Programmer's productivity tools

2012 May 23
The speaker

Davide Del Vento, PhD in Physics
Consulting Services, Software Engineer
NCAR - CISL
http://www2.cisl.ucar.edu/uss/csg
office: Mesa Lab, Room 55G
phone: (303) 497-1233
e-mail: ddvento@ucar.edu

ExtraView Tickets: cislhelp@ucar.edu
Getting ready

- Please ssh to mirage...
- Add `/fs/local/apps/Python-2.6.5/bin/` to PATH
  Or
- `source /contrib/modules-3.2.8/Modules/3.2.8/init/bash`
  module load python
Table of contents

- Refresh of UNIX tools and UNIX philosophy
- grep/awk/sed
- Makefiles (and make)
- Unit Testing
- python
Unix philosophy

- Unix philosophy is to have **small tools**...
- ...that do a **small job**...
- ...hopefully **very well**...
- ...and only that small job, **nothing else**...
- ...but are able to **interact with other** tools...
Unix philosophy (1)
Unix philosophy (1)
Unix philosophy (2)

```
exiftool `find . -iname *.MOV | grep toddlers` |
grep `^Duration` | sort | head -3
```

- which ones are the 3 shortest movies of my toddlers -
  shortest in time, not in file size
Backquotes ` `something` `.

- The something command is executed replaced with its standard output

```
exiftool `find . -iname *.MOV | grep toddlers` | grep '^Duration' | sort | head -3
```
Another example

- Replace \((xxx,yyy)\) with \([xxx][yyy]\)
Another example

- Replace (xxx,yyy) with [xxx][yyy]
- s/\((.\{-\})\),\((.\{-\})\)/[\1][\2]/g
Some sed syntax

\[ s/(\( . \{-\} \)), (\( . \{-\} \)))/[ \1 ][ \2 ]/g \]

- \ . \ any character
Some sed syntax

\( s/\((\\(.\\{\-}\)\)),\((\\(.\\{\-}\)))/[\1][\2]/g \)

- .  any character
- .\{\-\} repeated as little times as possible
Some sed syntax

\[ \text{s/} \((.\{\-\}\))\,(.\{\-\}\))\,/[\1][\2]/g \]

- . any character
- .\{\-\} repeated as little times as possible
- \((xxx\)) is given a numeric “name” for later use
Some sed syntax

\[ s/(\((.\{|-\}\)),\((.\{|-\}\)))/[\1][\2]/g \]

- . any character
- .\{|-\} repeated as little times as possible
- \((xxx)\) is given a numeric “name” for later use
- \((xx,yy)\) these just match themselves
Some sed syntax

```bash
s/(\(.\{-}\))/\[\1\]\[\2\]/g
```

- . any character
- \(.\{-}\) repeated as little times as possible
- \((xxx\)) is given a numeric “name” for later use
- \((xx,yy)\) these just match themselves
- \[\1\][\2\] and everything is replaced by this
Some sed syntax

`s/\((.\{-}\),(.\{-}\))/[^1][^2]/g`

- `. ` any character
- `.\{-}` repeated as little times as possible
- `\(xxx\)` is given a numeric “name” for later use
- `(xx,yy)` these just match themselves

- `[^1][^2]` and everything is replaced by this

- Replace `(xxx,yyy)` with `[xxx][yyy]`
Some grep syntax

grep "^Duration *[0-9]+"

- ^$ anchors
- [aaa] list
- . any char
- * repeat the previous as many times as possible (zero included)
- + at least once
- (a|b) a OR b (a and b can be everything)
Some awk syntax

- awk 'program' file.txt
- Simple program:
  
  ```
  {print $2 $6 $7}
  ```
- Intermediate program:
  
  ```
  BEGIN { count=0; }
  $4 ~ /something/ { count++; }
  END { print count; }
  ```
Regular Expressions

- I won't cover them more
- Their syntax is somehow dependent on the tool (e.g. grep not 100% the same as sed)
- Useful (and easier) also in python
- You should consider studying and using them
- Please give feedback
Rehearsing

- Shell scripts
- Shell variables
- Environmental variables
Makefiles

- Sort of shell scripts
- Usually with the goal of compiling programs
- Keeping track of lots of things, like what:
  - needs to be done to accomplish “something”
  - depends on what else
  - has been done and don't need to be redone
Makefiles

- For example, if the source code changes, it must be re-compiled
- But you don't have to recompile everything, just the piece that changed
- But you have to relink the piece that has changed with the ones that haven't
- Make can do that (and more)
Makefile example (1)

PAR=something

compile: my_prog.c
        icc $(PAR) my_prog.c -o my_prog.exe

clean:
        rm -f my_prog.exe

- ^^^^^^^ these are not spaces, but a TAB
- compile is the target (in this case phony)
- my_prog.c is the prerequisite
- icc $(PAR) blah blah is the action
- You would invoke this on the command line as make compile
Makefile example (2)

FC=ifort
LD=ld
FFLAGS=-c

compile: prog1 prog2
    @echo Compiled $?

