Welcome to SciCoder!

SciCoder Introduction

Demitri Muna

1 August 2016

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!).

Perl

- Another scripted language, very popular on the Internet.
- Largely supplanted by Python due to the fact that the syntax isn't as clean.
- More external libraries than Python (but Python catching up every day).
- *Very* similar to Python – you can easily learn Perl after knowing Python (or vice versa).
- Best implementation of regular expressions in a programming language.
- Has OO features, but to use true OO the syntax is awkward (added on later, not part of initial design).
- Perl 6 update has taken over 10 years to arrive (and hasn't yet); people just moved to Python.
- Bottom line: Use Python. However, so much code was written in Perl that you'll probably come across some. Worth learning the basics to at least be able to read Perl code and maybe make some modifications.

IDL

personal rant

- An anchor around the astronomical community's neck.
- Thin veil over C (but still has FORTRAN/VAX-isms!), and it doesn't hide nearly enough.
- The language has not been updated in *years*, and lacks now-common data structures such as dictionaries (a.k.a hashes, maps, associative arrays).
- ITT (the company) has no real interest in making the astronomical community happy.
- IDL costs *a lot*. Scientific computing should be *open*.
- The features that made IDL so appealing in the late 70s and 80s (visualization without complex programming, manipulating arrays as first-class objects) have been far surpassed by modern languages.
- The syntax is awkward. Graphics are ugly.
- IDL will frequently silently fail, so a lot of IDL code is checking that things worked as you expected them to.
- A large number of astronomical libraries exist in IDL.
- Bottom line: Move away from IDL when you can. As a community, we should move to convert the libraries we have into Python (people are doing this, but we need to be better coordinated). As best you can, don't write any new IDL code. When converting code, use OO concepts rather than a one-to-one conversion of routines.

Your professors used to use VAX when they were students. 'Nuff said.

Examples of Why IDL is Bad

```
openr, lun, 'data.txt', /get_lun
```

I have to keep track of file numbers? You're a computer, *you* do it!

```
print, a
```

Why can't I type the variable alone to print out the value? Typing "print," wastes time.

Can't create an empty array. There is such a thing as an empty set.

If you modify a source file, you have to remember to recompile it. You're a computer, notice it yourself.

When you resize a plot window, the plot doesn't redraw. This is something the Mac can do. Since 1984.

Objects (as in OO) are possible, but the syntax and handling is awkward at best. And no one does it anyway.

No database connectivity.* *Unless you pay more. And use ODBC (a niche Windows technology). This is a deal breaker - database access is not an advanced feature!

So as not to be wholly negative, here are two tips. Always place this at the beginning of every IDL procedure you write.

```
compile_opt idl2
compile_opt logical_predicate
TRUE = 1
FALSE = 0
```

- Defaults ints to 32-bit instead of 16.
- Does not allow use of () to access arrays.
- Non-zero values evaluate to true, zero evaluates to false.
- Use the TRUE and FALSE variables for readability.

```
IF (NOT TRUE) ...
IF (~TRUE) ...
```

If this is in your code, it doesn't do what you think it does (bitwise operator).

This is the boolean NOT operator.

And that's all I'll say about IDL this week.

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.

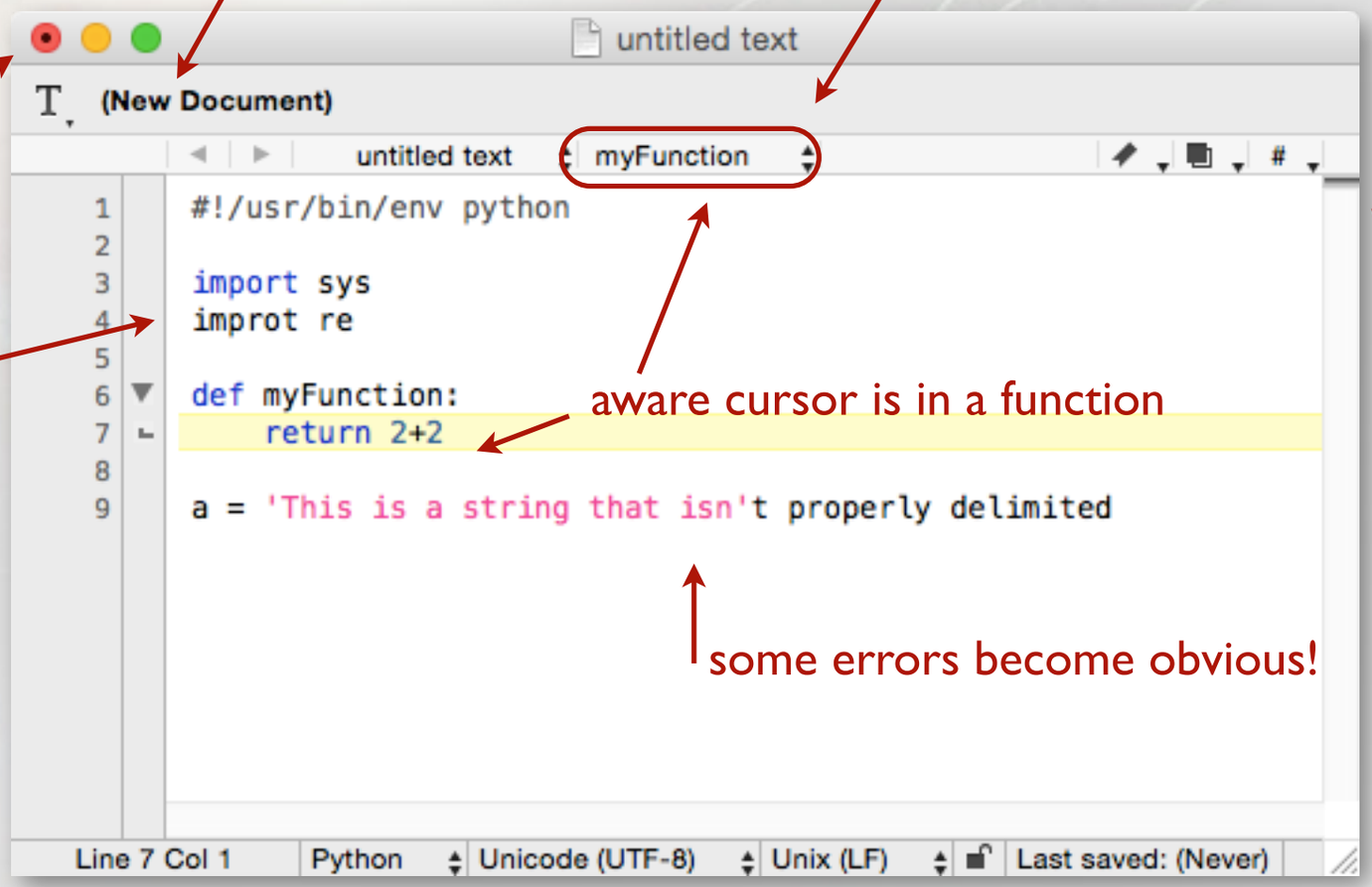
Mac Text Editor

Text Wrangler http://barebones.com

display invisible characters (under menu)

