
Once
I thought I was wrong,
but I was mistaken

Goals of the course

- To help you perform your research through instruction in the core components of data collection, organization, manipulation, analysis, interpretation and presentation.

- Provide a broad coverage of the core components of modern biological statistics

- Provide you with the computational tools necessary to carry out your work - namely R and affiliated tools

- This is a practical course and we will learn by doing

Why do we need statistics?

- We almost never know the world perfectly, but still want to draw conclusions or make decisions
- We need to estimate underlying parameters from samples of data
- Sometimes we need to test hypotheses using data
- Other times we need to more succinctly summarize and/or visualize large amounts of data
- There are well known mathematical rules that help us do both
- Statistics can be done by hand, but computers let us do most of the mathematics quickly

Why do we need statistics?

We want to turn data into conclusions about the world

- experimental design
- point estimates and confidence intervals
- hypothesis testing
- data reduction of highly dimensional data

<u>need</u>: a firm understanding of **probability**, **sampling** and **distributions**

How do we use statistics?

We need to work together.

Data, observations and variables

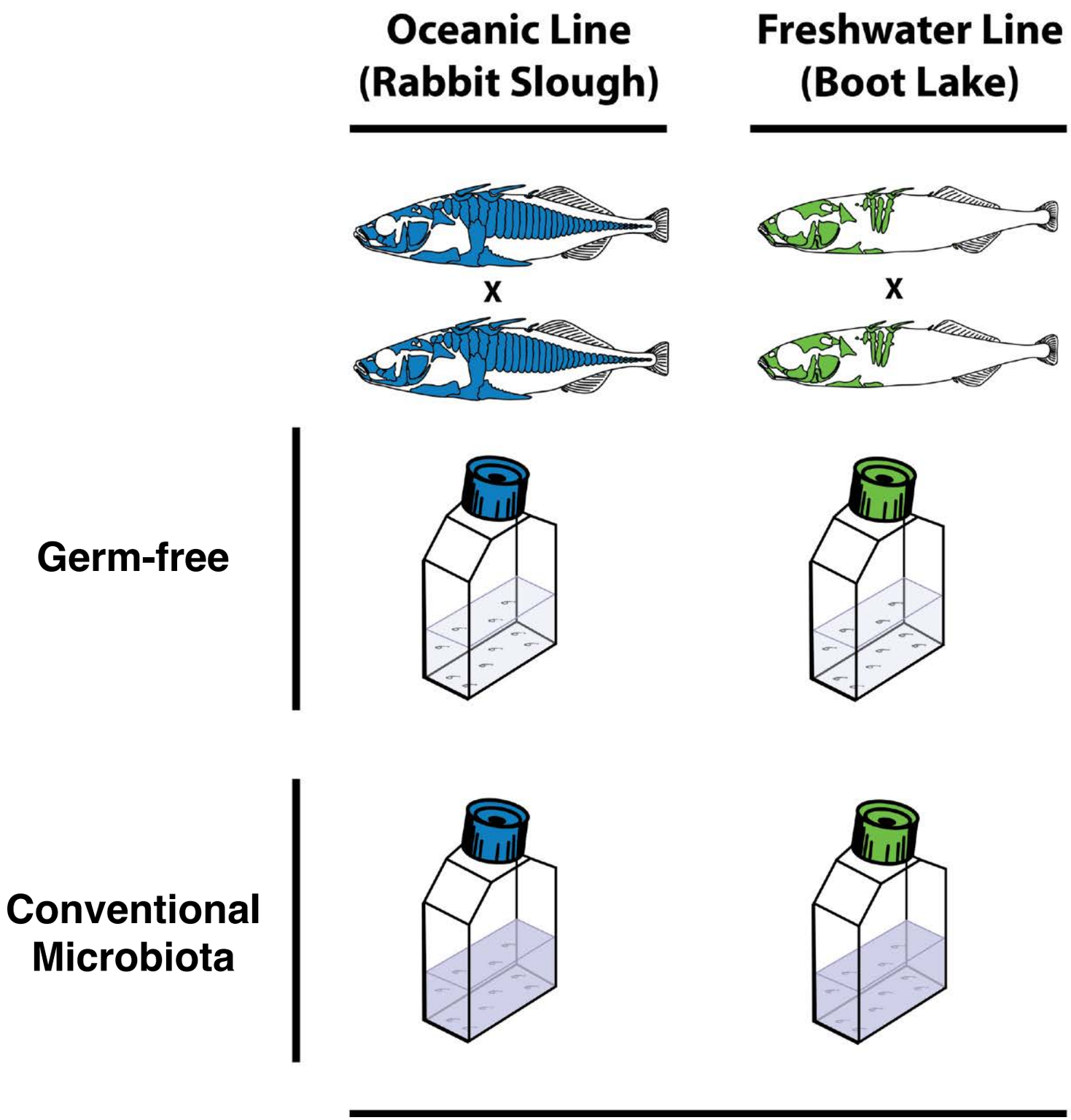
A biological example to get us started

- EXAMPLE - Say you perform an experiment on two different strains of stickleback fish, one from an ocean population (RS) and one from a freshwater lake (BP) by making them microbe free. Microbes in the gut are known to interact with the gut epithelium in ways that lead to a proper maturation of the immune system.

