How to write a well-behaved Python command line application

PyCon AU 2012
Tutorial
Graeme Cross

This is an introduction to…

- Writing robust, maintainable command line applications
- Easily processing command line options
- Filters and files for input/output
- Handling errors and signals
- Testing your app
- Documenting and packaging your app

I am assuming…

- You know how to program (at least a bit!)
- Some familiarity with Python basics
- Python 2.6 or 2.7 (but not Python 3)
- Know the difference between:
  - GUI
  - Command prompt (`C:\>` or `~$`)

The examples and these notes

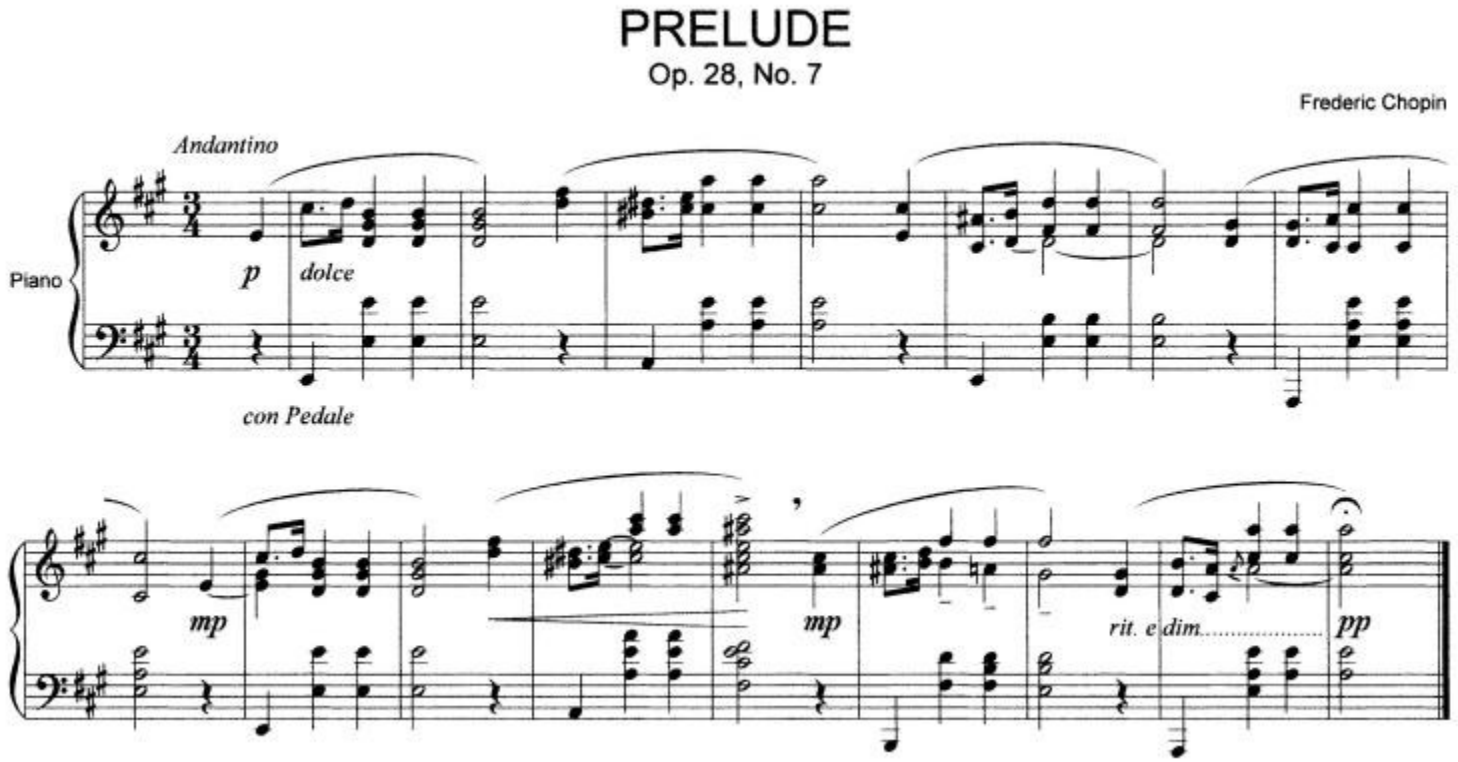
- We will use a number of demo scripts
  - This is an interactive tutorial
  - If you have a laptop, join in!
  - We will use ipython for the demos
- Code is on USB sticks being passed around
- Code & these notes are also at:

