Slides from INF3331
- Python and course intro

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2011

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
- Quick Python introduction (first 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
- Basic Bash 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

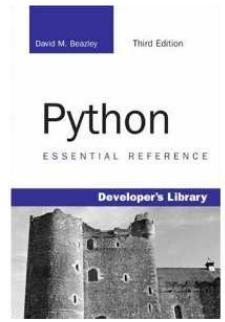
Teaching material (1)

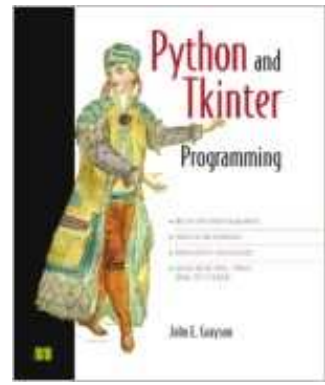
- Slides from lectures
  (by Skavhaug, Sundnes, Langtangen et al), download from
  `http://www.uio.no/studier/emner/matnat/ifi/INF3331/h11/inf3331.pdf`
- Associated book (for the Python material):
  H. P. Langtangen: *Python Scripting for Computational Science*, 3rd edition, Springer 2008

- 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 Tuesdays 10.15-12.00
- Groups Thursday 14.15, Monday 10.15, (Friday 10.15)
- 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 one week before each lecture.
- Groups and exercises are the core of the course; problem solving is in focus.

Lectures and groups (2)

- Tuesday 23rd:
  - “User survey”
  - Intro/motivation; scripting vs regular programming
- Tuesday 30th:
  - First encounter with Python

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 allow make transactions with certain objects)
- Disadvantage: much more work, much more code, less reuse of `debug` in new occasions

Flexible function interfaces

- 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')
```

Keyword arguments

- 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
```

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)

Basic Python Constructs

First encounter with Python

```
#!/usr/bin/env python

from math import sin
import sys

x = float(sys.argv[1])
print "Hello world, sin(%g) = %g." % (x, sin(x))
```

Running the Script

Code in file `hw.py`.
Run with command:

```
> python hw.py 0.5
Hello world, sin(0.5) = 0.479426.
```

Linux alternative if file is executable (`chmod a+x hw.py`):

```
> ./hw.py 0.5
Hello world, sin(0.5) = 0.479426.
```

Quick Run Through

On *nix; find out what kind of script language (interpreter) to use:

```
#!/usr/bin/env python
```

Access library functions:

```
from math import sin
import sys
```

Read command line argument and convert it to a floating point:

```
x = float(sys.argv[1])
```

Print out the result using a format string:

```
print "Hello world, sin(%g) = %g." % (x, sin(x))
```

Simple Assignments

```
a = 10 # a is a variable referencing an
       # integer object of value 10

b = True # b is a boolean variable

a = b # a is now a boolean as well
      # (referencing the same object as b)

b = increment(4) # b is the value returned by a function

is_equal = a == b # is_equal is True if a == b
```

Simple control structures

- Loops:

```
while condition:
    <block of statements>
```

Here, `condition` must be a boolean expression (or have a boolean interpretation), for example: `i < 10` or `!found`

```
for element in somelist:
    <block of statements>
```

Note that `element` is a copy of the list items, not a reference into the list!

- Conditionals:

```
if condition:
    <block of statements>
elif condition:
    <block of statements>
else:
    <block of statements>
```

Ranges and Loops

- `range(start, stop, increment)` constructs a list. Typically, it is used in for loops:

```
for i in range(10):
    print i
```

- `xrange(start, stop, increment)` is better for fat loops since it constructs an iterator:

```
for i in xrange(10000000):
    sum += sin(i*pi*x)
```

- Looping over lists can be done in several ways:

```
names = ["Ola", "Per", "Kari"]
surnames = ["Olsen", "Pettersen", "Bremnes"]
for name, surname in zip(names, surnames):
    print name, surname # join element by element

for i, name in enumerate(names):
    print i, name # join list index and item
```

Lists and Tuples

```
mylist  = ['a string', 2.5, 6, 'another string']
mytuple = ('a string', 2.5, 6, 'another string')
mylist[1]  = -10
mylist.append('a third string')
mytuple[1] = -10  # illegal: cannot change a tuple
```

A tuple is a constant list (immutable)

List functionality

`a = []`	initialize an empty list
`a = [1, 4.4, 'run.py']`	initialize a list
`a.append(elem)`	add `elem` object to the end
`a + [1,3]`	add two lists
`a[3]`	index a list element
`a[-1]`	get last list element
`a[1:3]`	slice: copy data to sublist (here: index 1, 2)
`del a[3]`	delete an element (index `3`)
`a.remove(4.4)`	remove an element (with value `4.4`)
`a.index('run.py')`	find index corresponding to an element's value
`'run.py' in a`	test if a value is contained in the list

