `git commit FILENAME`: commit a specific file
- `git pull`: update file tree from (remote) repository
- `git push`: push changes to central repository

For more information about the commands use `git help COMMAND`

Using github

- Everyone needs to obtain an account on github (its free).
- We will establish a *classroom* on github. For this we need from everyone:
  - Full name
  - Email
  - UiO username
  - Github user name

  For this, please fill out the web form on he course webpage.
- We will create a git repository on github where you can upload the assignemnt solutions.

Things to do in the first week

- Sign up to INF3331/INF4331 course on Piazza.
- Create github account and fill out webform (follow instructions here)
- (Optional) Do the online, interactive git tutorial on `https://try.github.io`
- Solve the first assignment.

Final words

- For more information, see the course web page.
- Use Piazza to ask, and answer questions.
- For feedback email me: simon@simula.no