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

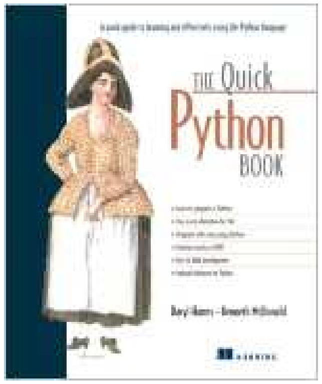
- Very little mathematics is needed to complete the course.
- Basic knowledge will make life easier;
  - General functions, such as $f(x) = ax + b$, and how they are turned into computer code
  - Standard mathematical functions such as $\sin(x), \cos(x)$ and exponential functions
  - Simple matrix-vector 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.

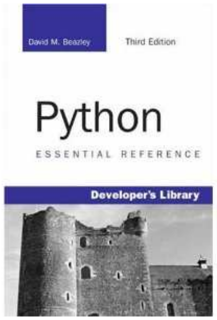
Teaching material (1)

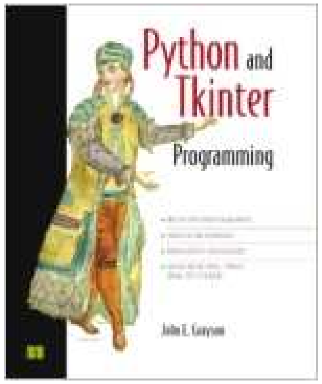
- Slides from lectures (by Sundnes, Skavhaug, Langtangen et al).
  A preliminary version is found here
  `http://www.uio.no/studier/emner/matnat/ifi/INF3331/h14/inf3331_h14.pdf`
  Do not print these slides now! Will be substantially updated through the fall.
- H.P. Langtangen and G. K. Sandve: *Illustrating 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)

Basic Bash programming

Overview of Unix shells

- The original scripting languages were (extensions of) command interpreters in operating systems
- Primary example: Unix shells
- Bourne shell (`sh`) was the first major shell
- C and TC shell (`csh` and `tcsh`) had improved command interpreters, but were less popular than Bourne shell for programming
- Bourne Again shell (Bash/`bash`): GNU/FSF improvement of Bourne shell
- Other Bash-like shells: Korn shell (`ksh`), Z shell (`zsh`)
- Bash is the dominating Unix shell today

Why learn Bash?

- Learning Bash means learning Unix
- Learning Bash means learning the roots of scripting (Bourne shell is a subset of Bash)
- Shell scripts, especially in Bourne shell and Bash, are frequently encountered on Unix systems
- Bash is widely available (open source) and the dominating command interpreter and scripting language on today's Unix systems

Why learn Bash? (2)

- Shell scripts evolve naturally from a workflow:
  1. A sequence of commands you use often are placed in a file
  2. Command-line options are introduced to enable different options to be passed to the commands
  3. Introducing variables, if tests, loops enables more complex program flow
  4. At some point pre- and postprocessing becomes too advanced for bash, at which point (parts of) the script should be ported to Python or other tools
- Shell scripts are often used to glue more advanced scripts in Perl and Python

Remark

- We use plain Bourne shell (`/bin/sh`) when special features of Bash (`/bin/bash`) are not needed
- Most of our examples can in fact be run under Bourne shell (and of course also Bash)
- Note that Bourne shell (`/bin/sh`) is usually just a link to Bash (`/bin/bash`) on Linux systems
  (Bourne shell is proprietary code, whereas Bash is open source)

More information

- `man bash`
- “Introduction to and overview of Unix” link in `doc.html` (part of the source code for *Python Scripting for Computational Science*, by H.P. Langtangen)
- Bash reference manual:
  `www.gnu.org/software/bash/manual/bashref.html`
- “Advanced Bash-Scripting Guide”:
  `http://www.tldp.org/LDP/abs/html/`

What Bash is good for

- File and directory management
- Systems management (build scripts)
- Combining other scripts and commands
- Rapid prototyping of more advanced scripts
- Simple output processing, plotting etc.

What Bash is not good for

- Cross-platform portability
- Graphics, GUIs
- Interface with libraries or legacy code
- More advanced post processing and plotting

Some common tasks in Bash

- file writing
- for-loops
- running an application
- pipes
- writing functions
- file globbing, testing file types
- copying and renaming files, creating and moving to directories, creating directory paths, removing files and directories
- directory tree traversal
- packing directory trees

Bash variables and commands

- Assign a variable by `x=3.4`, retrieve the value of the variable by `$x` (also called *variable substitution*).
- Variables passed as command line arguments when running a script are called `positional parameters`.
- Bash has a number of built in commands, type `help` or `help | less` to see all.
- The real power comes from all the available Unix commands, in addition to your own applications and scripts.