popup menu of functions in file for quick access

indicates file has changes since last save



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

syntax coloring indicates language typos

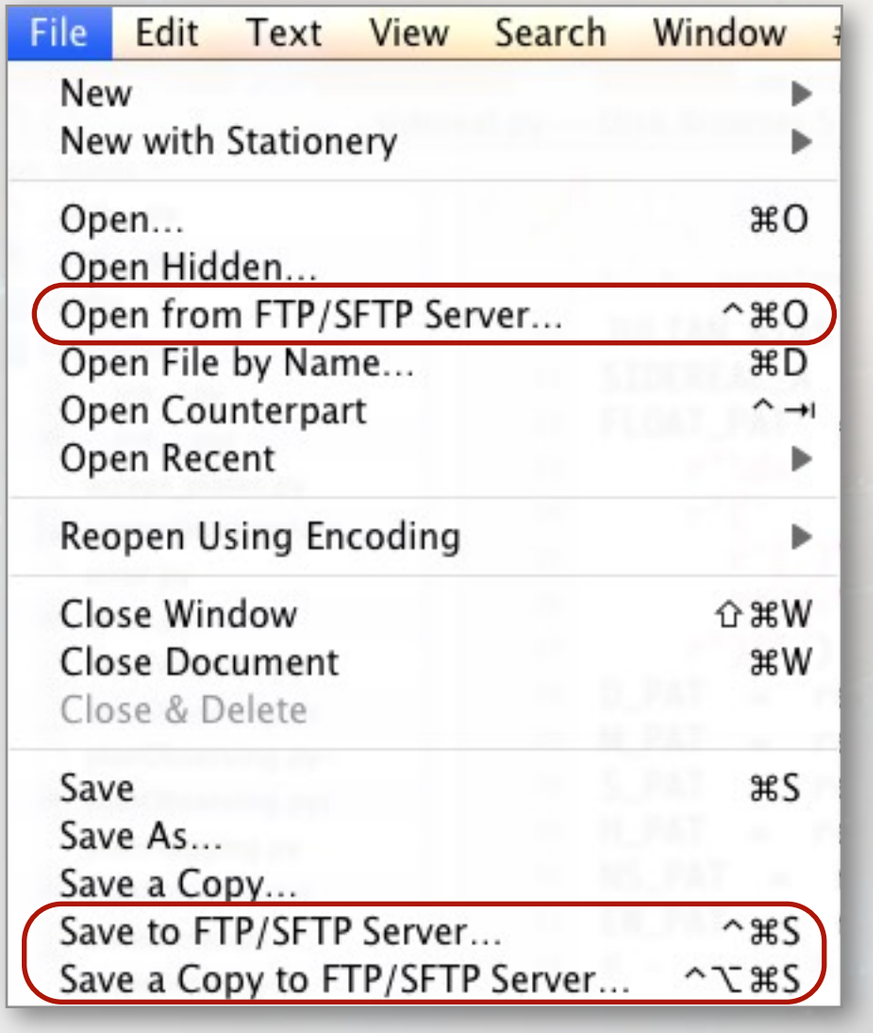
aware cursor is in a function

some errors become obvious!

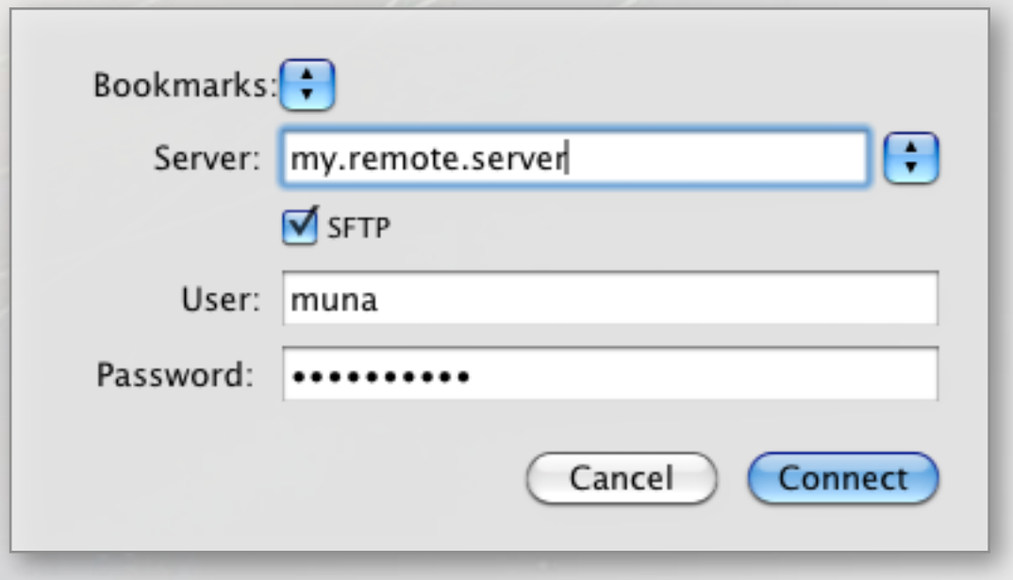
line and column numbers

is aware of the language being used, an IDL plug-in is even available (ask me if interested)

Edit Remote Files



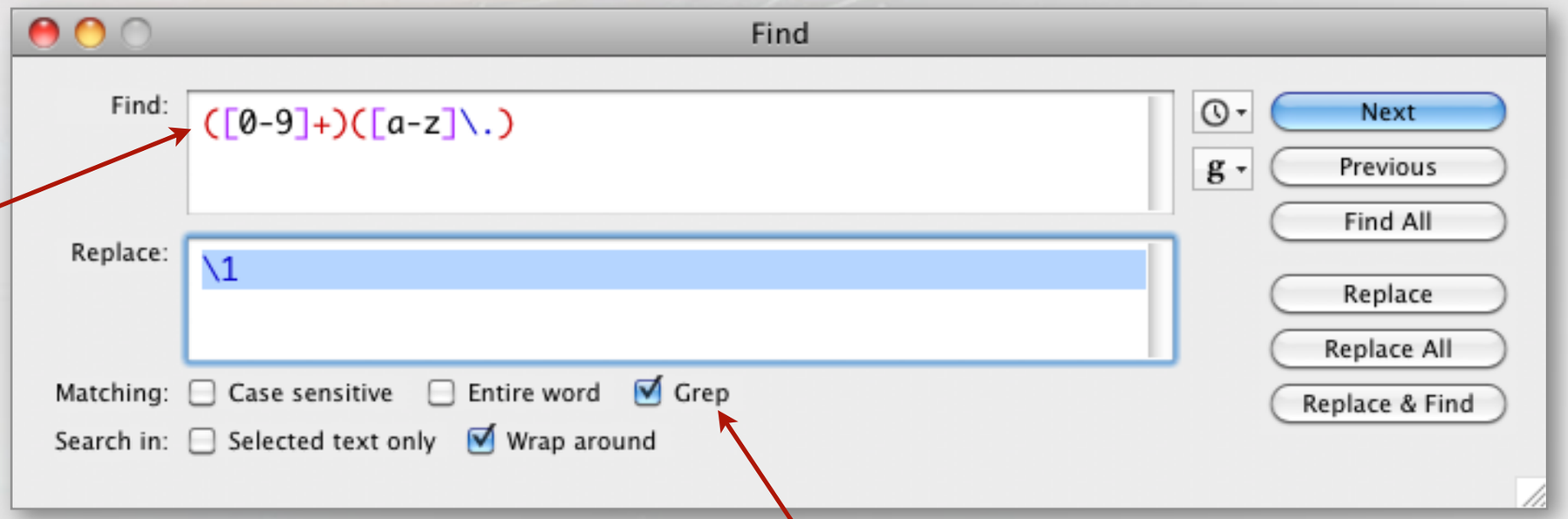
Open a file from your linux server, either locally or anywhere else. Edit it as if it's on your machine. When you save the file, it saves back to the server.



