Welcome to SciCoder!

SciCoder Introduction

Demitri Muna

31 July 2017

Optimize Your Code

"We should forget about small efficiencies, say
about 97% of the time:
premature optimization is the root of all evil."

Donald Knuth

the guy who
invented TeX

So important, I made this the first slide.

Software Usability (UX)

"Don't make me think."

Steve Krug

```
plot(x, y, 'r--')
```

unless you're familiar with this, it's not clear;
even if you *are* familiar, your mind has to parse it

```
plot(x, y, color='red', linestyle='--')
```

extremely clear, readable by anyone who's never
used this code

Aims of this Workshop

- Improving your work environment – introducing you to (hopefully!) new tools that will make your work easier.
- Introduce you to good programming practices.
- Show you how to design and program against a database.
- Start to separate "programming" from the details of "syntax".
- Apply what you are learning to a real research project.

Aims of this Workshop

- Feel free to ask questions!
- Much of Day One will be laying the foundation for the rest of the week. You will get the most from this workshop if you get today and tomorrow's material, so stop me if you don't get something.
- We are not tied to a specific timetable, so a discussion or clarification will not throw us off schedule.

Languages

Scripted

- The code you write is run by an interpreter, line by line.
- Syntax errors are found when the interpreter hits that line.
- The text file is the program.
- Much faster to write and experiment in (just type, then run).
- Slower than compiled languages, but modern techniques and computers have vastly narrowed this gap.
- Examples: Python, Perl, JavaScript, shell scripting

Compiled

- The code you write must be compiled, e.g. turned into machine code.
- Syntax errors will prevent the program from being compiled.
- Slower to develop with (must compile, link, run).
- Program are faster (but a poorly written C program can be slower than a well-written Python script!).
- Syntax is typically not very clean.
- Good for low-level programming when you need fine control of memory or direct access to hardware (and you don't).
- Examples: C, C++, Fortran, Java

So, What Language Should I Use?

- There is no one answer – it depends on what you are doing.
- Aim to minimize:
  - your development time
  - complexity of code

} The most important asset is your time!
(Or anyone using/modifying your code.)

- I'd rather spend one day writing code in Python that takes 10 hours to run than one week writing the same thing in C++ that takes 1 hour to run.
- Even for things like Monte Carlo simulations, typically there are one or two routines that take 90% of the time. You save nothing by optimizing the other routines in your program. And you probably don't know which routines they are!
- Always work at the highest possible level.

C

- Don't use it. Seriously.
- C is a bare-bones language. It doesn't come with anything – you have to build everything from scratch.
- C doesn't even know what a string is. You deal with strings *a lot*.
- You have to manage memory yourself.
- C happily lets you read beyond arrays. This is Bad. This is a deal breaker. Don't use arrays.
- C is universal though – just about anything can read a C library. C is then good for writing libraries that will be widely used. But you will pay for it in development time.
- C is highly portable – it will run just about anywhere.
- Bottom line: Only use C to write very portable code, and even then only in the form of libraries. You will rarely need to do this.

C++

- A true object-oriented (OO) language.
- If you don't know the concepts of OO, you can still write a C-like program. This is not a good thing (no gain over using C).
- The syntax is unfriendly and confusing.
- C++ is *strongly typed*.
- The language hasn't been 'updated' in over ten years (and don't hold your breath).
- You have to manage memory yourself. One of the most common types of bug in writing C++ (and C) are related to memory management (leading to increased development time).
- C++ libraries can be linked from other languages (e.g. Python), so also a good choice for portable libraries (but not as universal as C).
- Well-written code will produce programs about as fast as the hardware can run.
- Bottom line: Best language for time-critical code, but at the cost of higher development time and time spent debugging.

Python

- A true object-oriented (OO) language.
- Easy to learn, the syntax is clean and natural.
- Python is *weakly typed*.
- Memory is managed for you.
- The language has *many* built-in features so you don't have to keep reinventing the wheel, and hundreds of more specialized libraries are freely available.
- The language is continually being updated with more functionality.
- For most of what you'll need to do, the code is functionally as fast as C/C++.
- Can link to C/C++ libraries for the most CPU-intensive code.
- Bottom line: Best language for the vast majority of your programming tasks, if for no other reason the short development time (remember, *your* time is more valuable than writing code that's 10% faster!).

Optimize Your Code

- Your computer is not the computer your advisor grew up with.
- Computers are *fast* today. You don't have to worry about a few tens of bytes here and there – people still do this.
- Code readability and reuse is *far* more important than running time for nearly everything you'll do.
- Learn to use profiling for time-critical or CPU intensive code – let the computer tell you the bottleneck, don't guess (we'll come back to this later).