Bash variables (1)

- Variables in Bash are untyped!
- Generally treated as character arrays, but permit simple arithmetic and other operations
- Variables can be explicitly declared to integer or array;

```
declare -i i    # i is an integer
declare -a A    # A is an array
```

Bash variables (2)

```
x=3
y=2
z=$x+$y
echo $z            #output; 3+2

z=$((x+y))
((v=x+y))
let w=x+y
echo $z $v $w #output; 5 5 5

declare -i x=3
declare -i y=4
z=$x+$y
echo $z              #output; 3+2

y=a
echo $y              #output; 0
```

Bash variables (3)

- Comparison of two integers use a syntax different comparison of two strings:

```
if [ $i -eq 10 ]; then          # integer comparison
if [ "$name" == "10" ]; then    # string  comparison
```

- Unless you have declared a variable to be an integer, assume that all variables are strings and use double quotes (strings) when comparing variables in an if test

```
if [ "$?" != "0" ]; then  # this is safe
if [  $?  !=  0  ]; then  # might be unsafe
```

Example: the very basics

- Let’s start with a script writing "Hello, World!"
- Scientific computing extension: compute the sine of a number as well
- The script (hw.sh) should be run like this:

```
./hw.sh 3.4
```

  or (less common):

```
bash hw.sh 3.4
```

- Output:

```
Hello, World! sin(3.4)=-0.255541102027
```

- Can be done with a single line of code:

```
echo "Hello, World! sin($1)=$(echo "s($1)" | bc -l)"
```

Purpose of this script

Demonstrate

- how to read a command-line argument
- how to call a math (sine) function
- How to combine different commands (piping)
- how to work with variables
- how to print text and numbers

The code, in expanded version

File hw.sh:

```
#!/bin/sh
r=$1  # store first command-line argument in r
s=`echo "s($r)" | bc -l`

# print to the screen:
echo "Hello, World! sin($r)=$s"
```

Comments

- The first line specifies the interpreter of the script (here `/bin/sh`, could also have used `/bin/bash`, or skipped this line altogether...)
- The command-line variables are available as the script variables

  ```
  $1  $2  $3  $4 and so on
  ```

- Variables are initialized as

  ```
  r=$1
  ```

  while the *value* of `r` requires a dollar prefix:

  ```
  my_new_variable=$r  # copy r to my_new_variable
  ```

Bash and math

- Bourne shell and Bash have very little built-in math, we therefore need to use bc, Perl (or some other tool) to do the math:

```
s=`echo "s($r)" | bc -l`
s = $(echo 's($r)' | bc -l')
s=`perl -e '$s=sin($ARGV[0]); print $s;' $r`
```

- Back quotes means executing the command inside the quotes and assigning the output to the variable on the left-hand-side

```
some_variable=`some Unix command`

# alternative notation:
some_variable=$(some Unix command)
```

The bc program

- bc = interactive calculator
- Documentation: man bc
- bc -l means bc with math library
- Note: sin is s, cos is c, exp is e
- echo sends a text to be interpreted by bc and bc responds with output (which we assign to `s`)

```
variable=`echo "math expression" | bc -l`
```

- The Bash construct `(( ))` or builtin command `let` *`expression`* does somthing similar, but only for very simple math expressions

Printing

- The `echo` command is used for writing:

```
echo "Hello, World! sin($r)=$s"
```

  and variables can be inserted in the text string (variable interpolation)

- Bash also has a printf function for format control:

```
printf "Hello, World! sin(%g)=%12.5e\n" $r $s
```

- `cat` is usually used for printing multi-line text (see next slide)

Convenient debugging tool: -x

- Each source code line is printed prior to its execution if you add -x as option to `/bin/sh` or `/bin/bash`
- Either in the header

  ```
  #!/bin/sh -x
  ```

  or on the command line:

  ```
  unix> /bin/sh -x hw.sh
  unix> sh -x hw.sh
  unix> bash -x hw.sh
  ```

- Very convenient during debugging

Example: the classical Unix script

A combination of commands, or a single long command, that you use often;

```
../build/app/pulse_app --cmt WinslowRice --casename ellipsoid
    < ellipsoid.i | tee main_output
```

(should be a single line)
In this case, flexibility is often not a high priority. However, there is room for improvement;

- Not possible to change command line options, input and output files
- Output file `main_output` is overwritten for each run
- Can we edit the input file for each run?