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

Regular Expressions in the Editor

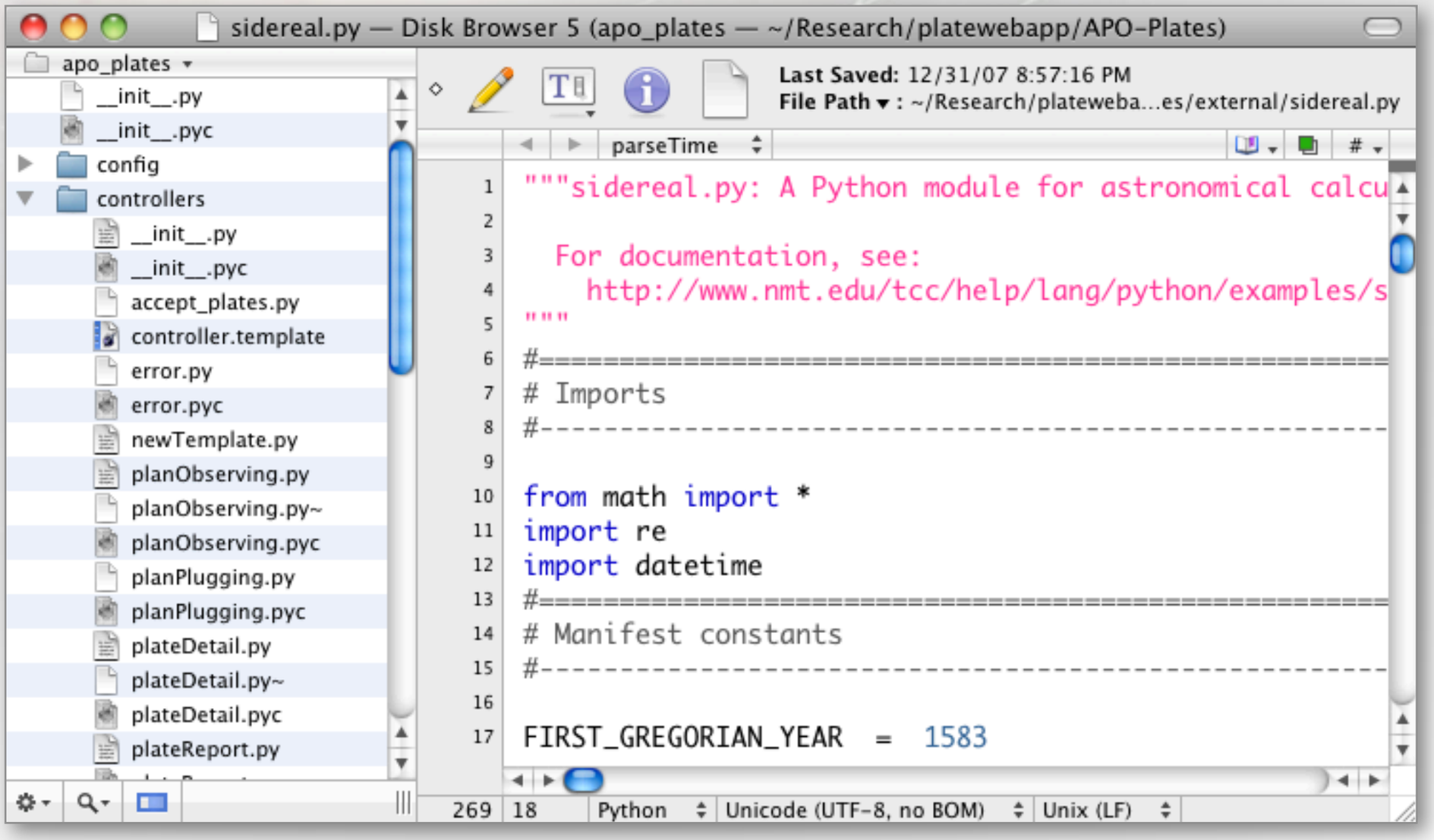
A good editor will handle regular expressions in the search and replace panel. When you learn these (which we will later cover), you can often perform certain tasks with this instead of having to write a program.



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.