- EXPERIMENTAL SETUP - You decide to carry out an experiment by treating multiple fish from each strain so that some of them have a conventional microbiota, and some of them are inoculated with only one bacterial species. You then measure the levels of gene expression in the stickleback gut using RNA-seq. Because you have a suspicion that the sex of the fish might be important, you track it too.

- GETTING THE DATA READY TO ANALYZE - How should the data set be organized to best analyze it? What are the properties of the variables, and why does that matter?

A biological example to get us started



Term	Definition	Example
Measurement	A single piece of recorded information reflecting a characteristic of interest (e.g. length of a leaf, pH of a water aliquot mass of an individual, number of individuals per quadrat etc)	Protein content of the milk of a single female koala
Observation	A single measured sampling or experimental unit (such as an individual, a quadrat, a site etc)	A small quantity of milk from a single koala
Population	All the possible observations that could be measured and the unit of which wish to draw conclusions about (note a statistical population need not be a viable biological population)	The milk of all female koalas
Sample	The (representative) subset of the population that are observed	A small quantity of milk collected from 15 captive female koalas[a]
Variable	A set of measurements of the same type that comprise the sample. The characteristic that differs (varies) from observation to observation	The protein content of koala milk.

Data set rules of thumb (aka Tidy Data)

- Store a copy of data in **nonproprietary** software and hardware formats, such as plain ASCII text (aka a flat file)
- Leave an **uncorrected file** when doing analyses
- Use **descriptive** names for your data files and variables
- Include a **header line** with descriptive variable names
- Maintain effective **metadata** about the data
- When you add **observations** to a database, add rows
- When you add **variables** to a database, add columns, not rows
- A **column** of data should contain only one data type