%.o: %.f90
    $(FC) $(FFLAGS) -o $@ ??

%: %.o
    $(LD) ?? -o $@

Davide Del Vento
Makefile example (3)
Configure scripts

- Automated script generating makefiles:
  - Chooses which compiler(s) to use
  - Checks what libraries are available
  - Where to install the program
  - Other options the user have to compile

- For large programs, usually the configure script is generated by autoconf
Alternatives to Make

- cmake
- qmake
- scons
- ant
- maven
- etc
Testing
Ideal Testing job
Ideal Testing job
Unit testing

- Software testing is often an overlooked, underestimate field

- Overlooked because it is:
  - Not done
  - Not done early enough
  - Not done well enough

- Underestimated because it is not realized that:
  - it is hard to plan
  - it is easy to perform
  - it saves a lot of time/effort/money
Unit test goals

- Verify that the program is doing what is supposed to do:
  - somehow this is done anyway
  - but usually not consistently
  - with throw-away, untracked code (e.g. print)

- Guide the program design:
  - explicitly say what the program is supposed to do (like docs or comments)
  - but executable, and thus always up-to-date

- APIs/libraries/language exploration
How is this possible?
python

- The programming language, I mean...
Python quick facts

- Free
- Machine independent
- Well documented
- Easy to learn
- Easy to use (includes easy to read)
- With lot of libraries
- Not super-fast (at least by default)
Python interactive shell
Plotting in python
```python
from pylab import *

from numpy.random import *

hist(randn(1000), 100)
savefig('foo.png')
```
Python Libraries for Scientific Programmer

- PyAOS
- OpenOpt
- IPython Interactive Computing
- pandas
- pypy
- fenics project
- matplotlib
- PyNGL and PyNIO Python packages
- mpg4py
- NumPy
- SciKits
- NCAR Computational & Information Systems Laboratory
- petsc4py PETSc for Python - Python bindings for PETSc libraries.
- graph-tool
- sympy Python library for symbolic mathematics
- PyACTS Project
- Scientific Python

Davide Del Vento
Python scripts
Fibonacci numbers
Classes

- Basic elements of object-oriented programming
- A fully-fledged oop course would take weeks
- Suffice to say that “classes” are “something in memory” like C's structs and Fortran's derived TYPES
- They contain *data* (unsurprisingly) *fields*, in oop jargon
- But most importantly functions/subroutines *methods*, in oop jargon
- **Instances** of a class are occurrences of it
class MyMath:
    def fibonacci(self, n):
        a,b = 0,1
        for i in range(0,n):
            print "debug:", a
            a,b, = b,a+b
        return a

- MyMath class
- fibonacci method
- self reference to container instance
Some Python neatness

- Variable management
- Strong typing
- Dynamic typing
- Duck typing
- Name-based polymorphism
- Automatic overloading
Automatic overloading

- Python

```python
class Math(object):
    def sum(self, a, b):
        return a + b
```

- Fortran

```fortran
module overloaded_sum
private
public :: binary_sum
 interface binary_sum
    module procedure int_sum
    module procedure real_sum
end interface binary_sum
contains
function int_sum( a, b) result( s)
    integer :: s
    integer, intent( in) :: a, b
    continue
    s = a + b
    return
end function int_sum
function real_sum( a, b) result( s)
    real :: s
    real, intent( in) :: a, b
    continue
    s = a + b
    return
end function real_sum
end module overloaded_sum
```
Some Python neatness

- Variable management
- Strong typing
- Dynamic typing
- **Duck typing**
- Name-based polymorphism
- Automatic overloading
More Python neatness

- Text processing (incl. regular expressions)
- I/O without boilerplate
- Expressive iterators (on string, lists, arrays, etc)
- Convenient command line argument parsing with `argparse` module - including automatic(!) usage documentation and validity (e.g. range) checking
Poor's man grep example

- Python

```python
import sys
for line in open(sys.argv[2]):
    if sys.argv[1] in line:
        print line
```

- Fortran

```fortran
program pmg
    character(len= 80) :: line
    character(len= 20) :: word
    integer :: loc
    integer :: ios
    continue
    call get_command_argumeny(value=word)
    do
        read(unit=*, fmt='(a)', iostat= ios) line
        if(ios < 0 ) exit
        loc = index( line, word)
        if(loc > 0 )then
            write( unit= *, fmt= *) line
        end if
    end do
stop
end program pmg
```
List comprehensions