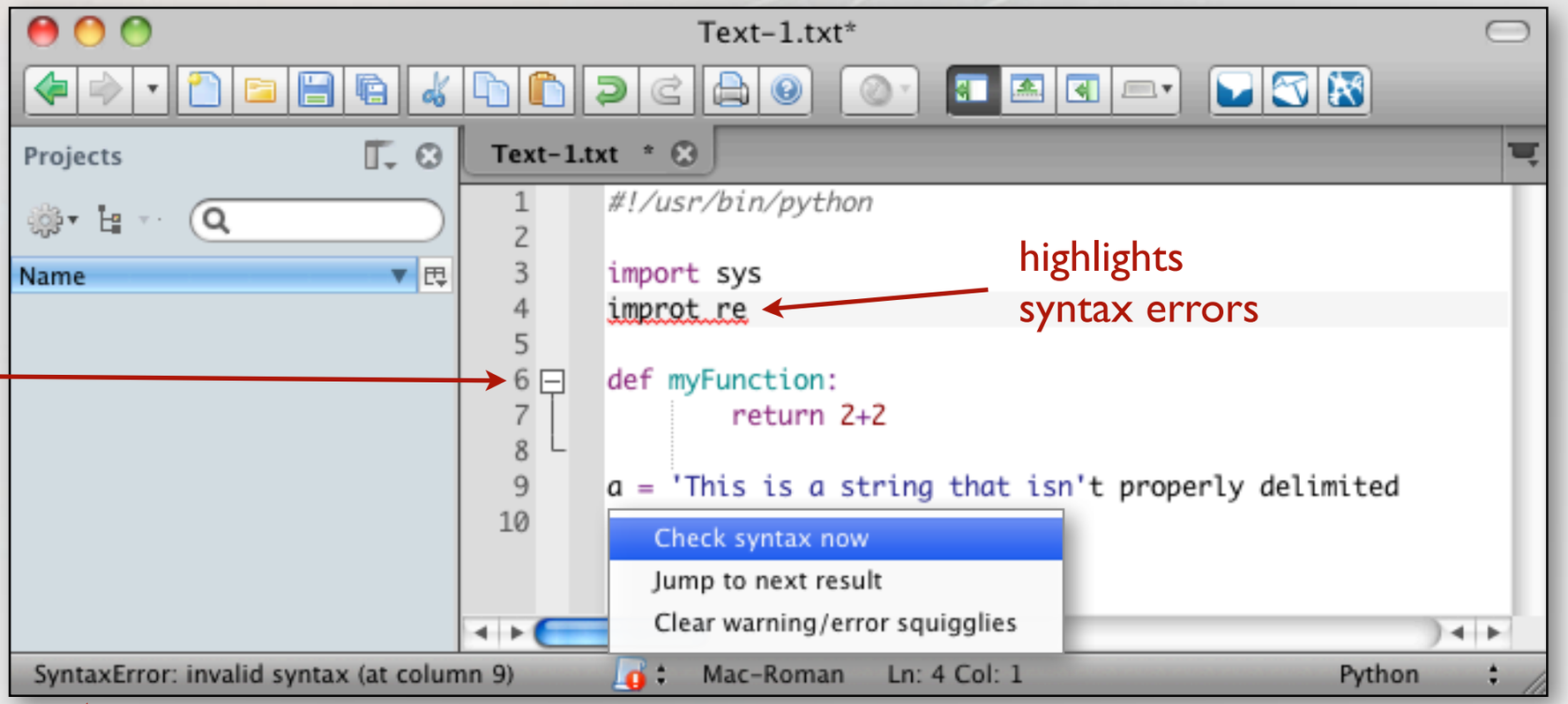


Linux Text Editors

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

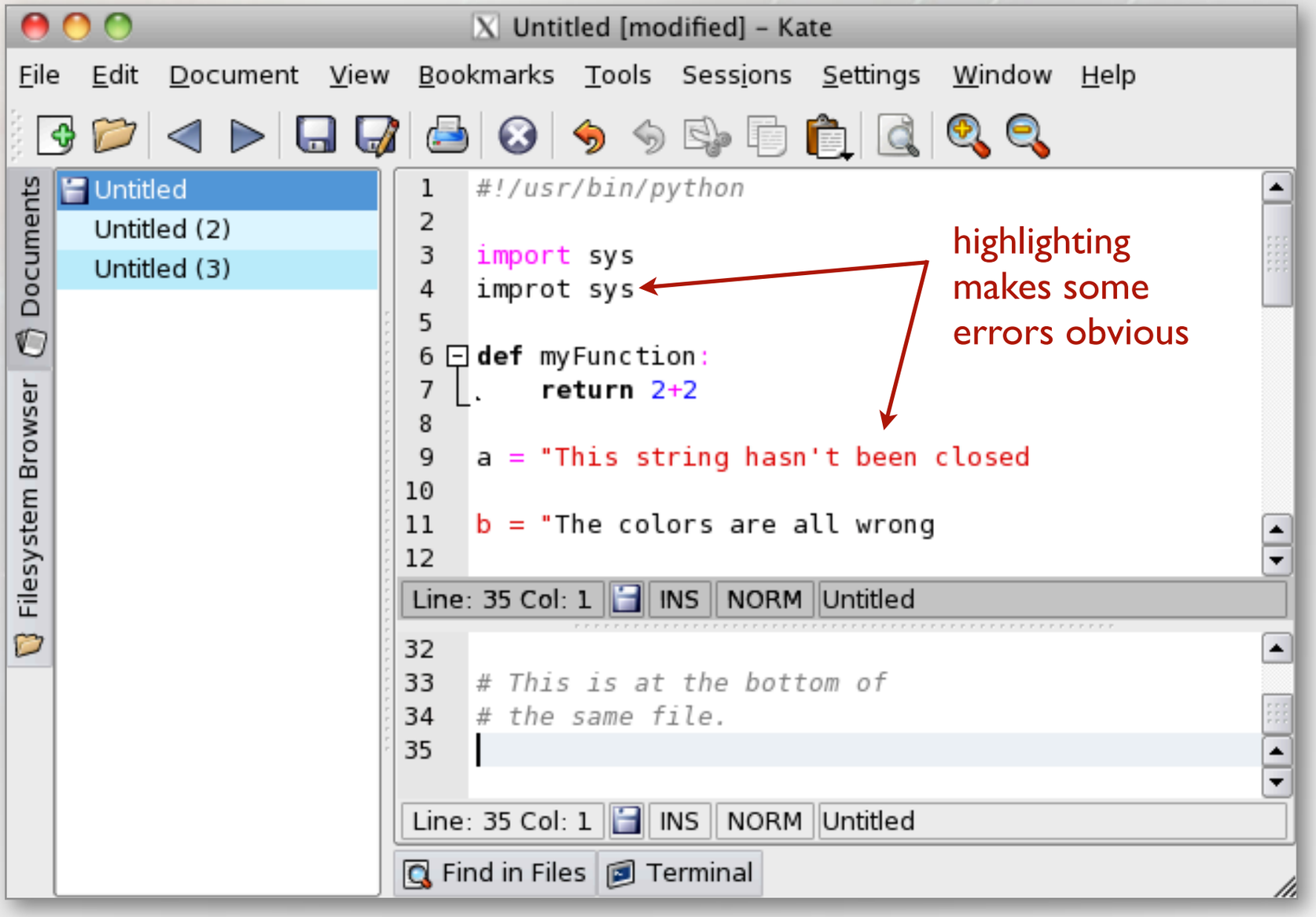
it even tells you the problem

Available for free on Linux/Mac/Windows.

select language here

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 "Everything Box"

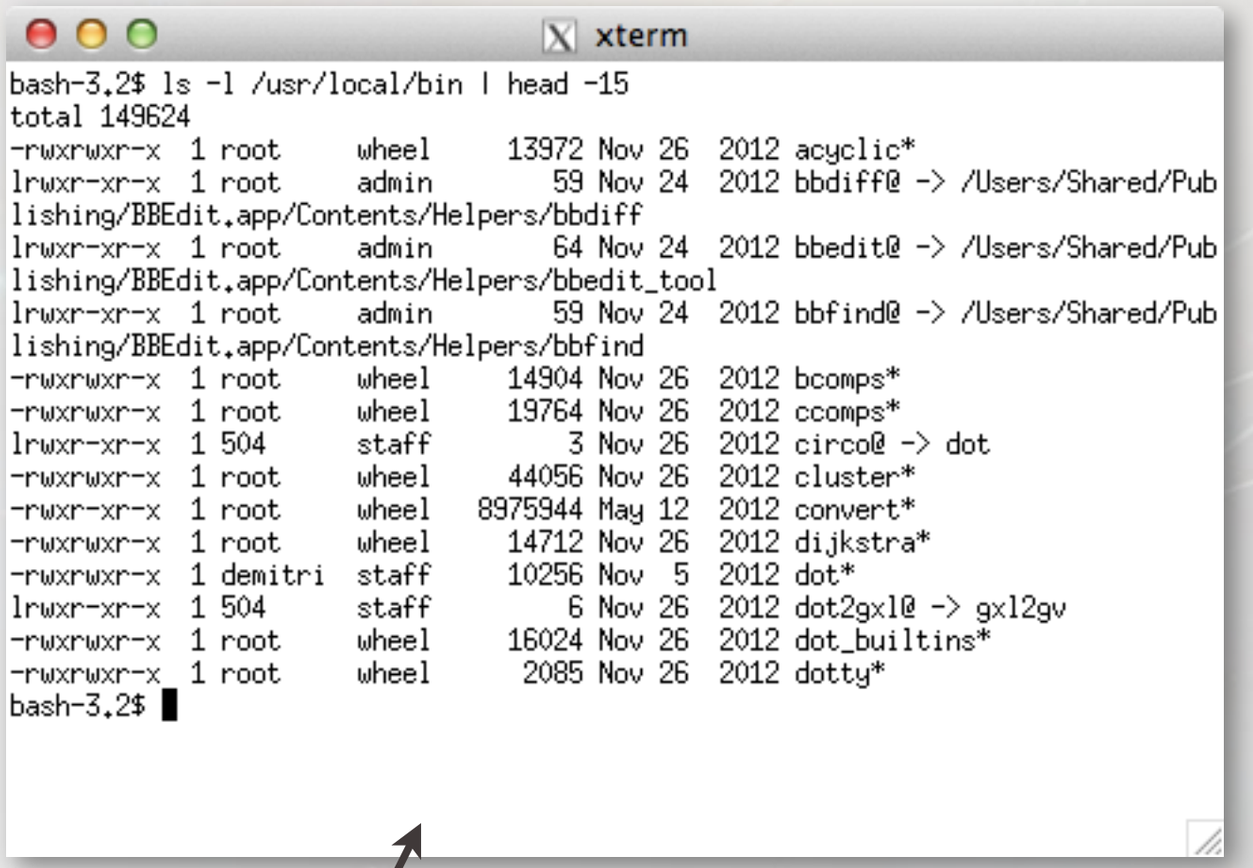
We gather tons of information that takes the form of small snippets of data: web bookmarks, locations of data or servers that we use, frequent flier numbers, etc. It doesn't make sense to save each of these things in a text document on our hard drive — that solves the problem of saving it, but not finding it quickly or easily. And it generates more work to keep it organized (which most people don't do, so data gets lost).

An "everything box" is a program that accepts all kinds of data like this — text, pictures, passwords, etc.

You can tag the information so you can find things in a way that makes sense to you (even using multiple tags).

Evernote is what I currently recommend.