  http://www.curiousvenn.com/

ipython

- A very flexible Python shell
- Works on all platforms
- Lots of really nice features:
  - Command line editing with history
  - Coloured syntax
  - Edit and run within the shell
  - Web notebooks, Visual Studio, Qt, …
  - `%lsmagic`
- http://ipython.org/
  http://ipython.org/presentation.html

Prelude


PRELUDE
Op. 28, No. 7
Frederic Chopin
Andantino
Piano
p
dolce
con Pedale
mp
mp
rit. e dim.
pp

The command line

- One liners → shell scripts → applications
- Lots of interfaces:
  - Files
  - Pipes
  - User input/output
  - Processes
  - Networking
- The Unix model: **lots of small tools that can be combined in lots of useful ways**

What is a “well-behaved” app?

- Does one thing well
- Flexible
  - eg. handles input from files or pipes
- Robustly handles bad input data
- Gracefully handles errors
- Well-documented for new users

Why Python for the command line?

- Available on a **wide range of platforms**
- **Readable**, consistent syntax
  - Easy to write & easy to maintain
- **Scales well** for large apps & libraries
- **Lots of modules** = excellent support for:
  - Operating system functions (eg POSIX)
  - Networking
  - File systems

Why not Python?

- Simple one-liners often easier in bash
- eg. Neatly list all users in an LDAP group:

```
smbldap-groupshow $1 | tail -1 | tr [:lower:] [:upper:] | sed s/\,/\ /g | sed s/MEMBERUID:\ //
```

- Some operating systems are rumoured to not ship with Python
- Any other reasons??? Ummmm.....

Be platform aware

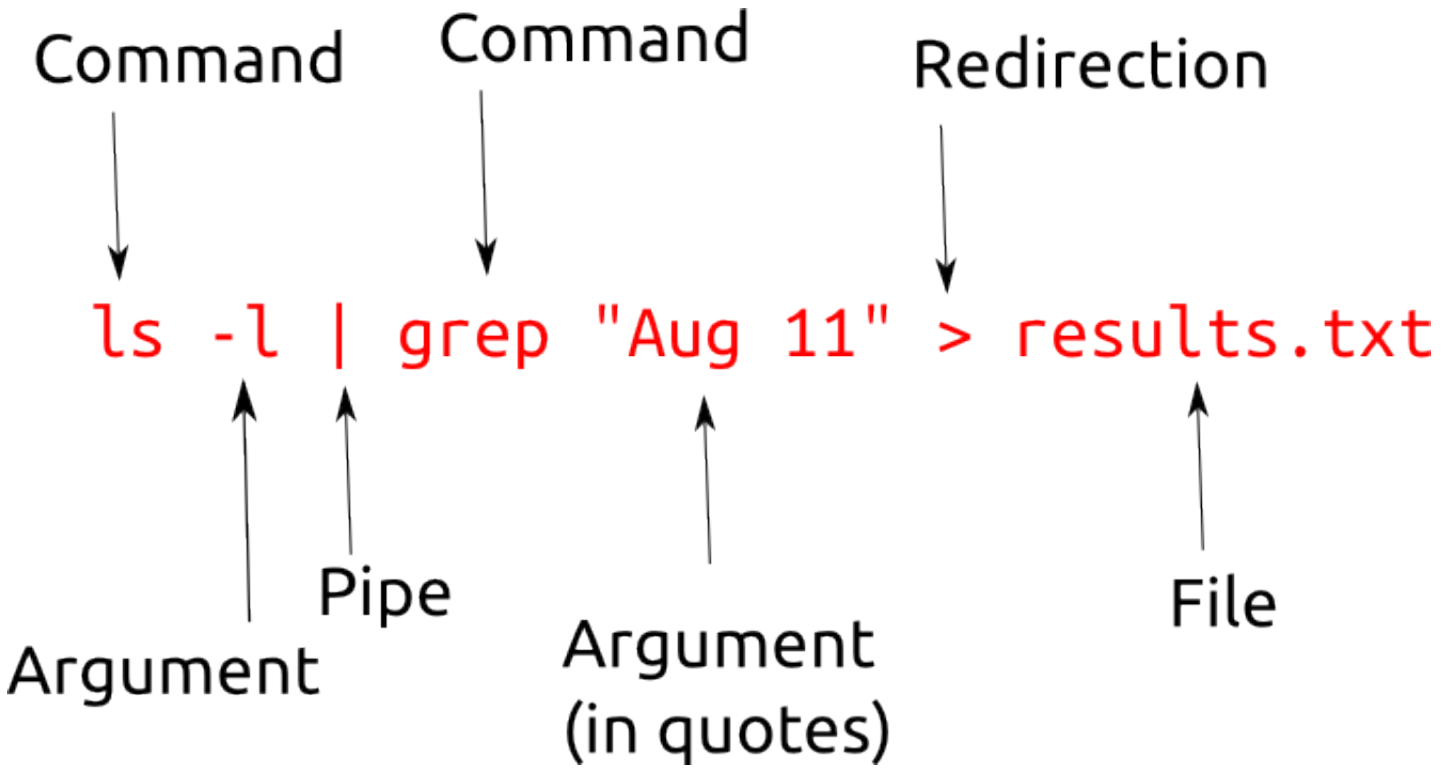
- Lots of standard library support
- No excuse to not support other platforms!
- Recommended modules for portability:
  - os
  - os.path
  - shutil
  - fileinput
  - tempfile
- Lots of other modules in PyPI