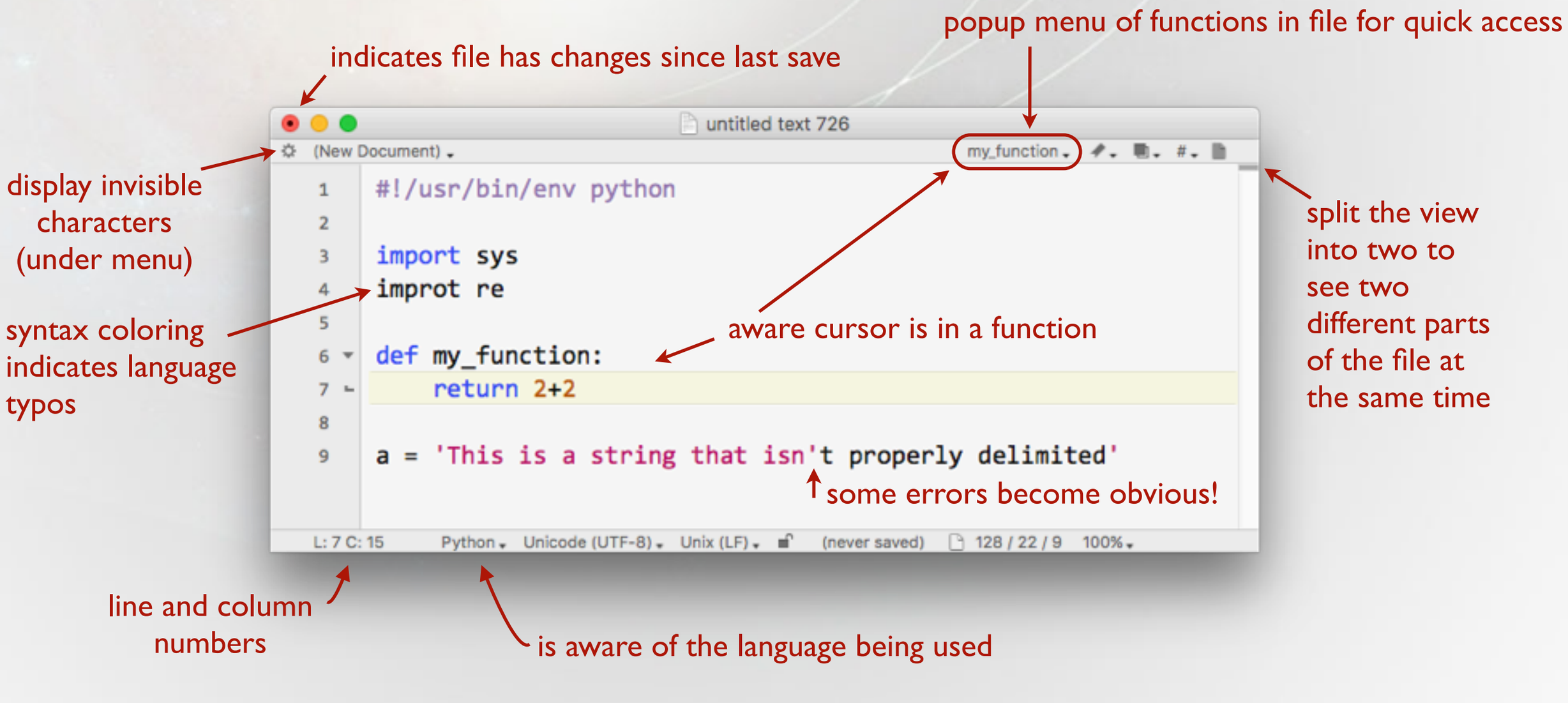
Choose a Good Text Editor

- You spend most of your time coding in a text editor – pick one that is easy to use and can help you.
- The choice of a text editor for some is like a religion. If you've picked yours, at least see what's available.
- Features to look for:
  - syntax highlighting (must have!!)
  - line numbers
  - can execute code directly from the editor (*very* handy)
  - aware of functions
- I recommend not using terminal editors such as emacs/vi for day to day coding. You *should* be very comfortable in one or the other, but there are simpler tools available.

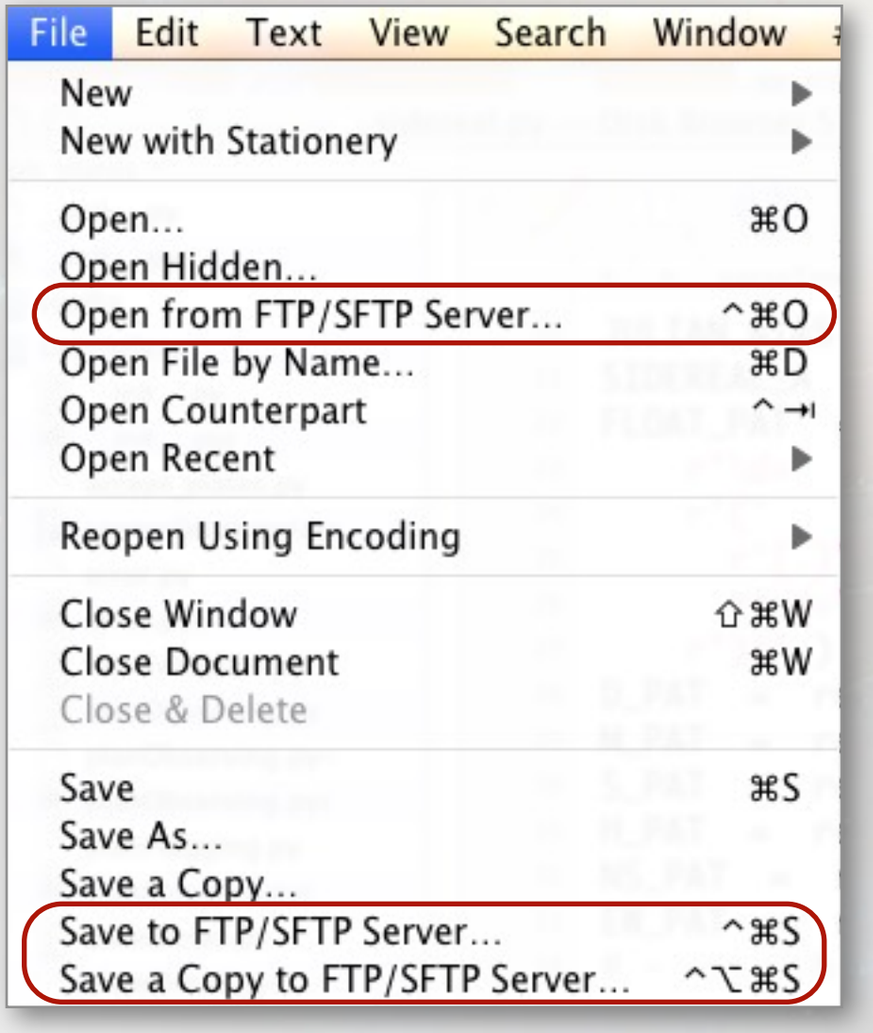
Mac Text Editor

BBEdit http://barebones.com

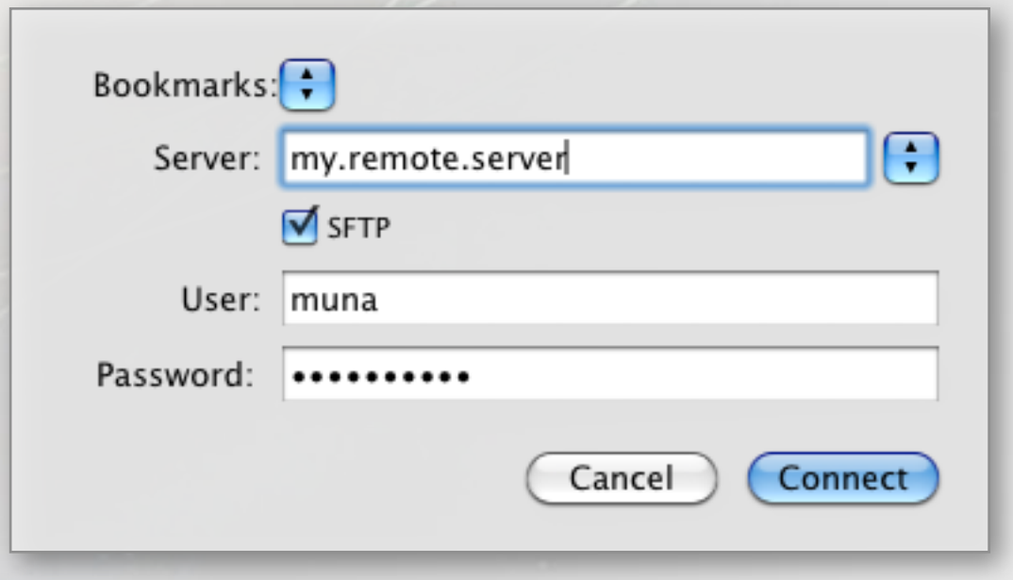
(commercial, but free trial turns into "lite" version)