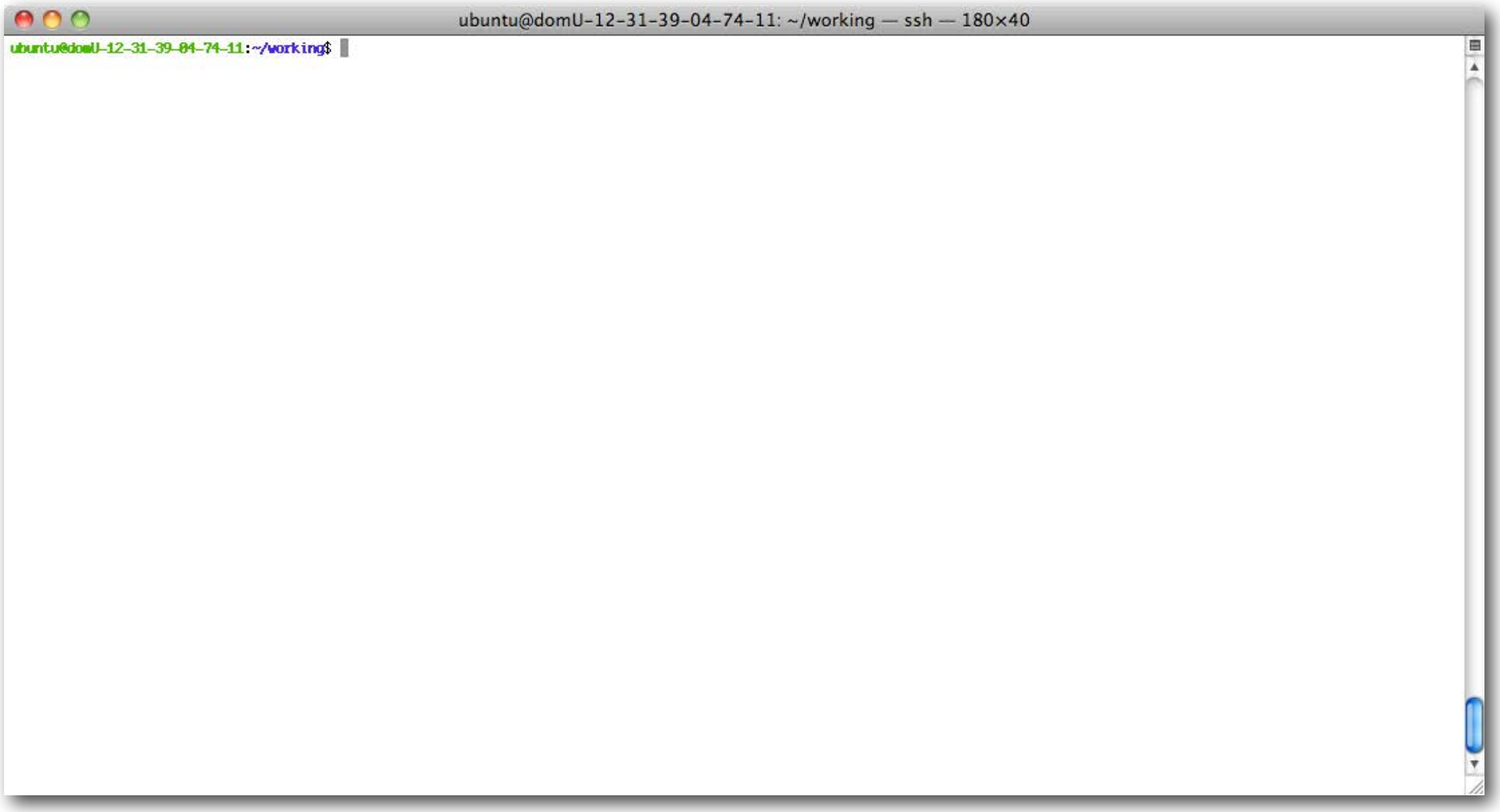
Repeatable science rules of thumb (aka Tidy Data)

- Use a scripting program like **R** for analysis, and
- Make your data and analyses freely available <u>and</u> understandable
- Use **R Markdown** for documentation and dissemination
- Use **Git** and **GitHub** to collaborate and distribute your files
- Know the basics of **command line tools** on your computer

Linux

Unix and Unix-like environments:

Convenient handling and manipulation of data files



- Apple OS X Macs
- Linux workstations and servers
- Virtual Machines
- Google's Android phones

