Slides from INF3331 - Course intro

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2012

About this course

Teachers

- Joakim Sundnes
- Glenn Lines
- 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

Contents

- Scripting in general
- Basic Bash programming
- Quick Python introduction (two weeks)
- Python problem solving
- More advanced Python (class programming++)
- Regular expressions
- Combining Python with C, C++ and Fortran
- The Python C API and the NumPy C API
- Distributing Python modules (incl. extension modules)
- Verifying/testing (Python) software
- Documenting Python software
- Optimizing Python code
- Python coding standards and ’Pythonic’ programming

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

INF3331 vs INF1100

- In 2011, about 50% of INF3331 students had INF1100
- Wide range of backgrounds with respect 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

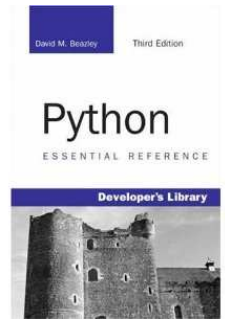
Teaching material (1)

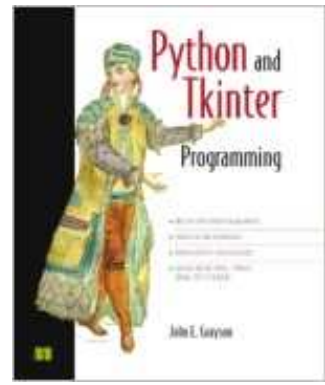
- Slides from lectures
  (by Skavhaug, Sundnes, Langtangen et al), download from
  `http://www.uio.no/studier/emner/matnat/ifi/INF3331/h12/inf3331.pdf`
- Associated book (for the Python material):
  H. P. Langtangen: *Python Scripting for Computational Science*, 3rd edition, Springer 2008

- 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 Mondays 12.15-14.00
- Groups Tuesday 12.15, Wednesday 08.15, (Thursday 12.15)
- Slides will be updated as we go. Printing the entire pdf file in August is not recommended.
- Topics for the lecture, updated slides and page numbers will be made available approximately one week before each lecture.
- Groups and exercises are the core of the course; problem solving is in focus.

Lectures and groups (2)

- August 20th:
  - “User survey”
  - Intro/motivation; scripting vs regular programming
- August 27th:
  - Basic Bash scripting
  - September 3rd & 10th:
  - Python introduction (not needed if you have INF1100)

Lectures and groups (3)

Three alternative course paths:

1. 75% of weekly assignments approved
2. 37.5% of weekly assignments + small project (approximately 32 hrs)
3. No weely assignments, large project (64 hrs)

+ written exam for everyone.

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
- 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

Two main reasons:

- no variable declarations,
  but lots of consistency checks at run time
- 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:

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

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):

```
b = '1.2'
c = 5*b      # illegal; no implicit type conversion
c = 5*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
c3 = StringFunction('A*sin(x)')
```

- How can we read this file and define variables `a`, `no_of_iterations`, `solution_strategi`, `c1`, `c2`, `A` with the specified values?
- And can we make `c3` a function `c3(x)` as specified?

Yes!

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 (1)

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

Scripts may be fast enough (2)

Read a text in a human language and generate random nonsense text in that language (from "The Practice of Programming" by B. W. Kernighan and R. Pike, 1999):

Language	CPU-time	lines of code
C	0.30	150
Java	9.2	105
C++ (STL-deque)	11.2	70
C++ (STL-list)	1.5	70
Awk	2.1	20
Perl	1.0	18

Machine: Pentium II running Windows NT

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)