Edit Remote Files



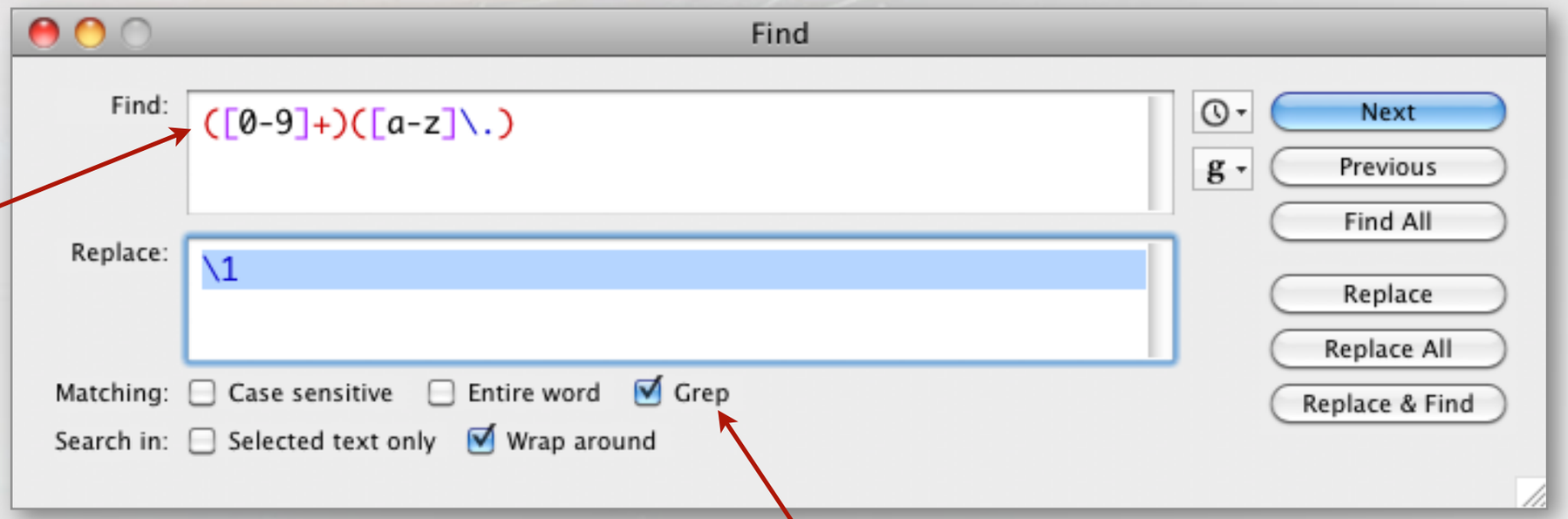
Open a file from your linux server, either locally or anywhere else. Edit it as if it's on your machine. When you save the file, it saves back to the server.



You shouldn't have to download a remote file, edit it locally, and upload it back again.

Regular Expressions in the Editor

A good editor will handle regular expressions in the search and replace panel. When you learn these (which we will later cover), you can often perform certain tasks with this instead of having to write a program.



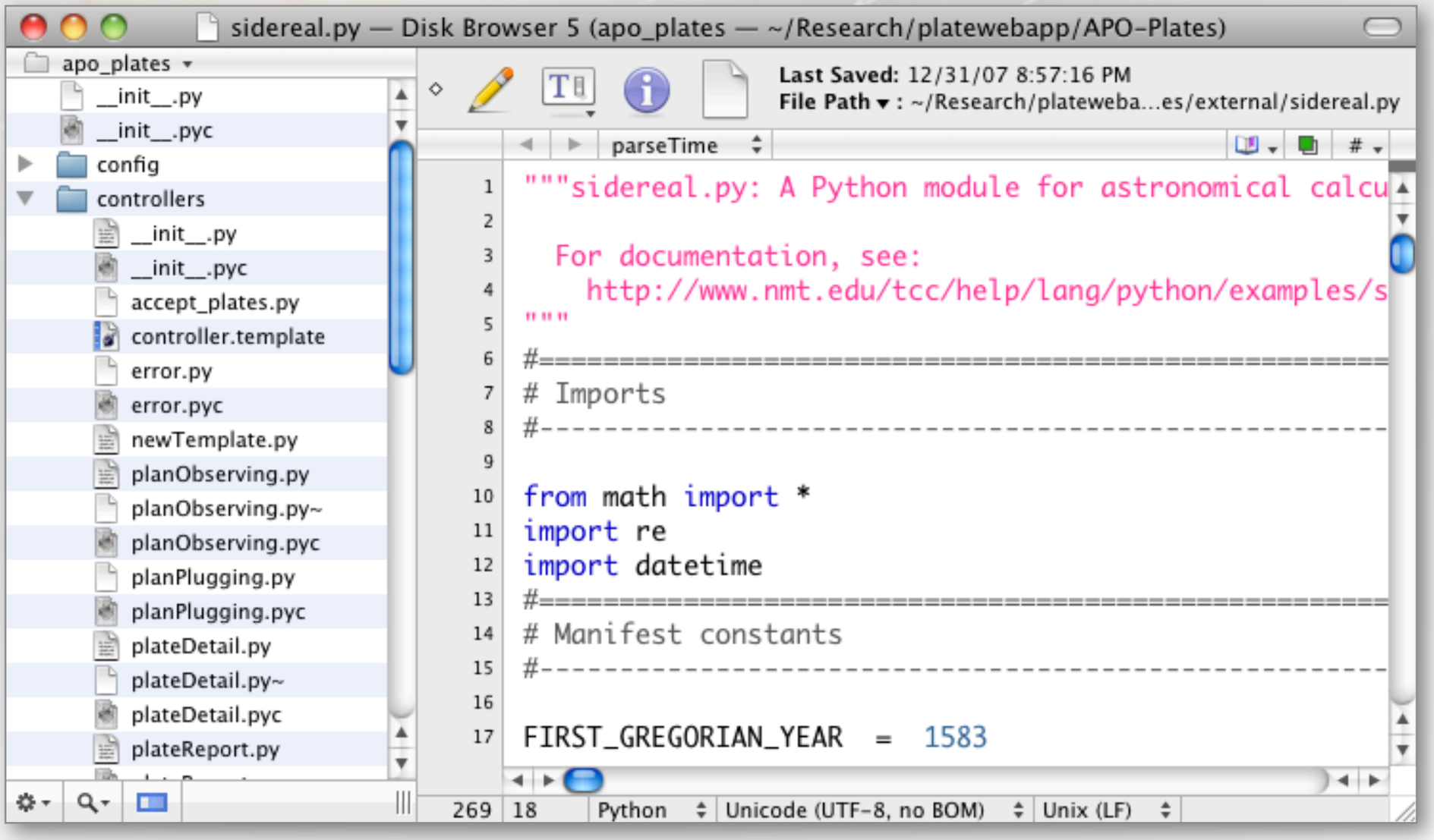
syntax highlighting will indicate whether the regular expression is valid

can use grep (regular expressions) or not

This just scratches the surface – there are many more features we will gradually cover.

Project Editing

Several editors offer "disk browsers" that allow you to access several files at one time. This is similar to an IDE (integrated programming environment) such as Xcode and Eclipse. This eases work in larger projects and offers features such as multi-file search and replace.