Here we go!

if __name__ == '__main__'

- For any Python script, break it up into:
  - Functions
  - A “main” function, called from command line
- Makes it easy to:
  - Test functionality
  - Reuse functions
- **Example: main1.py**

Anatomy of the command line



Files

- Reading, writing & appending to files
- Text or binary formats
- This is a tutorial on its own!
- **Example: file1.py**
- **Example: file2.py**

Pipes

- Instead of a filename, pipe input/output
- Create chains of tools
- Standard pipes:
  - Input: stdin
  - Output: stdout & stderr
- The **sys** module has support for these
- The **fileinput** module supports reading from stdin and files
- **Example: stdout.py**

Argument parsing

- Allow the user to specify arguments
  - Edit the script?
  - Modify a configuration file?
  - Specify arguments on the command line
- Need to handle:
  - Flags: `-h` or `--help`
  - Strings: “Run Forrest, Run”
  - Pipes
  - Invalid number of commands
  - Ideally: type checking, range checking, etc.

Argument parsing options

- Standard library: 3 different modules!
- Recommended module: **argparse**
- A series of examples:
  - **uniq1.py → uniq4.py**
  - **uniqsort.py**
- Lots more in PyPI!
- Recommended modules from PyPI:
  - clint
  - docopt

Argument parsing thoughts

- Always provide help at the command line
- Be consistent
  - Short and/or long flags?
  - Intuitive?
  - Ensure dangerous flags are obvious
  - Sensible mnemonics for abbreviated flags

Configuration files

- Useful for arguments that:
  - Could change
  - Don't change very often
  - Are probably machine- or user-specific
- Number of standard library modules:
  - **ConfigParser** (INI file format)
  - **json** (human & machine readable)
  - **xml.*** (if you must)
  - As well as **csv**, **plistlib** (for Mac .plist)
- Don't roll your own config file format!

Calling commands

- Python can execute other applications
- The **subprocess** module
  - The best of the standard library modules
  - Spawn a process
  - Read/write the input/output/error pipes
  - Get return code for error checking
  - Does not scale well
  - **Examples: subprocess1.py & subprocess2.py**

Calling commands, the easy way

- The **envoy** module (from PyPI)
  - A whole lot easier
  - More Pythonic
  - Recommended alternative to the subprocess module
  - https://github.com/kennethreitz/envoy/
  - **Example: envoy1.py**

Error handling

- Robust apps gracefully handle errors
  - Catch (all reasonable) errors
  - Report errors to the user
- Silently failing is rarely acceptable
- Blowing up is not much better!