Problem 1; changing application input

In many cases only one parameter is changed frequently;

```
CASE='testbox'
CMT='WinslowRice'
if [ $# -gt 0 ]; then
   CMT=$1
fi
INFILE='ellipsoid_test.i'
OUTFILE='main_output'

../build/app/pulse_app --cmt $CMT --casename $CASE
  < $INFILE | tee $OUTFILE
```

Still not very flexible, but in many cases sufficient. More flexibility requires more advanced parsing of command line options, which will be introduced later.

Problem 2; overwriting output file

- A simple solution is to add the output file as a command line option, but what if we forget to change this from one run to the next?
- Simple solution to ensure data is never over-written:

```
jobdir=$PWD/`date +%s`
mkdir $jobdir
cd $jobdir

../../build/app/pulse_app --cmt $CMT --casename $CASE  < $INFILE
cd ..
if [ -L 'latest' ]; then
rm latest
fi
ln -s $jobdir latest
```

Problem 2; overwriting output file (2)

Alternative solutions;

- Use process ID of the script (`$$`, not really unique)
- `mktemp` can create a temporary file with a unique name, for use by the script
- Check if subdirectory exists, exit script if it does

```
dir=$case
# check if $dir is a directory:
if [ -d $dir ]
  #exit script to avoid overwriting data
  then
    echo "Output directory exists, provide a different name"
    exit
fi
mkdir $dir    # create new directory $dir
cd $dir       # move to $dir
```

Alternative `if`-tests

As with everything else in Bash, there are multiple ways to do `if`-tests:

```
# the 'then' statement can also appear on the 1st line:
if [ -d $dir ]; then
  exit
fi

# another form of if-tests:
if test -d $dir; then
  exit
fi

# and a shortcut:
[ -d $dir ] && exit
test -d $dir && exit
```

Problem 3; can we edit the input file at run time?

- Some applications do not take command line options, all input must read from standard input or an input file
- A Bash script can be used to equip such programs with basic handling of command line options
- We want to grab input from the command line, create the correct input file, and run the application

File reading and writing

- File writing is efficiently done by 'here documents':

```
cat > myfile <<EOF
multi-line text
can now be inserted here,
and variable substition such as
$myvariable is
supported. The final EOF must
start in column 1 of the
script file.
EOF
```

Setting default parameters

```
#!/bin/sh

pi=3.14159
m=1.0; b=0.7; c=5.0; func="y"; A=5.0;
w=`echo 2*$pi | bc`
y0=0.2; tstop=30.0; dt=0.05; case="tmp1"
screenplot=1
```

Parsing command-line options

```
# read variables from the command line, one by one:
while [ $# -gt 0 ]  # $# = no of command-line args.
do
    option = $1; # load command-line arg into option
    shift;       # eat currently first command-line arg
    case "$option" in
        -m)
            m=$1; shift; ;;  # load next command-line arg
        -b)
            b=$1; shift; ;;
        ...
        *)
            echo "$0: invalid option \"$option\""; exit ;;
    esac
done
```

Alternative to case: if

`case` is standard when parsing command-line arguments in Bash, but if-tests can also be used. Consider

```
case "$option" in
    -m)
        m=$1; shift; ;;  # load next command-line arg
    -b)
        b=$1; shift; ;;
    *)
        echo "$0: invalid option \"$option\""; exit ;;
esac
```

versus

```
if [ "$option" == "-m" ]; then
    m=$1; shift;  # load next command-line arg
elif [ "$option" == "-b" ]; then
    b=$1; shift;
else
    echo "$0: invalid option \"$option\""; exit
fi
```

After assigning variables, we can write the input file

```
# write to $infile the lines that appear between
# the EOF symbols:

cat > $infile <<EOF
        m=$m
        b=$b
        c=$c
        A=$A
        w=$w
        y0=$y0
        tstop=$tstop
        dt=$dt
EOF
```

Then execute the program as usual

- Redirecting input to read from the new input file

```
../../build/pulse_app < $infile
```

- Can add a check for successful execution:

```
# the shell variable $? is 0 if last command
# was successful, otherwise $? != 0

if [ "$?" != "0" ]; then
  echo "running pulse_app failed"; exit 1
fi

# exit n sets $? to n
```

Making plots with Gnuplot (old-style)