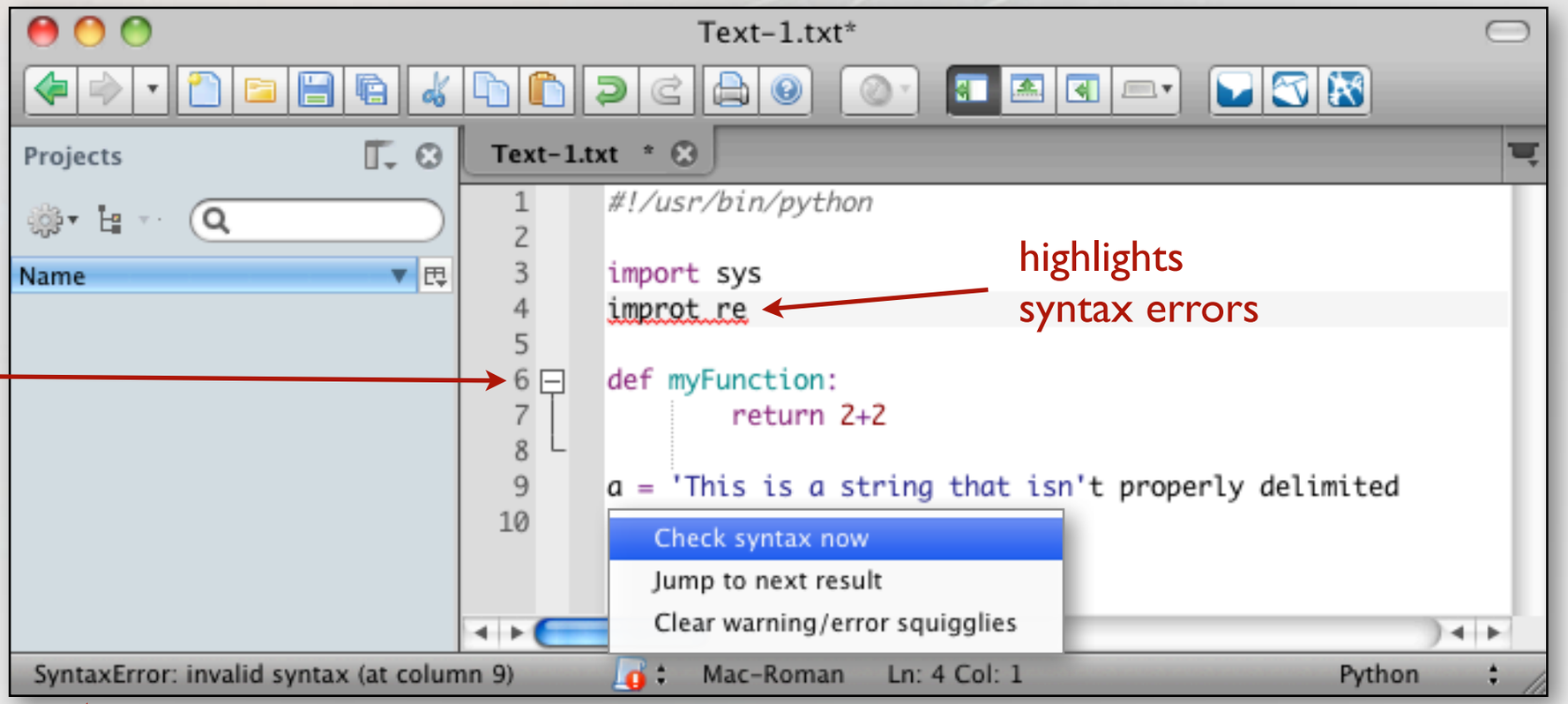


Linux Text Editors

For Linux users (obviously) but also for Mac/Windows users when they are working on a remote server. These applications can be launched over X11 (but I prefer to use a local editor whenever possible).

Komodo Edit http://www.activestate.com/komodo-edit/downloads

features code completion



highlights syntax errors

code folding: collapse long functions for better readability

split the view into two to see two different parts of the file at the same time

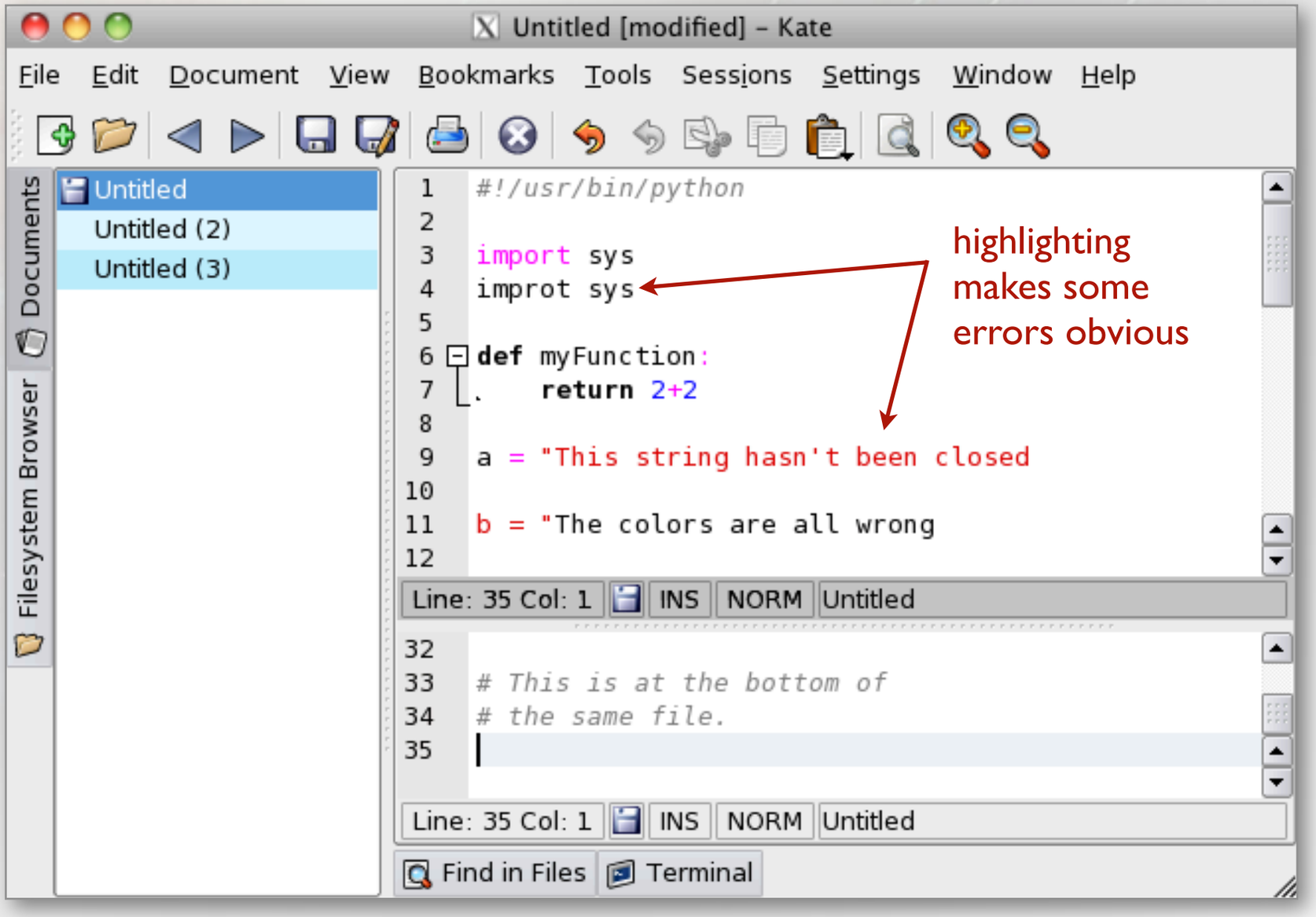
it even tells you the problem

select language here

Available for free on Linux/Mac/Windows.