Your Terminal & You



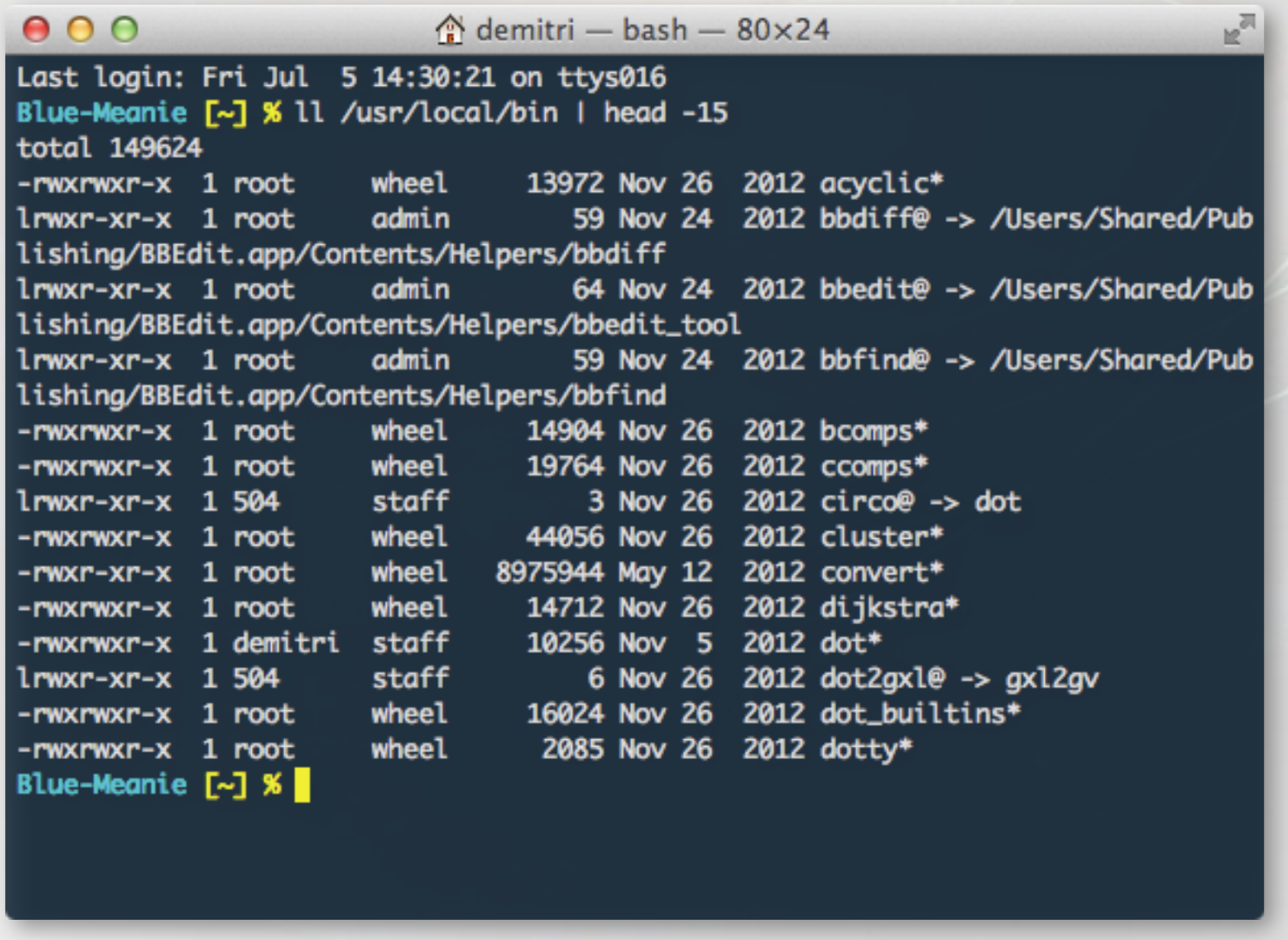
You don't want to look at this all day long. You have better standards than that.

What's wrong with xterm?

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

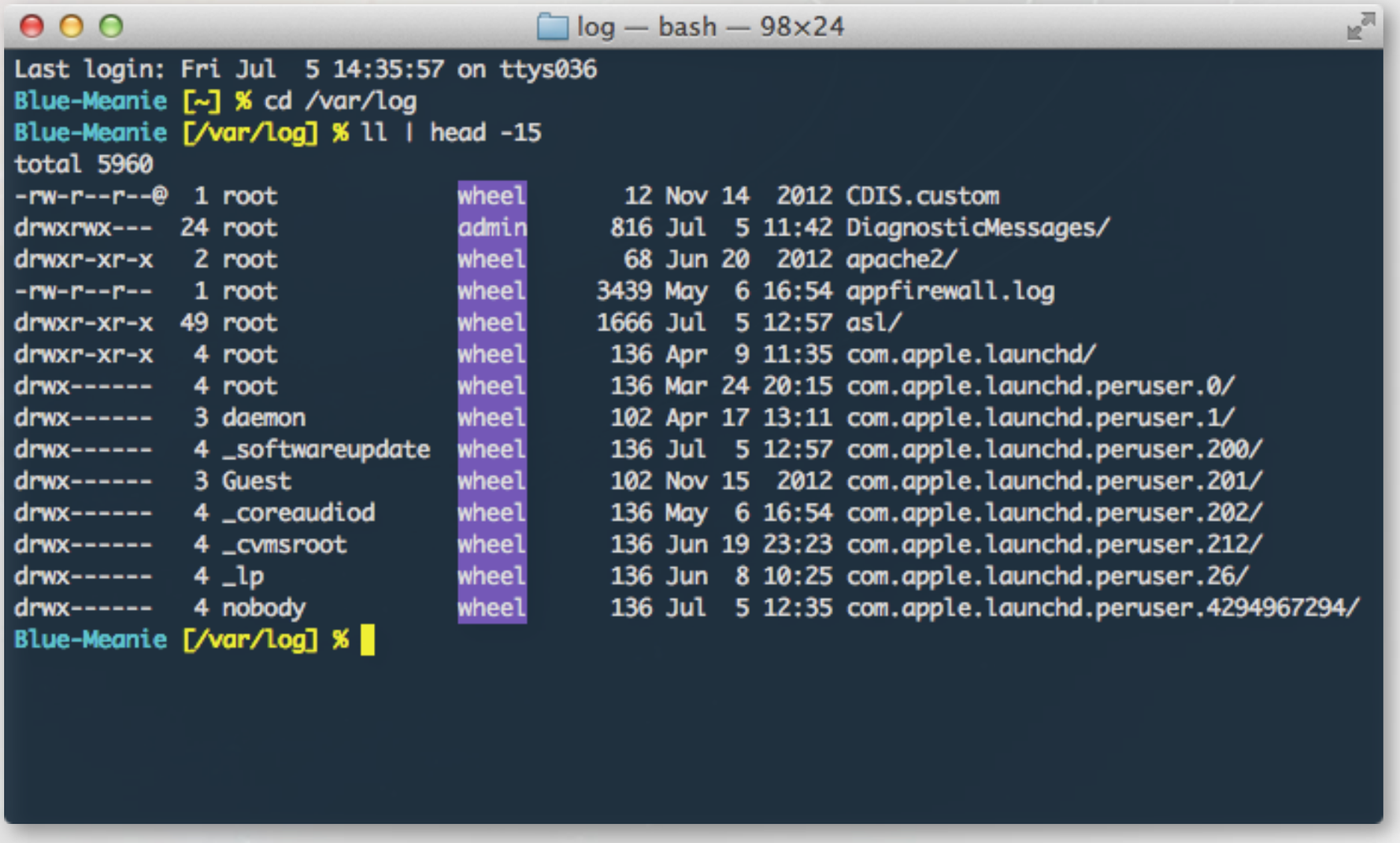
There is NO good reason to use Xterm. Don't do it.

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!

Managing SSH

Standard way to connect via SSH:

```
% ssh muna@bender.osu.edu
```

SCP to the same machine:

```
% scp file.txt muna@bender.osu.edu:
```

Often servers are behind a gateway server, so to log into them from outside your institution, you have to SSH into two machines:

```
% ssh muna@gateway.osu.edu
% ssh muna@bender.osu.edu
```

This can turn into a lot of typing for common tasks, particularly if the names are long. You may also have to remember different user names or port numbers on different machines.

Managing SSH

Create a file called `config` in the `.ssh` directory in your home directory:

```
Host bender
    User muna
    HostName bender.osu.edu
    Port 1234
    ForwardX11Trusted yes
    ForwardAgent yes
    Compression yes
```

set if the port is non-standard (i.e. not 22)

open any remote X11 application on your computer

carry your SSH key authentication to other hosts you connect to

```
Host benderosu
    User muna
    Hostname bender.osu.edu
    Port 1234
    ProxyCommand ssh -q muna@gateway.edu nc %h %p
```