- Make Gnuplot script:

```
echo "set title '$case: m=$m ...'" > $case.gnuplot
...
# continue writing with a here document:
cat >> $case.gnuplot <<EOF
set size ratio 0.3 1.5, 1.0;
...
plot 'sim.dat' title 'y(t)' with lines;
...
EOF
```

- Run Gnuplot:

```
gnuplot -geometry 800x200 -persist $case.gnuplot
if [ "$?" != "0" ]; then
  echo "running gnuplot failed"; exit 1
fi
```

- Python is preferred over Bash for most kinds of plotting

Other uses of `cat`

```
cat myfile              # write myfile to the screen
cat myfile >  yourfile # write myfile to yourfile
cat myfile >> yourfile # append myfile to yourfile
cat myfile | wc        # send myfile as input to wc
```

For-loops

- What if we want to run the application for multiple input files?

  ```
  ./run.sh test1.i test2.i test3.i test4.i
  ```

  or

  ```
  ./run.sh *.i
  ```

- A for-loop over command line arguments

  ```
  for arg in $@; do
  ../../build/app/pulse_app < $arg
  done
  ```

- Can be combined with more advanced command line options, output directories, etc...

For-loops (2)

- For loops for file management:

```
files=`/bin/ls *.tmp`
# we use /bin/ls in case ls is aliased

for file in $files
do
  echo removing $file
  rm -f $file
done
```

Counters

- Declare an integer counter:

```
declare -i counter
counter=0
# arithmetic expressions must appear inside (( ))
((counter++))
echo $counter  # yields 1
```

- For-loop with counter:

```
declare -i n; n=1
for arg in $@; do
  echo "command-line argument no. $n is <$arg>"
  ((n++))
done
```

C-style for-loops

```
declare -i i
for ((i=0; i<$n; i++)); do
  echo $c
done
```

Example: bundle files

- Pack a series of files into one file
- Executing this single file as a Bash script packs out all the individual files again
- Usage:

```
bundle file1 file2 file3 > onefile  # pack
bash onefile # unpack
```

- Writing `bundle` is easy:

```
#/bin/sh
for i in $@; do
    echo "echo unpacking file $i"
    echo "cat > $i <<EOF"
    cat $i
    echo "EOF"
done
```

The bundle output file

- Consider 2 fake files; file1

  ```
  Hello, World!
  No sine computations today
  ```

  and file2

  ```
  1.0 2.0 4.0
  0.1 0.2 0.4
  ```

- Running `bundle file1 file2` yields the output

  ```
  echo unpacking file file1
  cat > file1 <<EOF
  Hello, World!
  No sine computations today
  EOF
  echo unpacking file file2
  cat > file2 <<EOF
  1.0 2.0 4.0
  0.1 0.2 0.4
  EOF
  ```

Running an application

- Running in the foreground:

```
cmd="myprog -c file.1 -p -f -q";
$cmd < my_input_file

# output is directed to the file res
$cmd < my_input_file > res

# process res file by Sed, Awk, Perl or Python
```

- Running in the background:

```
myprog -c file.1 -p -f -q < my_input_file &
```

or stop a foreground job with Ctrl-Z and then type `bg`

Pipes

- Output from one command can be sent as input to another command via a pipe

```
# send files with size to sort -rn
# (reverse numerical sort) to get a list
# of files sorted after their sizes:

/bin/ls -s | sort -r


cat $case.i | oscillator
# is the same as
oscillator < $case.i
```

- Make a new application: sort all files in a directory tree `root`, with the largest files appearing first, and equip the output with paging functionality:

```
du -a root | sort -rn | less
```

Functions

```
# compute x^5*exp(-x) if x>0, else 0 :

function calc() {
   echo "
   if ( $1 >= 0.0 ) {
      ($1)^5*e(-($1))
   } else {
      0.0
   } " | bc -l
}

# function arguments: $1 $2 $3 and so on
# return value: last statement

# call:
r=4.2
s=`calc $r`
```

Another function example

```
#!/bin/bash

function statistics {
  avg=0; n=0
  for i in $@; do
    avg=`echo $avg + $i | bc -l`
    n=`echo $n + 1 | bc -l`
  done
  avg=`echo $avg/$n | bc -l`

  max=$1; min=$1; shift;
  for i in $@; do
    if [ `echo "$i < $min" | bc -l` != 0 ]; then
      min=$i; fi
    if [ `echo "$i > $max" | bc -l` != 0 ]; then
      max=$i; fi
  done
  printf "%.3f %g %g\n" $avg $min $max
}
```