Linux Text Editors

Kate (part of KDE, install package "kate" in Ubuntu)

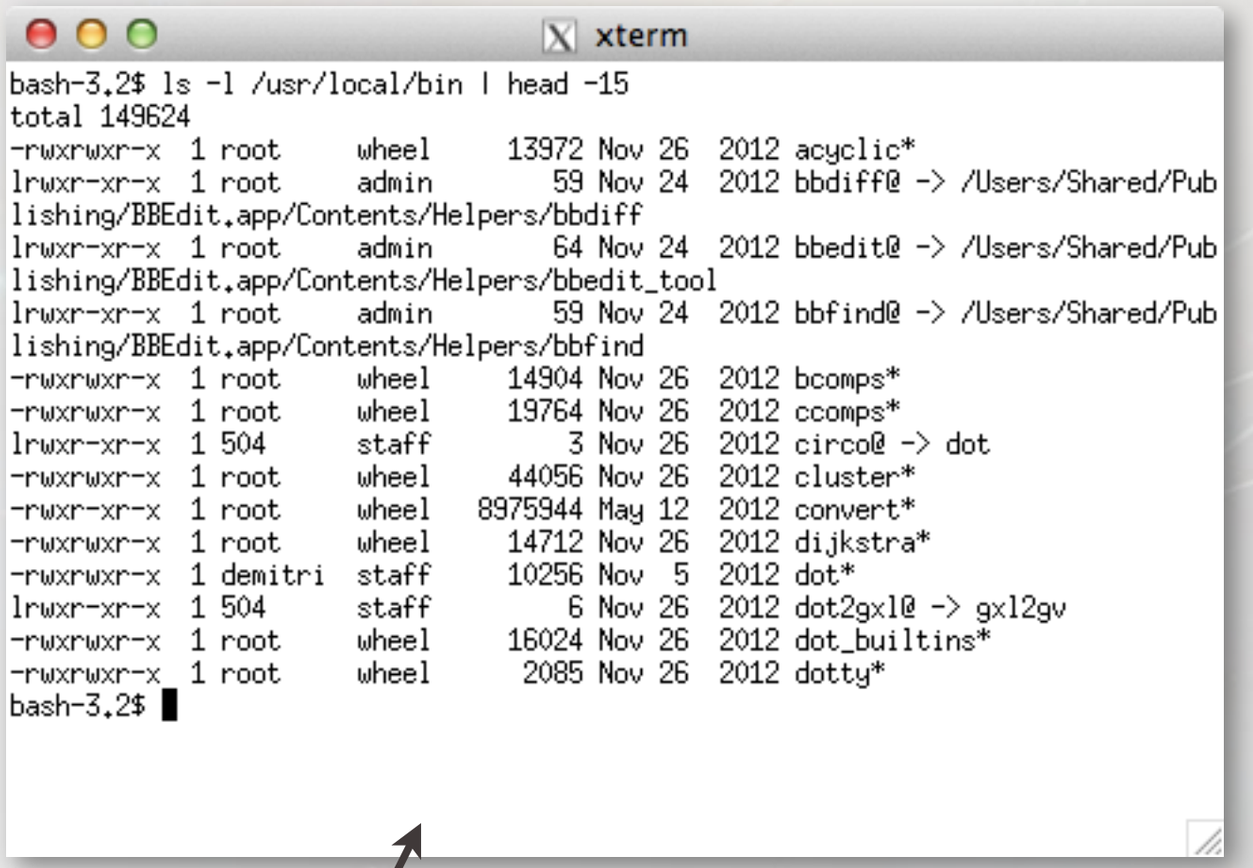


work on multiple files at once

highlighting makes some errors obvious

split view of same file

Your Terminal & You



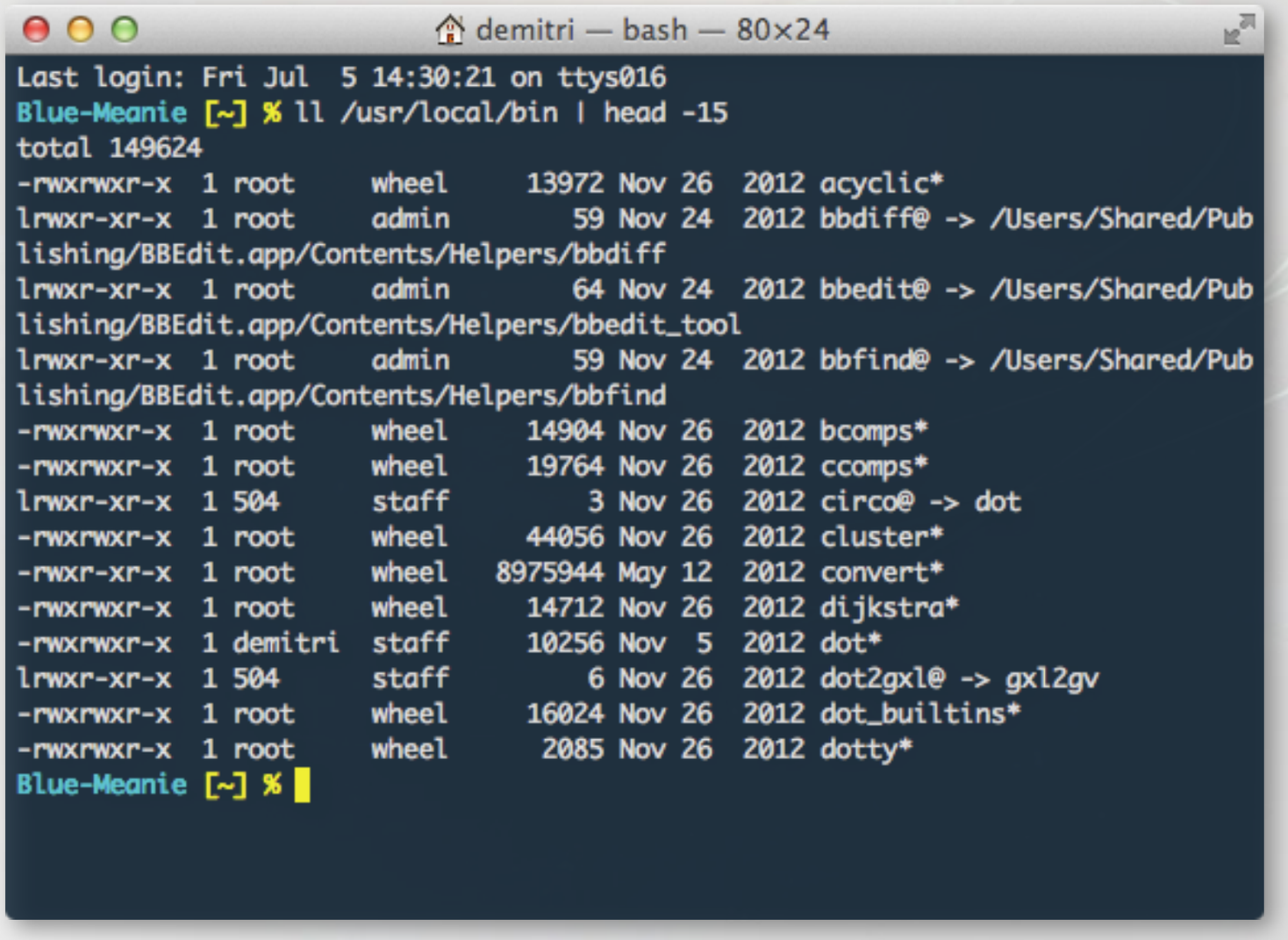
You don't want to look at this all day long. You have better standards than that.