first connect to the gateway server, then immediately SSH into the next machine

Now this works:

```
ssh bender
scp file.txt bender:
ssh bender
```

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>
```

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://mouapp.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

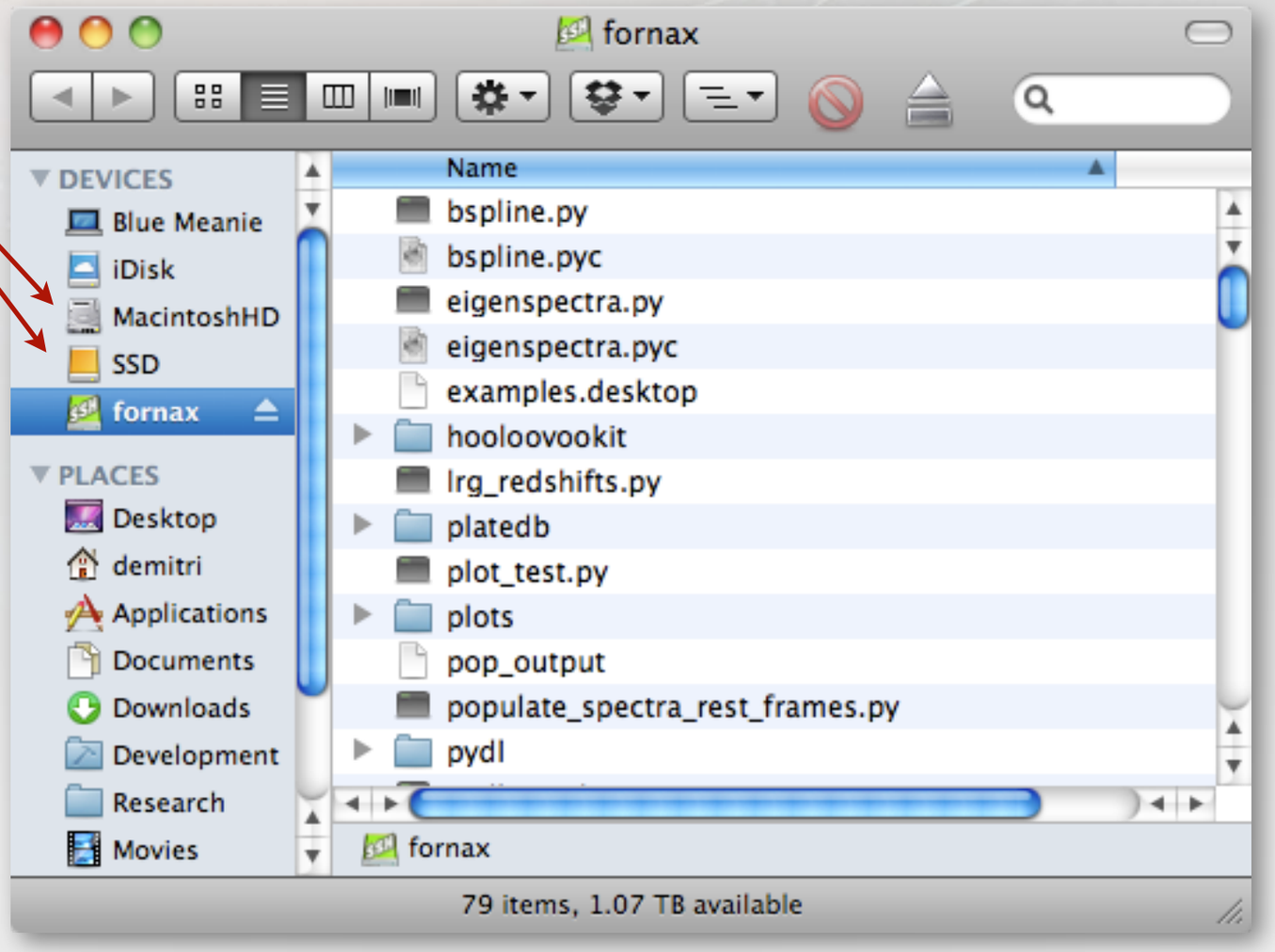
FUSE

We interact with files on remote machines on a daily basis. To read or edit these files, people typically ftp (or scp) files back and forth to your local machine.

A better way is to use FUSE. This allows you to mount any disk you can ssh to as if it were directly plugged into your machine.

Local disks

Remote Linux server



Files act as if they are on my computer (local programs can open them directly), but they are actually sitting on the remote server.

Installing FUSE – Mac

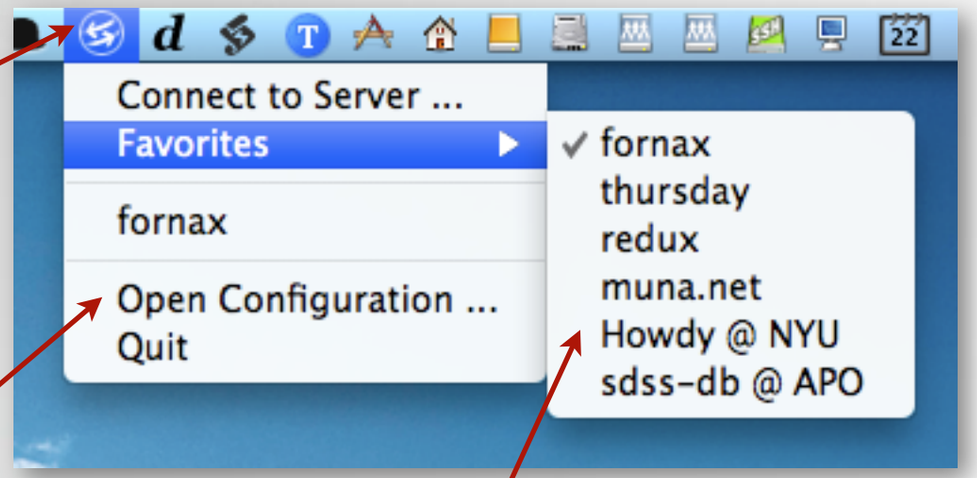
- First install OSXFuse:

  http://osxfuse.github.io

- Then install Macfusion, a graphical interface to mac-fuse:

  http://www.danielzhang.info/drupal6/system/files/Macfusion.app-built-by-Daniel-Zhang.zip

In Macfusion, "Start Macfusion Menuitem" to add shortcut menu to the menu bar.



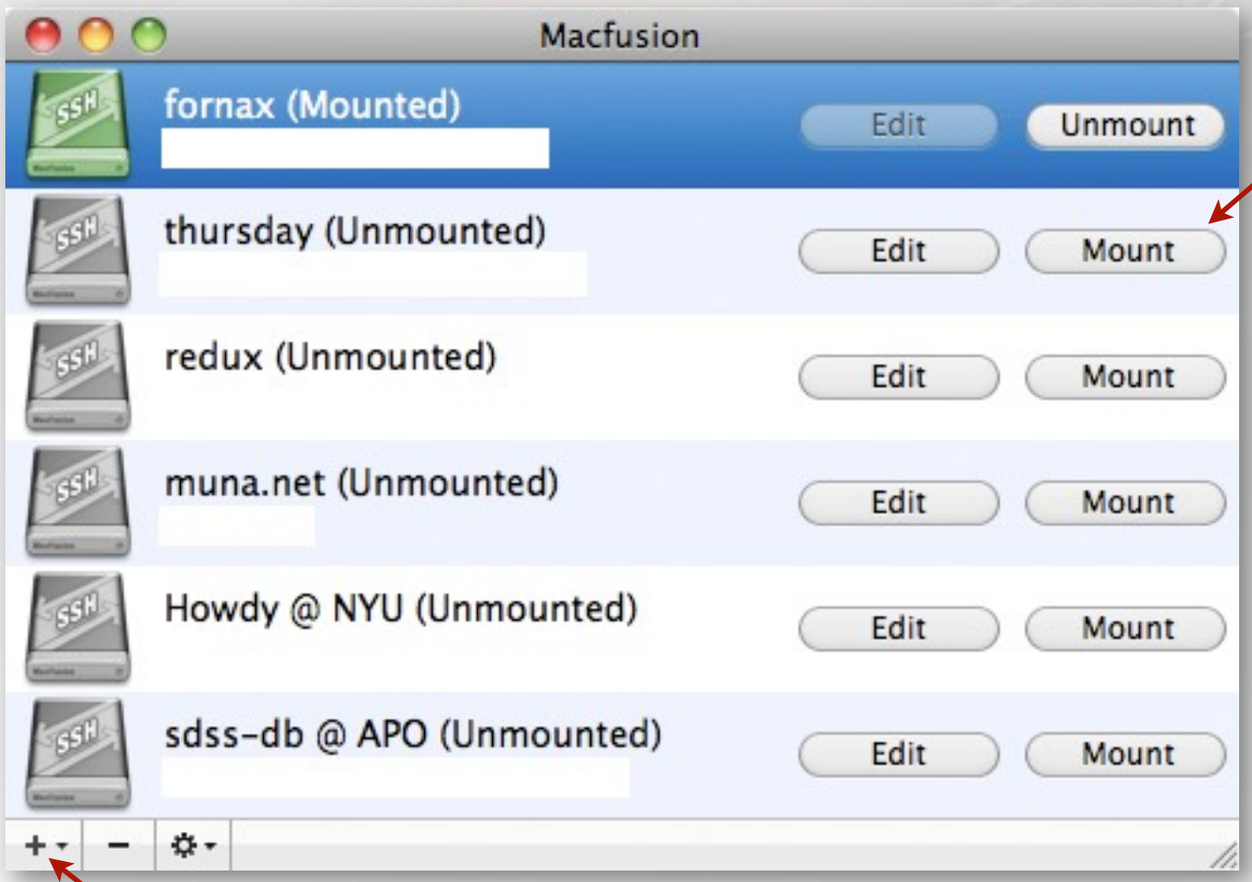
Create new connections.

Just select a pre-configured server to mount on desktop.

Macfusion

We interact with files on remote machines on a daily basis. To read or edit these files, people typically ftp (or scp) files back and forth to your local machine.

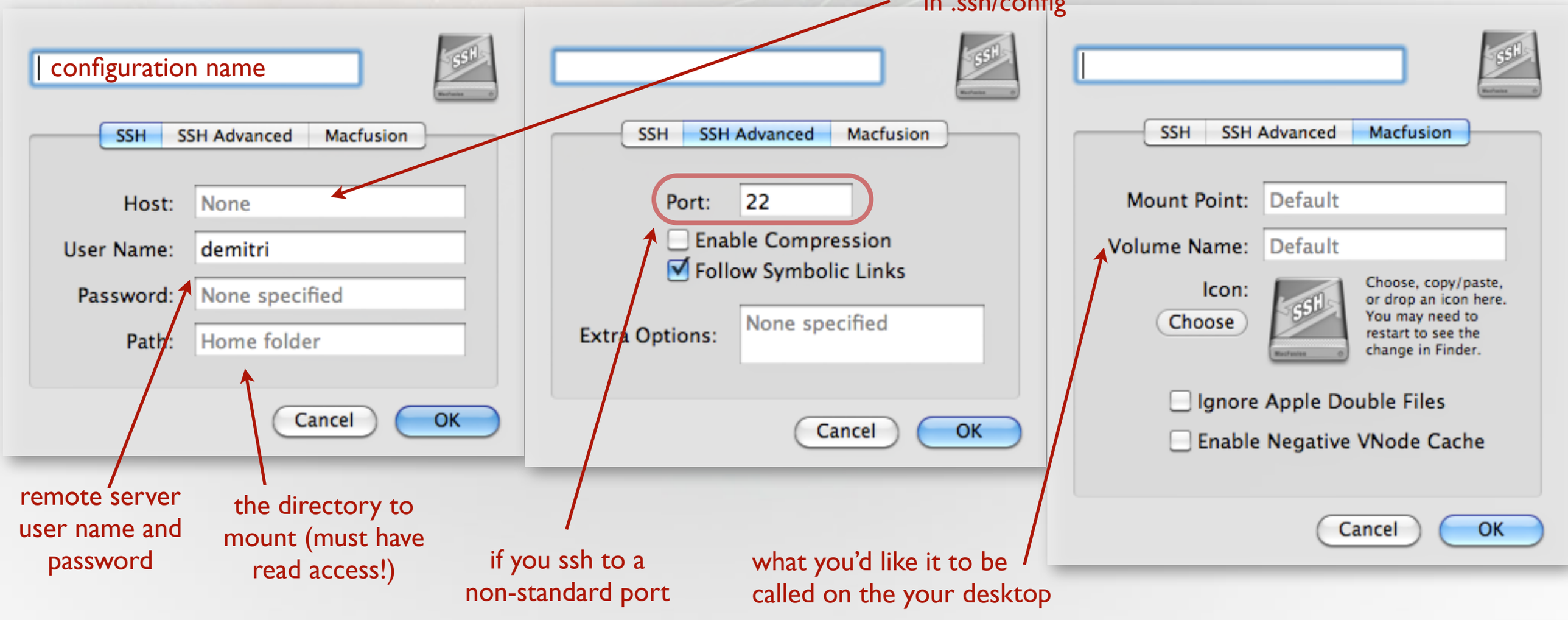
A better way is to use FUSE. This allows you to mount any disk you can ssh to as if it were directly plugged into your machine.



Macfusion

We interact with files on remote machines on a daily basis. To read or edit these files, people typically ftp (or scp) files back and forth to your local machine.

A better way is to use FUSE. This allows you to mount any disk you can ssh to as if it were directly plugged into your machine.



FUSE – Linux

Install via apt-get (on Ubuntu) or similar:

```
% sudo apt-get install sshfs
% sudo adduser <your username> fuse
```

gives your user permission to use fuse

Log out and then log in. Next, create a directory where we will mount the remote disks:

```
% sudo mkdir -p /mnt/sshfs
% sudo chown <your username>:fuse /mnt/sshfs
```

If your account name on the remote server 'astro.state.edu' is 'abc', you can mount your home directory there in your local home directory to a folder called "astro" there as:

```
% sudo sshfs abc@astro.state.edu: /mnt/sshfs/astro
```

your remote files now appear here as if they were local

To mount another directory, use:

```
% sudo sshfs abc@astro.state.edu:/some/other/dir /mnt/sshfs/astro
```

To unmount the "drive": `% sudo fusermount -u /mnt/sshfs/astro`

FUSE – Linux

You can create aliases to make this simpler for commonly used hosts:

```
# astro
alias mount_astro="sudo sshfs abc@astro.state.edu: /mnt/sshfs/astro"
alias unmount_astro="sudo fusermount -u /mnt/sshfs/astro"
```

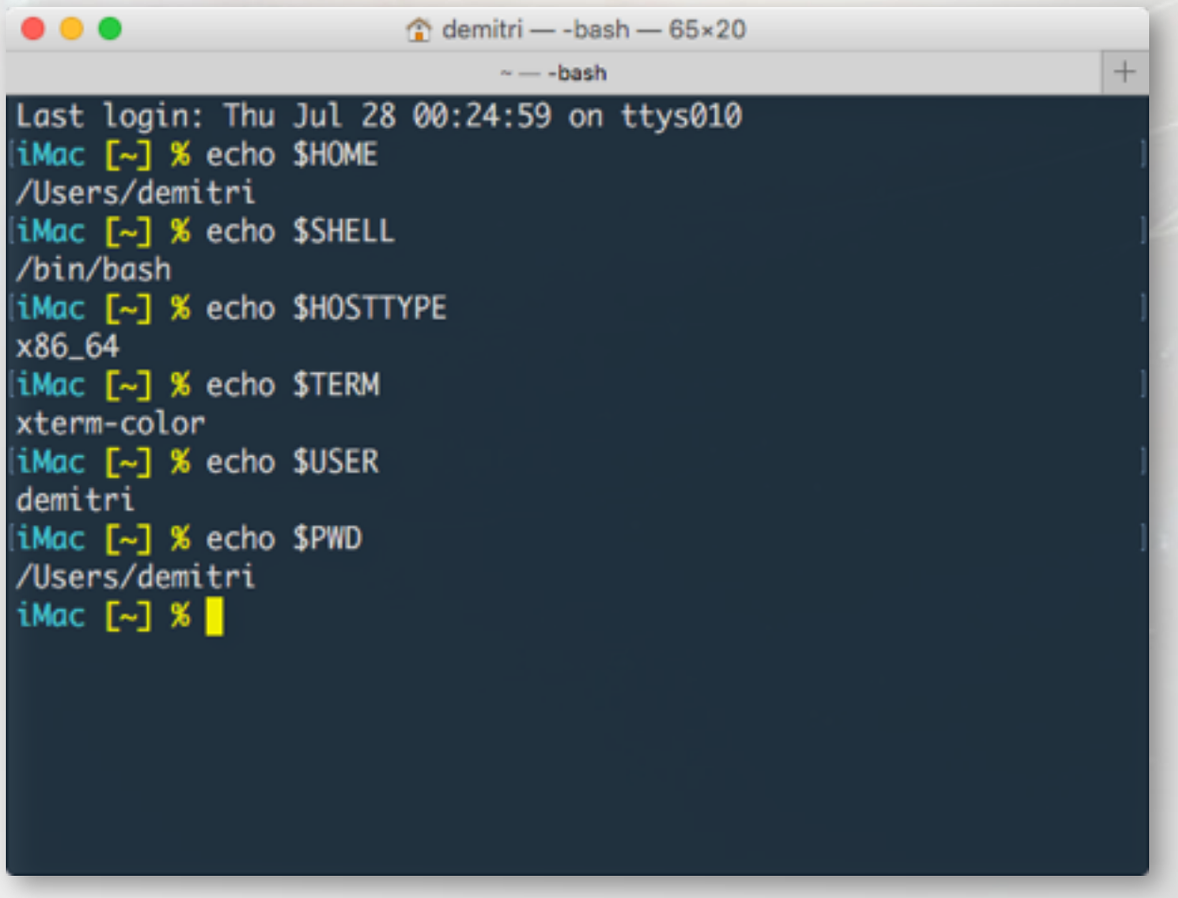
bring your towel