Error handling: catching errors

- Exceptions
  - Recommended way to handle errors in Python
  - Also used for non-error notification
  - **Example: exception1.py**
- Error codes
  - Some functions return an error code (instead of raising an exception)
  - Common with C/C++ code interfaced to Python
  - Best to wrap these and then raise an exception

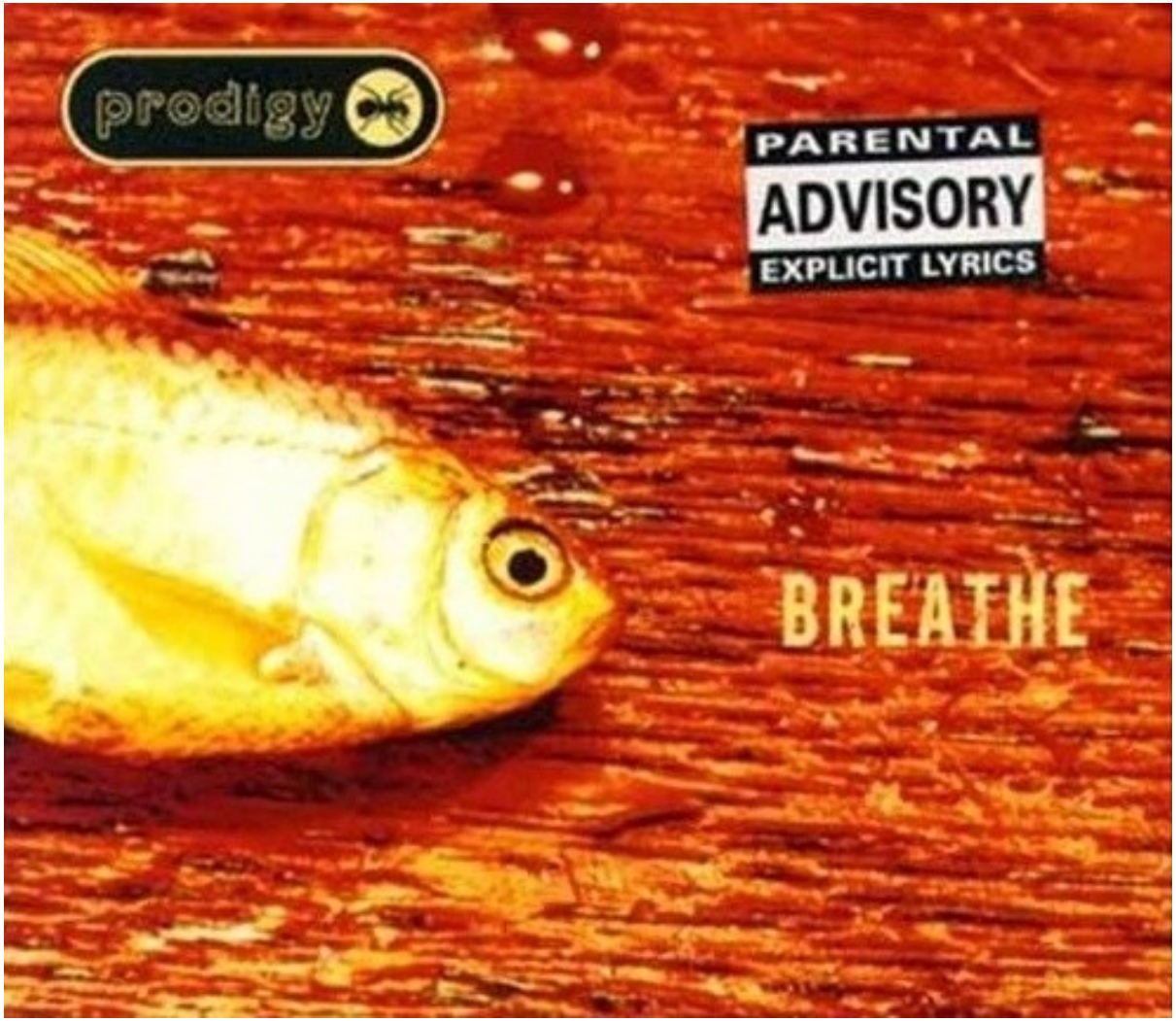
Error handling: reporting errors

- For command line apps:
  - Print to stderr
  - Don't just print errors (to stdout)
- For daemons/services:
  - Dedicated log file for the application
  - Write to the operating system event log(s)
- Use the **logger** module
  - Don't roll your own!
  - http://docs.python.org/library/logging.html

Signal handling

- Support is provided via the **signal** module
  - Can raise signals
  - Can handle incoming signals
- Useful to catch keyboard interrupts
  - eg. interrupt a long running process
- Good form to not ignore system exit events
- **Example: signal1.py**
- **Example: signal2.py**

Let's take a breather...