More list functionality

`a.count(v)`	count how many elements that have the value `v`
`len(a)`	number of elements in list `a`
`min(a)`	the smallest element in `a`
`max(a)`	the largest element in `a`
`min(["001", 100])`	tricky!
`sum(a)`	add all elements in `a`
`a.sort()`	sort list `a` (changes `a`)
`as = sorted(a)`	sort list `a` (return new list)
`a.reverse()`	reverse list `a` (changes `a`)
`b[3][0][2]`	nested list indexing
`isinstance(a, list)`	is `True` if `a` is a list

Functions and arguments

- User-defined functions:

```
def split(string, char):
    position = string.find(char)
    if  position > 0:
        return string[:position+1], string[position+1:]
    else:
        return string, ""

# function call:
message = "Heisann"
print split(message, "i")
```

  prints out `('Hei', 'sann')`.

- Positional arguments must appear before keyword arguments:

```
def split(message, char="i"):
    [...]
```

How to find more Python information

- The book contains only fragments of the Python language (intended for real beginners!)
- These slides are even briefer
- Therefore you will need to look up more Python information
- Primary reference: The official Python documentation at `docs.python.org`
- Very useful: The Python Library Reference, especially the index
- Example: what can I find in the `math` module? Go to the Python Library Reference index, find "math", click on the link and you get to a description of the module
- Alternative: `pydoc math` in the terminal window (briefer)
- Note: for a newbie it is difficult to read manuals (intended for experts) – you will need a lot of training; just browse, don't read everything, try to dig out the key info

eval and exec

- Evaluating string expressions with `eval`:

```
>>> x = 20
>>> r = eval('x + 1.1')
>>> r
21.1
>>> type(r)
<type 'float'>
```

- Executing strings with Python code, using `exec`:

```
exec("""
def f(x):
    return %s
""" % sys.argv[1])
```

Exceptions

- Handling exceptions:

```
try:
    <statements>
except ExceptionType1:
    <provide a remedy for ExceptionType1 errors>
except ExceptionType2, ExceptionType3, ExceptionType4:
    <provide a remedy for three other types of errors>
except:
    <provide a remedy for any other errors>
...
```

- Raising exceptions:

```
if z < 0:
    raise ValueError\
          ('z=%s is negative - cannot do log(z)' % z)
a = math.log(z)
```

File reading and writing

- Reading a file:

```
infile = open(filename, 'r')
for line in infile:
    # process line

lines = infile.readlines()
for line in lines:
    # process line

for i in xrange(len(lines)):
    # process lines[i] and perhaps next line lines[i+1]

fstr = infile.read()
# process the while file as a string fstr

infile.close()
```

- Writing a file:

```
outfile = open(filename, 'w')   # new file or overwrite
outfile = open(filename, 'a')   # append to existing file
outfile.write("""Some string
....
""")
```

Dictionary functionality

`a = {}`	initialize an empty dictionary
`a = {'point':[2,7], 'value':3}`	initialize a dictionary
`a = dict(point=[2,7], value=3)`	initialize a dictionary
`a['hide'] = True`	add new key-value pair to a dictionary
`a['point']`	get value corresponding to key `point`
`'value' in a`	`True` if `value` is a key in the dictionary
`del a['point']`	delete a key-value pair from the dictionary
`a.keys()`	list of keys
`a.values()`	list of values
`len(a)`	number of key-value pairs in dictionary `a`
`for key in a:`	loop over keys in unknown order
`for key in sorted(a.keys()):`	loop over keys in alphabetic order
`isinstance(a, dict)`	is `True` if `a` is a dictionary

String operations

```
s = 'Berlin: 18.4 C at 4 pm'
s[8:17]          # extract substring
s.find(':')      # index where first ':' is found
s.split(':')     # split into substrings
s.split()        # split wrt whitespace
'Berlin' in s    # test if substring is in s
s.replace('18.4', '20')
s.lower()        # lower case letters only
s.upper()        # upper case letters only
s.split()[4].isdigit()
s.strip()        # remove leading/trailing blanks
', '.join(list_of_words)
```

Modules

Import module as namespace:

```
import sys
x = float(sys.argv[1])
```

Import module member `argv` into current namespace:

```
from sys import argv
x = float(argv[1])
```

Import everything from sys into current namespace (evil)

```
from sys import *
x = float(argv[1])
```

Import `argv` into current namespace under an alias

```
from sys import argv as a
x = float(a[1])
```