What's wrong with xterm?

- Boring
- Hard to read
- Can't paste text into window
- Pixelated
- Black and white
- Makes small children cry

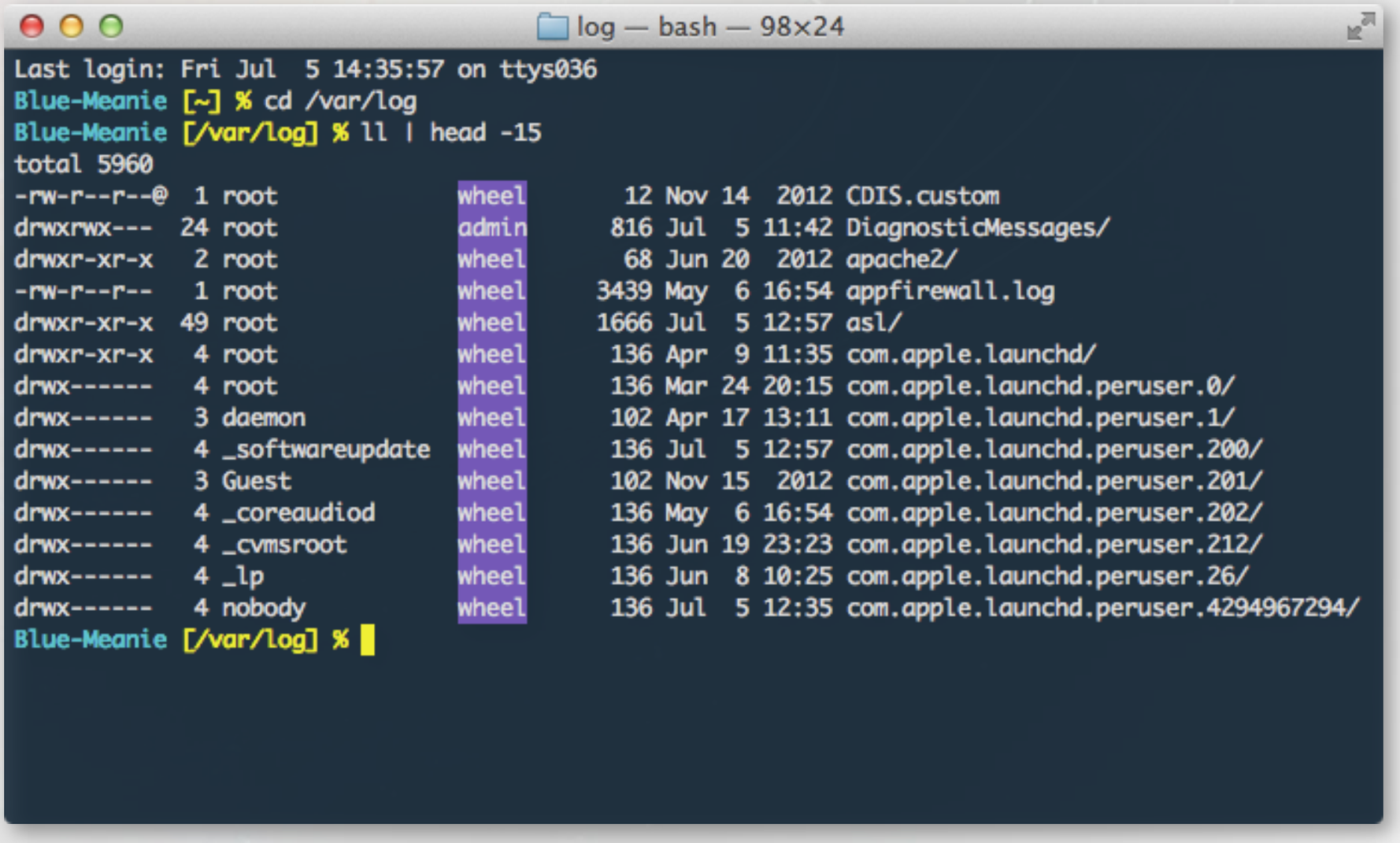
There is NO good reason to use Xterm. Don't do it.

Upgrading Your Terminal



- On Mac OS X, use the Terminal program. It will work seamlessly with any X application.
- Each Linux will have a native Terminal application.
- Copy/paste work with everything (except X applications).
- Set up a color scheme for each server you use on a regular basis – it's a quick visual indicator to see where you are.
- Text is easier on your eyes.
- Color will help guide your eye and find information more quickly.
- Can drag/drop text into window.

Upgrading Your Terminal



Discontiguous text selection!

A Nicer README

- It's always a good idea to put "readme" files with your code. They can describe the overall structure of your code or data and provide a good entry point to how to use the code and how it's organized.
- This is useful for your own code – you forget things.
- Plain text files still dominate – they can be easily read anywhere and are easy to modify by anyone (anywhere).
- But they're bland and boring...

Markup Languages

- A markup language provides a means to indicate decoration (e.g. bold, italics, boxes) with plain text.
- Example languages: HTML, LaTeX

HTML

```
<h3>A header</h3>
<table border="1">
<tr><td align="center">
<b>Audentes</b> fortuna
iuvat.
</td></tr>
</table>
```

A header

Audentes fortuna iuvat.