Calling the function

```
statistics 1.2 6 -998.1 1 0.1

# statistics returns a list of numbers
res=`statistics 1.2 6 -998.1 1 0.1`

for r in $res; do echo "result=$r"; done

echo "average, min and max = $res"
```

File globbing, for loop on the command line

- List all .ps and .gif files using wildcard notation:

```
files=`ls *.ps *.gif`

# or safer, if you have aliased ls:
files=`/bin/ls *.ps *.gif`

# compress and move the files:
gzip $files
for file in $files; do
  mv ${file}.gz $HOME/images
```

Testing file types

```
if [ -f $myfile ]; then
    echo "$myfile is a plain file"
fi

# or equivalently:
if test -f $myfile; then
    echo "$myfile is a plain file"
fi

if [ ! -d $myfile ]; then
    echo "$myfile is NOT a directory"
fi

if [ -x $myfile ]; then
    echo "$myfile is executable"
fi

[ -z $myfile ] && echo "empty file $myfile"
```

Rename, copy and remove files

```
# rename $myfile to tmp.1:
mv $myfile tmp.1

# force renaming:
mv -f $myfile tmp.1

# move a directory tree my tree to $root:
mv mytree $root

# copy myfile to $tmpfile:
cp myfile $tmpfile

# copy a directory tree mytree recursively to $root:
cp -r mytree $root

# remove myfile and all files with suffix .ps:
rm myfile *.ps

# remove a non-empty directory tmp/mydir:
rm -r tmp/mydir
```

Directory management

```
# make directory:
$dir = "mynewdir";
mkdir $mynewdir
mkdir -m 0755 $dir  # readable for all
mkdir -m 0700 $dir  # readable for owner only
mkdir -m 0777 $dir  # all rights for all

# move to $dir
cd $dir
# move to $HOME
cd

# create intermediate directories (the whole path):
mkdirhier $HOME/bash/prosjects/test1
# or with GNU mkdir:
mkdir -p  $HOME/bash/prosjects/test1
```

The find command

Very useful command!

- `find` visits all files in a directory tree and can execute one or more commands for every file
- Basic example: find the `oscillator` codes

```
find $scripting/src -name 'oscillator*' -print
```

- Or find all PostScript files

```
find $HOME \( -name '*.ps' -o -name '*.eps' \) -print
```

- We can also run a command for each file:

```
find rootdir -name filenamespec -exec command {} \; -print
# {} is the current filename
```

Applications of find (1)

- Find all files larger than 2000 blocks a 512 bytes (=1Mb):

```
find $HOME -name '*' -type f -size +2000 -exec ls -s {} \;
```

- Remove all these files:

```
find $HOME -name '*' -type f -size +2000 \
     -exec ls -s {} \; -exec rm -f {} \;
```

  or ask the user for permission to remove:

```
find $HOME -name '*' -type f -size +2000 \
     -exec ls -s {} \; -ok rm -f {} \;
```

Applications of find (2)

- Find all files not being accessed for the last 90 days:

  ```
  find $HOME -name '*' -atime +90 -print
  ```

  and move these to /tmp/trash:

  ```
  find $HOME -name '*' -atime +90 -print \
       -exec mv -f {} /tmp/trash \;
  ```

Tar and gzip

- The `tar` command can pack single files or all files in a directory tree into one file, which can be unpacked later

```
tar -cvf myfiles.tar mytree file1 file2

# options:
# c: pack, v: list name of files, f: pack into file

# unpack the mytree tree and the files file1 and file2:
tar -xvf myfiles.tar

# options:
# x: extract (unpack)
```

- The tarfile can be compressed:

```
gzip mytar.tar

# result: mytar.tar.gz
```

Two find/tar/gzip examples

- Pack all PostScript figures:

```
tar -cvf ps.tar `find $HOME -name '*.ps' -print`
gzip ps.tar
```

- Pack a directory but remove CVS directories and redundant files

```
# take a copy of the original directory:
cp -r myhacks /tmp/oblig1-hpl
# remove CVS directories
find /tmp/oblig1-hpl -name CVS -print -exec rm -rf {} \;
# remove redundant files:
find /tmp/oblig1-hpl \( -name '*~' -o -name '*.bak' \
 -o -name '*.log' \) -print -exec rm -f {} \;
# pack files:
tar -cf oblig1-hpl.tar /tmp/tar/oblig1-hpl.tar
gzip oblig1-hpl.tar
# send oblig1-hpl.tar.gz as mail attachment
```