- A concise, expressive way to create lists
- Each element is result of an operation to each element of a “parent” list
- Sub-list of only the elements that satisfy a given condition
- Or both
Some python native data structures

- Numbers (boolean, plain and long integer, float, complex)
- Strings (with lots of useful methods)
- Lists (just seen, kind of arrays)
- Stacks and Queues (just instances of lists)
- Tuples (like lists but immutable)
- Sets (like lists but order is irrelevant and duplicates are not allowed)
- Dictionaries (think list with arbitrary index, e.g. strings: my_dict["ciao"]="hello")

- Functional programming
Unit Testing in python

- Various frameworks to make your life easier
- I'll use the unittest module, shipped with python itself
Unit Testing in python

- Various frameworks to make your life easier
- I'll use the unittest module, shipped with python itself
Tester's dilemma

- I should write a lot of tests cases
- But how many?
It's not a dilemma

- Two possible answers

1) White box testing: tests for edge cases, plus a couple of internal cases

2) Black box testing and Test Driven Development: try to use the tests to define what the code is supposed to do
Overlapping rectangles

- From softwarecarpentry.org
Overlapping rectangles

- From softwarecarpentry.org
Overlapping rectangles

- From softwarecarpentry.org
Bottom line

- Test what a piece of code is **supposed** to do
- Not **what** it does
- Not **how** it does it
**Bottom line**

- Test what a piece of code is **supposed** to do
- Not **what** it does
- Not **how** it does it
Hands on example

- Given a 1-dimensional list of floats
- e.g. $[-34, 1, 0, -2, 123, -83]$ 
- Find the biggest sub-list with the largest sum 
- e.g. $[1, 0, -2, 123]$ 
- The sub-list must be contiguous elements 
- i.e. the $-2$ cannot be skipped 
- but $-34$ and $-83$ can 

- A little artificial problem, but useful to prove Unit Testing in practice
Hands on example (2)
Divide and Conquer
Divide and Conquer
D&C maximum sublist

- The maximum sublist must be:
  - Entirely in the “left” sublist
  - Entirely in the “right” sublist
  - Part in the “left” sublist, part in the “right” one
D&C maximum sublist

- The maximum sublist must be:
  - Entirely in the “left” sublist
  - Entirely in the “right” sublist
  - Part in the “left” sublist, part in the “right” one

- Idea: solve the 3 cases and return the one with the maximum sum
More on tests

- Test patterns
- Stateless vs Stateful programming
- Refactoring
- Performance tuning
- Try to have fun and gain experience
Unit testing

- Software testing is often an overlooked, underestimate field

- Overlooked because it is:
  - Not done
  - Not done early enough
  - Not done well enough

- Underestimated because it is not realized that:
  - it is hard to plan
  - it is easy to perform
  - it saves a lot of time/effort/money
References (1)

- grep, awk and sed/VI tutorials
  http://www.panix.com/~elflord/unix/grep.html
  http://www.grymoire.com/Unix/Awk.html
  http://www.eng.hawaii.edu/Tutor/vi.html

- Regular expressions
  Tony Stubblebine *Regular Expression Pocket Reference*
  http://www.regular-expressions.info/quickstart.html

- Make
  http://www.gnu.org/software/make/manual/
  http://www.eng.hawaii.edu/Tutor/Make/
References (2)

- Python's books & tutorials
  http://docs.python.org/tutorial/index.html
  http://www.diveintopython.net/
  http://learnpythonthehardway.org/

- Python's notebooks (sage and ipython):
  http://mail.scipy.org/pipermail/ipython-user/2012-January/009021.html

- Unit Testing
  Kent Beck - JUnit Pocket Guide
  Gerard Meszaros - xUnit Test Patterns: Refactoring Test Code
  http://pytddmon.org/
References (3)

- Python for scientific computing
  
  http://scipy.org/
  http://scipy.org/Topical_Software
  http://ipython.org/
  http://sagemath.org/
  http://sagenb.org/
  http://pandas.pydata.org/

- HPC and parallel programming in Python
  
  http://www.tacc.utexas.edu/user-services/training/course-materials
  http://archive.org/details/Scipy2010-B.e.Granger-ParallelComputingWithIpythonAnApplicationTo

- Other libraries mentioned in slide 41
Attribution-Noncommercial-Share Alike 3.0 United States

You are free:

- **to Share** — to copy, distribute, display, and perform the work
- **to Remix** — to make derivative works

Under the following conditions:

- **Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Noncommercial.** You may not use this work for commercial purposes.
- **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author’s moral rights.

Your fair use and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license).