```
# traal
% alias mount_traal="sudo sshfs abc@traal.stat.edu: /mnt/sshfs/traal"
% alias unmount_traal="sudo fusermount -u /mny/sshfs/traal"
```

Environment Variables

The program that runs when you use the command line is called a "shell". Most people use bash (`/bin/bash`), many 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
```

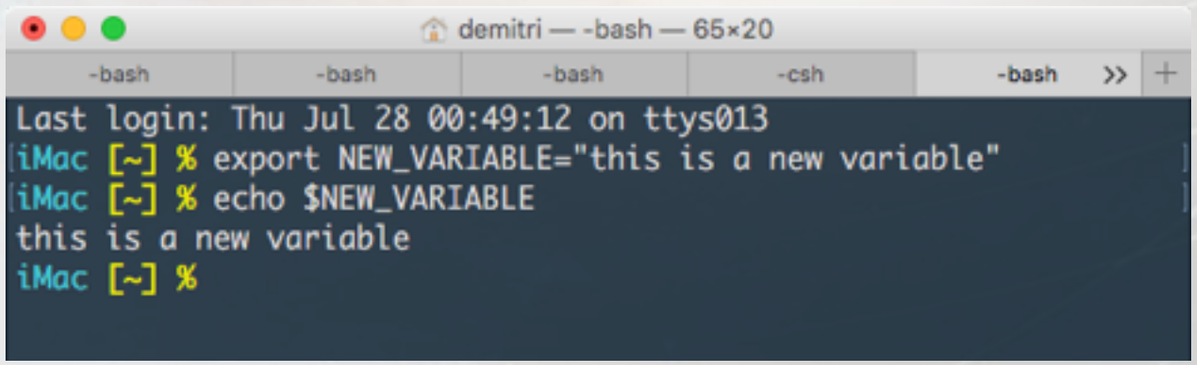
csh/tcsh

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

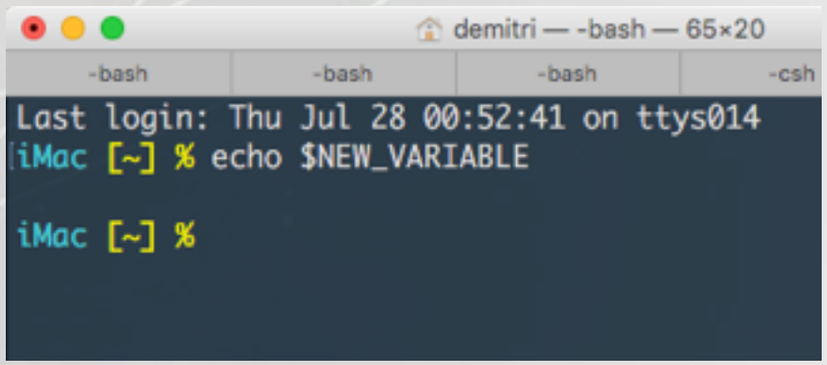
Environment Variable Scope

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

Define variable here...



Open new shell, variable not defined.

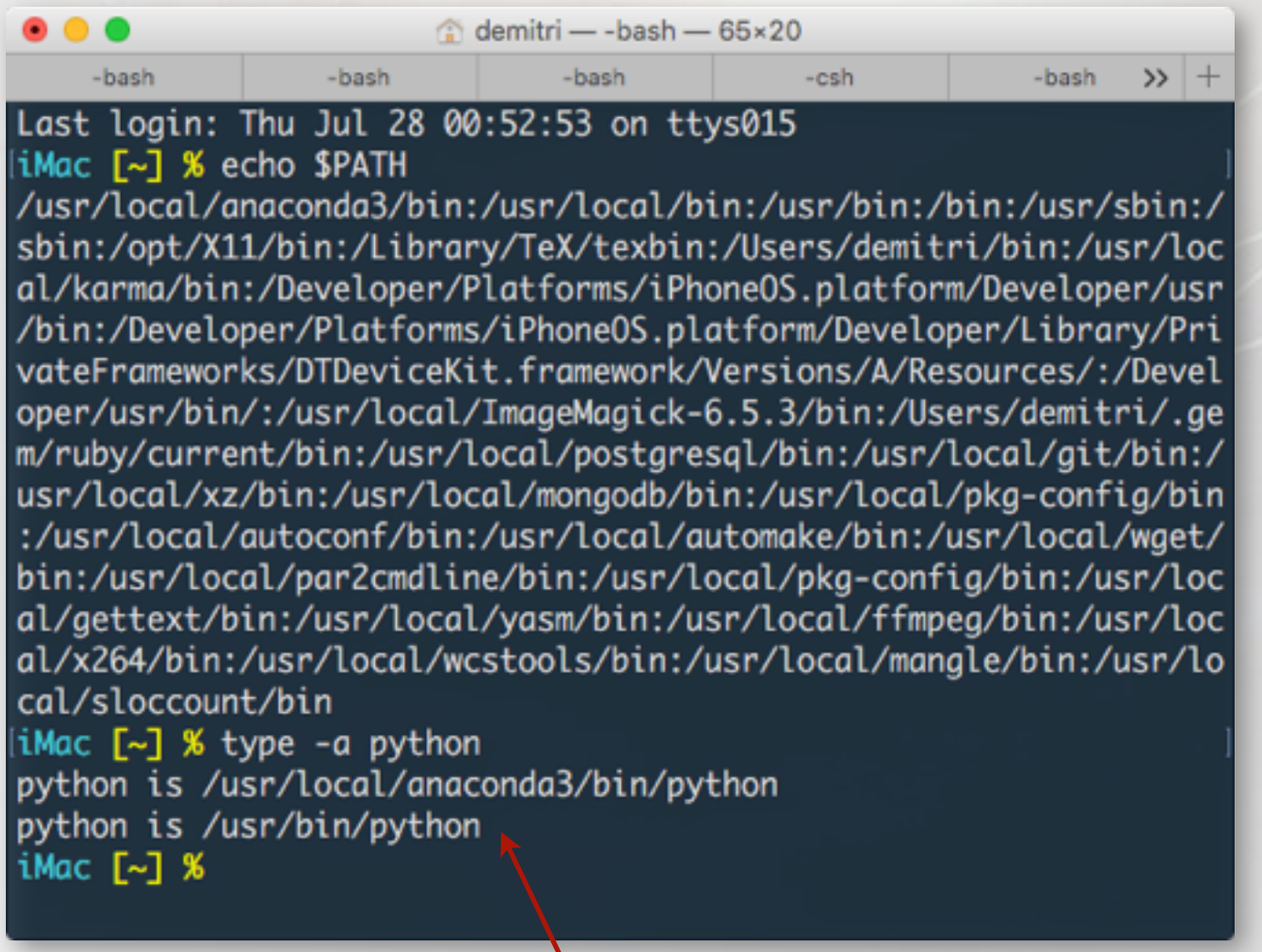


If you want certain variables to always be defined, create them in your shell startup files (e.g. `~/.bashrc`, `~/.cshrc`).

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?



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.

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.