LaTeX

```
\Psi =
\frac{e^{2/\theta_{0}}}
{(2\pi - 86)(i + 99)}
```


$$\Psi = \frac{e^{2/\theta_0}}{(2\pi - 86)(i + 99)}$$

plain text, but hard to read

easy to read, but not editable

Markdown

- Markdown is a markup language that is easy to read, even with the markup "code".
- Supports lists, styles, links, tables, images, inline code, math equations, more.
- GitHub renders markdown files automatically.
- Files are still plain text, but are indicated with a "`.md`" extension.
- There are editors that provide a split view – the text on one side, and the rendered view in the other.

Editors

Linux: http://uberwriter.wolfvollprecht.de
Mac: http://typora.io • http://www.bear-writer.com
Web: http://benweet.github.io/stackedit/
Mac QuickLook plugin: https://github.com/toland/qlmarkdown
Emacs: https://github.com/jamesnvc/emacs.d/blob/master/modes/multimarkdown-mode.el

Markdown Example

markdown_example.md — 83 Words

```
###Welcome to my README file

__Here are some things you can do__

 * An inline link to [SciCoder](http://
scicoder.org)
 * A raw URL: <http://xkcd.com>
    * nest list items
 * inline code: try the `try` statement!

Indicate block code with four leading spaces:

    for i in range(100):
        print i


_Have a table!_

First Header | Second Header | Third Header
------------ | ------------- | ------------
Content Cell | Content Cell  | Content Cell
Content Cell | Content Cell  | Content Cell

Equations are surrounded by "$$":

$$\Psi = \frac{e^{2/\theta_{0}}}{(2\pi - 86)
(i + 99)}$$
```

Welcome to my README file

Here are some things you can do

- An inline link to SciCoder
- A raw URL: http://xkcd.com
    - nest list items
- inline code: try the `try` statement!

Indicate block code with four leading spaces:

```
for i in range(100):
    print i
```

Have a table!

First Header	Second Header	Third Header
Content Cell	Content Cell	Content Cell
Content Cell	Content Cell	Content Cell

Equations are surrounded by "$$":

$$\Psi = \frac{e^{2/\theta_{0}}}{(2\pi - 86)(i + 99)}$$

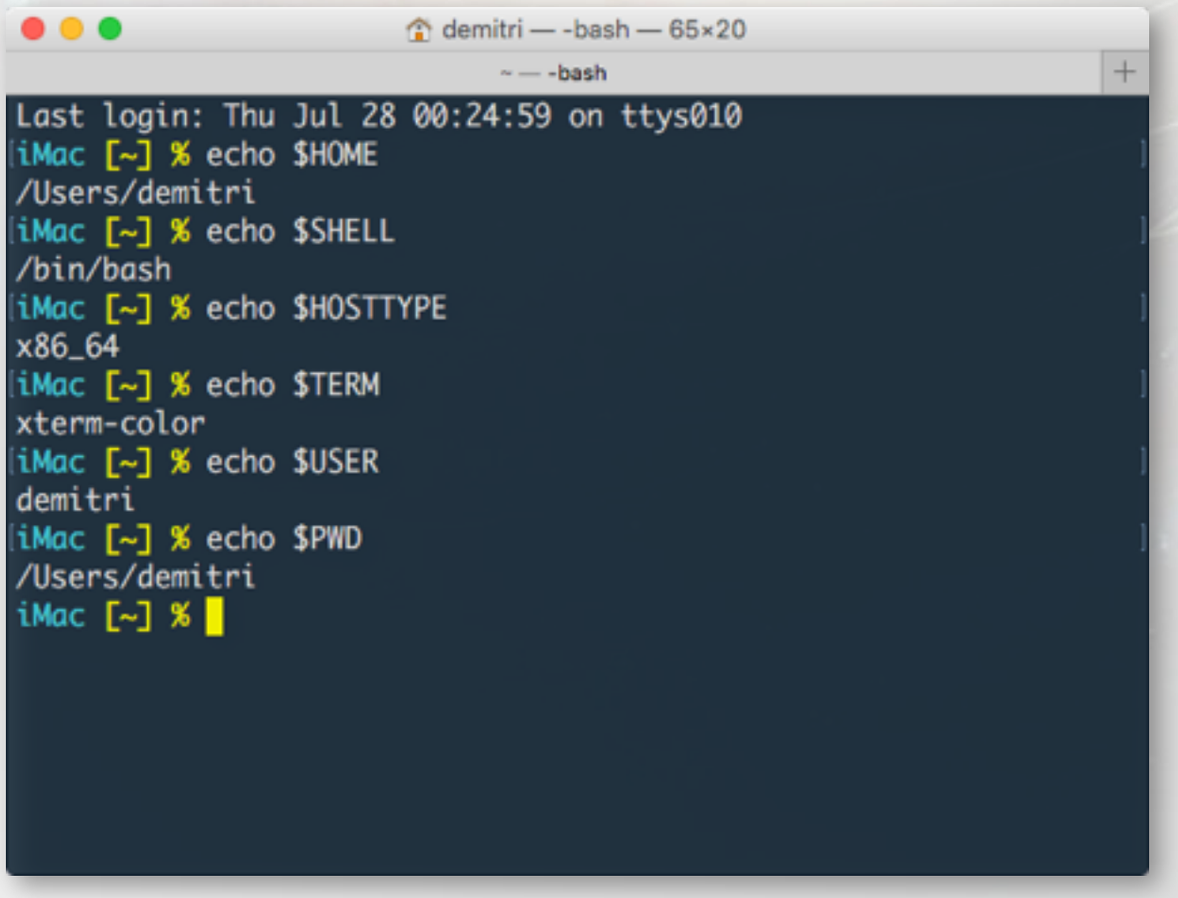
Edit this side

Live updates here

Environment Variables

The program that runs when you use the command line is called a "shell". Most people use bash (`/bin/bash`), some use the C shell (`/bin/csh`) or T-shell (`/bin/tcsh`).

Bourne*-again shell



This is different from the terminal (GUI) program (e.g. Terminal.app on macOS or xterm**). The terminal emulates a 1960/70s era terminal to a mainframe, the shell is the program you are interacting with.

You can define variables in your shell. Many are defined automatically for you. Variables are accessed by prepending a "$" to the name. Enter `env` on the command line to see all of the variables currently defined.

Environment variables do not have to be in all caps, but almost always are by convention.