Testing

- Well-tested = happy users and maintainers
  - Design your app for unit testing
  - **doctest** and **unittest** are two good approaches
  - **nose** (from PyPI) builds on **unittest**
  - **mock** for mock testing
  - **pylint** and **pychecker**: good “lint” tools
- Python Testing Tools Taxonomy:
  - Links to testing libraries and tools
  - http://wiki.python.org/moin/PythonTestingToolsTaxonomy

Documenting your application

- Essential:
  - README.txt (overview)
  - LICENSE.txt (essential)
  - CHANGES.txt (application changelog)
  - User documentation/manual
- Formats
  - Text file
  - HTML (for online or offline use)
  - man page
  - **Example: rsync man page**

Packaging your application

- Another whole tutorial topic!
- Use the standard Python distribution tools:
  - **setup.py**
  - PyPI (for public distribution)
  - http://guide.python-distribute.org/
- Other approaches for specific platforms:
  - Debian package (.deb)
  - RedHat/SuSE/CentOS (.rpm)
  - MSI (Windows)
  - etc.

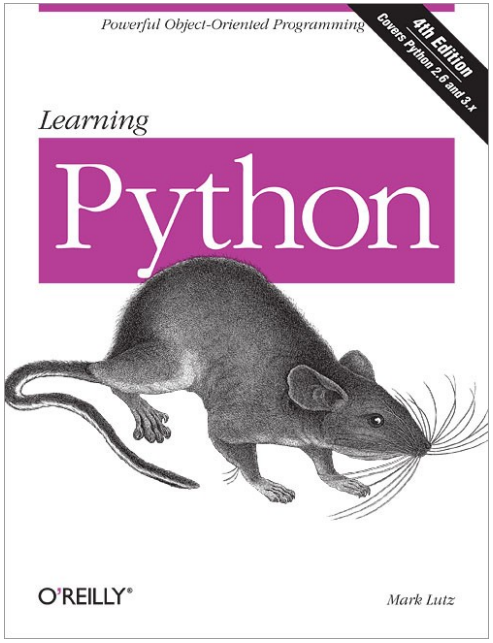
This is the end...

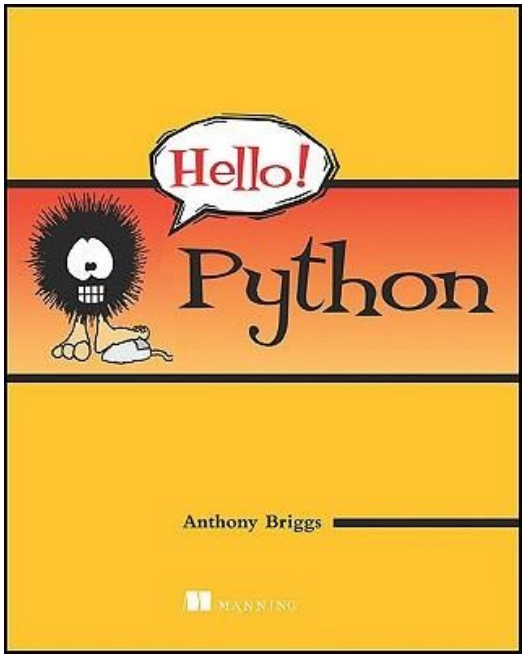
For more information...

- The Python tutorial
  - http://python.org/
- Python Module of the Week
  - http://www.doughellmann.com/PyMOTW/

Some good books…

- “Learning Python”, Mark Lutz
- “Hello Python”, Anthony Briggs





In the beginning

- http://www.cryptonomicon.com/beginning.html