* not Jason
** don't use xterm ever; you're better than that

Defining Environment Variables

How to define an environment variable varies by shell.

bash

```
% export NEW_VARIABLE="this is a new variable"
% echo $NEW_VARIABLE
this is a new variable
```

csh/tcsh

```
% setenv NEW_VARIABLE "this is a new variable"
% echo $NEW_VARIABLE
this is a new variable
```

Appending to existing variables.

bash

```
% export PATH=${PATH}:/usr/local/bin
```

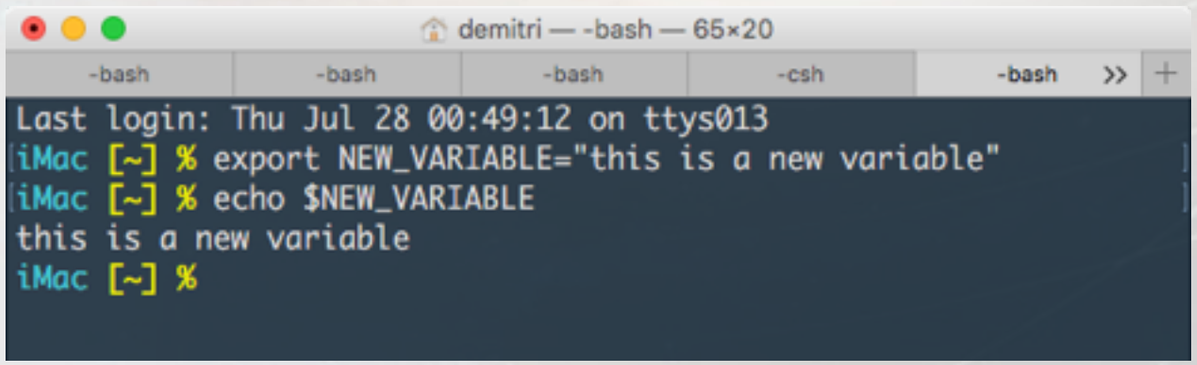
csh/tcsh

```
% setenv PATH "$PATH:/usr/local/bin"
```

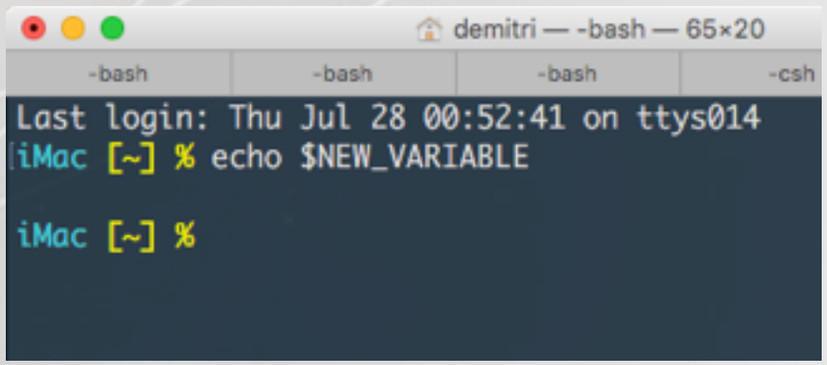
Environment Variable Scope

Once defined, the variable is only defined *in that shell*.

Define variable here...



Open new shell, variable not defined.



If you want certain variables to always be defined, create them in your shell startup files (e.g. `~/.bashrc`, `~/.bash_profile`, `/.cshrc`).

Better, create a dedicated file for all of your environment variables, then source that from your shell startup script.

The $PATH Environment Variable

When you type a command in the shell, how does the system know where to find the program?

```
iMac [~] % echo $PATH
/usr/local/anaconda/bin:/usr/local/bin:/
usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/
bin:/Developer/Platforms/
iPhoneOS.platform/Developer/usr/bin:/
Developer/usr/bin/:/usr/local/postgresql/
bin:/usr/local/wget/bin:/usr/local/
wcstools/bin:/usr/local/mangle/bin:/usr/
local/sloccount/bin
iMac [~] % type -a python
python is /usr/local/anaconda/bin/python
python is /usr/bin/python
```

The $PATH variable contains a list of colon-separated paths. When you type in a command, the system looks for that program in the first path, then the second, and so on until it finds the program.

Do you have several `python` programs on your computer? The one that will run will be the first found in $PATH.

two 'python' programs found in all paths;
the one in /usr/local/anaconda/bin will be used

When To Use Environment Variables

One common use for environment variables is to define one for a directory path, e.g.

```
% export DATA='/usr/local/my_data_directory'
```

You might have this data in different locations on different computers (e.g your laptop, remote server). Define the variable on each machine, then have your code access the location via the environment variable. Further, if the location of the data changes, just change the variable once – not your code.

Another use is to store passwords, e.g. database passwords. These should never be located in code (particularly code checked in to a repository!). Make sure the file that declares these variables is only readable by you and no one else.

When Not To Use Environment Variables

Don't use environment variables to be lazy.

If you are distributing code that needs the location of a file or a directory, don't ask the user to create variables that point to those locations.

Your code should get this information through parameters on the command line.

Installing Anaconda

- Anaconda is a Python distribution `https://www.continuum.io/downloads`
- contains the Python executable
- includes a large number of packages
- comes with a package manager called "conda"

There are two installers – a command line version and a GUI installer. I use the former as I can specify where I want the installation to go. Once downloaded, use this to install:

```
% bash Anaconda3-4.4.0-MacOSX-x86_64.sh
```

Since you can install anywhere you like, you can put a full Python distribution in your home directory – useful when you don't have full access on a server but want your own Python.

Python 2 or Python 3?

You should be using Python 3. Use Python 2 only when you have to or code that you are running doesn't support 3. For this reason, I install both Python installations (2 & 3) separately into:

Python 3: `/usr/local/anaconda`

Python 2: `/usr/local/anaconda2`

Use the Unix `type` command to see how many Python executables are on your $PATH:

```
% type -a python
python is /usr/local/anaconda/bin/python
python is /usr/bin/python
```

what will be run when I type "python"

the built-in system Python - leave this alone!

Use the which command to see which will be called when you type "python".

```
% which python
/usr/local/anaconda/bin/python
```

To use Python 2, either call `/usr/local/anacaonda2/bin/python`, or better, create an alias:

```
% alias python2='/usr/local/anaconda/bin/python'
```

Updating with conda

The Anaconda manager "conda" can update Python packages, including itself:

```
% sudo conda update conda
Fetching package metadata .........
Solving package specifications: .

Package plan for installation in environment /usr/local/anaconda:

The following packages will be UPDATED:

    conda: 4.3.22-py35_0 --> 4.3.23-py35_0

Proceed ([y]/n)? y

conda-4.3.23-p 100% |######################| Time: 0:00:00   4.10 MB/s
```

"sudo" is needed since I installed Anaconda in /usr/local

conda will also install any dependencies if needed.

It can install packages (but isn't aware of everything):

```
% sudo conda install seaborn
Solving package specifications: .

Package plan for installation in environment /usr/local/anaconda:

The following NEW packages will be INSTALLED:

    seaborn:  0.8-py35_0

The following packages will be UPDATED:

    anaconda: 4.2.0-np111py35_0 --> custom-py35_0

Proceed ([y]/n)?

anaconda-custo 100% |###############################################| Time: 0:00:00   3.33 MB/s
seaborn-0.8-py 100% |###############################################| Time: 0:00:00   3.79 MB/s
```

Install with pip

If conda isn't aware of a package, you can install with "pip". However, be sure to use the "pip" that's part of Anaconda or you will install the package in the wrong place!

```
% /usr/local/anaconda/bin/pip install astropy
```

Again, you can check to see which "pip" is called by default:

```
% which pip
/usr/local/anaconda/bin/pip
```
we're ok

```
% which pip
/usr/local/bin/pip
```
we're installing into the OS version of Python - bad