The Terminal Window

ubuntu@ip-10-4-230-31: ~/working — ssh — 141×44

```
ubuntu@ip-10-4-230-31:~/working$ ls -la ~/
total 444
drwxr-xr-x 18 ubuntu ubuntu   4096 2012-01-09 22:50 .
drwxr-xr-x  6 root   root     4096 2011-11-14 17:12 ..
-rw-------  1 ubuntu ubuntu   3757 2012-03-12 12:11 .bash_history
-rw-r--r--  1 ubuntu ubuntu    220 2011-05-18 10:00 .bash_logout
-rw-r--r--  1 ubuntu ubuntu   3581 2011-11-14 17:16 .bashrc
lrwxrwxrwx  1 root   root       21 2011-11-14 12:25 bin -> ../../usr/proftpd/bin
drwxrwxr-x  2 ubuntu ubuntu   4096 2011-11-14 17:16 .byobu
drwxrwxr-x  4 ubuntu ubuntu   4096 2011-11-14 14:33 .cabal
drwx------  3 ubuntu ubuntu   4096 2011-11-14 11:37 .cache
lrwxrwxrwx  1 root   root       20 2011-11-14 12:23 conf -> ../../usr/nginx/conf
-rwxrwxrwx  1 ubuntu ubuntu    992 2011-11-14 17:12 configure_freenx.sh
drwx------  3 ubuntu ubuntu   4096 2012-01-08 22:52 .emacs.d
lrwxrwxrwx  1 root   root       21 2011-11-14 12:25 etc -> ../../usr/proftpd/etc
drwxr-xr-x  2 ubuntu ubuntu   4096 2011-11-14 12:51 .fontconfig
drwx------  2 ubuntu ubuntu   4096 2011-11-14 14:18 .gconf
drwxr-xr-x  3 root   root     4096 2011-11-14 16:58 .gem
drwx------  2 ubuntu ubuntu   4096 2011-11-14 14:51 .gnupg
lrwxrwxrwx  1 root   root       20 2011-11-14 12:23 html -> ../../usr/nginx/html
lrwxrwxrwx  1 root   root       25 2011-11-14 12:25 include -> ../../usr/proftpd/include
drwxrwxr-x  4 ubuntu ubuntu   4096 2011-11-28 17:49 install
drwxrwxr-x  3 ubuntu ubuntu   4096 2011-11-14 12:27 .lein
-rw-------  1 ubuntu ubuntu     65 2011-11-14 13:07 .lesshst
lrwxrwxrwx  1 root   root       21 2011-11-14 12:25 lib -> ../../usr/proftpd/lib
lrwxrwxrwx  1 root   root       25 2011-11-14 12:25 libexec -> ../../usr/proftpd/libexec
lrwxrwxrwx  1 root   root       20 2011-11-14 12:23 logs -> ../../usr/nginx/logs
drwxrwxr-x  2 ubuntu ubuntu   4096 2011-11-14 12:51 .m2
drwxrwxr-x  2 ubuntu ubuntu   4096 2011-11-14 14:18 .matplotlib
-rw-------  1 ubuntu ubuntu   2964 2012-01-09 22:50 .mysql_history
-rw-r--r--  1 ubuntu ubuntu    675 2011-11-14 17:16 .profile
drwxr-xr-x  2 root   root     4096 2011-11-14 12:25 sbin
-rw-rw-r--  1 ubuntu ubuntu      0 2011-11-14 11:37 .screenrc
lrwxrwxrwx  1 root   root       23 2011-11-14 12:25 share -> ../../usr/proftpd/share
drwx------  2 ubuntu ubuntu   4096 2011-11-14 11:33 .ssh
-rw-rw-r--  1 ubuntu ubuntu 338416 2012-01-09 16:12 stacks-0.998.tar.gz
drwxrwxr-x  3 ubuntu ubuntu   4096 2011-11-14 13:44 .subversion
-rw-r--r--  1 ubuntu ubuntu      0 2011-11-14 11:38 .sudo_as_admin_successful
drwxrwxr-x  2 ubuntu ubuntu   4096 2012-01-09 04:42 tmp
lrwxrwxrwx  1 root   root       21 2011-11-14 12:25 var -> ../../usr/proftpd/var
-rw-------  1 ubuntu ubuntu   7522 2012-01-09 20:35 .viminfo
lrwxrwxrwx  1 ubuntu ubuntu     12 2012-01-08 22:49 working -> /mnt/working
-rw-------  1 ubuntu ubuntu    196 2011-12-12 12:04 .Xauthority
ubuntu@ip-10-4-230-31:~/working$
```

Obtain a cheat sheet

google "unix commands"

Basic UNIX commands

Basic UNIX commands

mally.stanford.edu/~sr/computing/basic-unix.html

Google

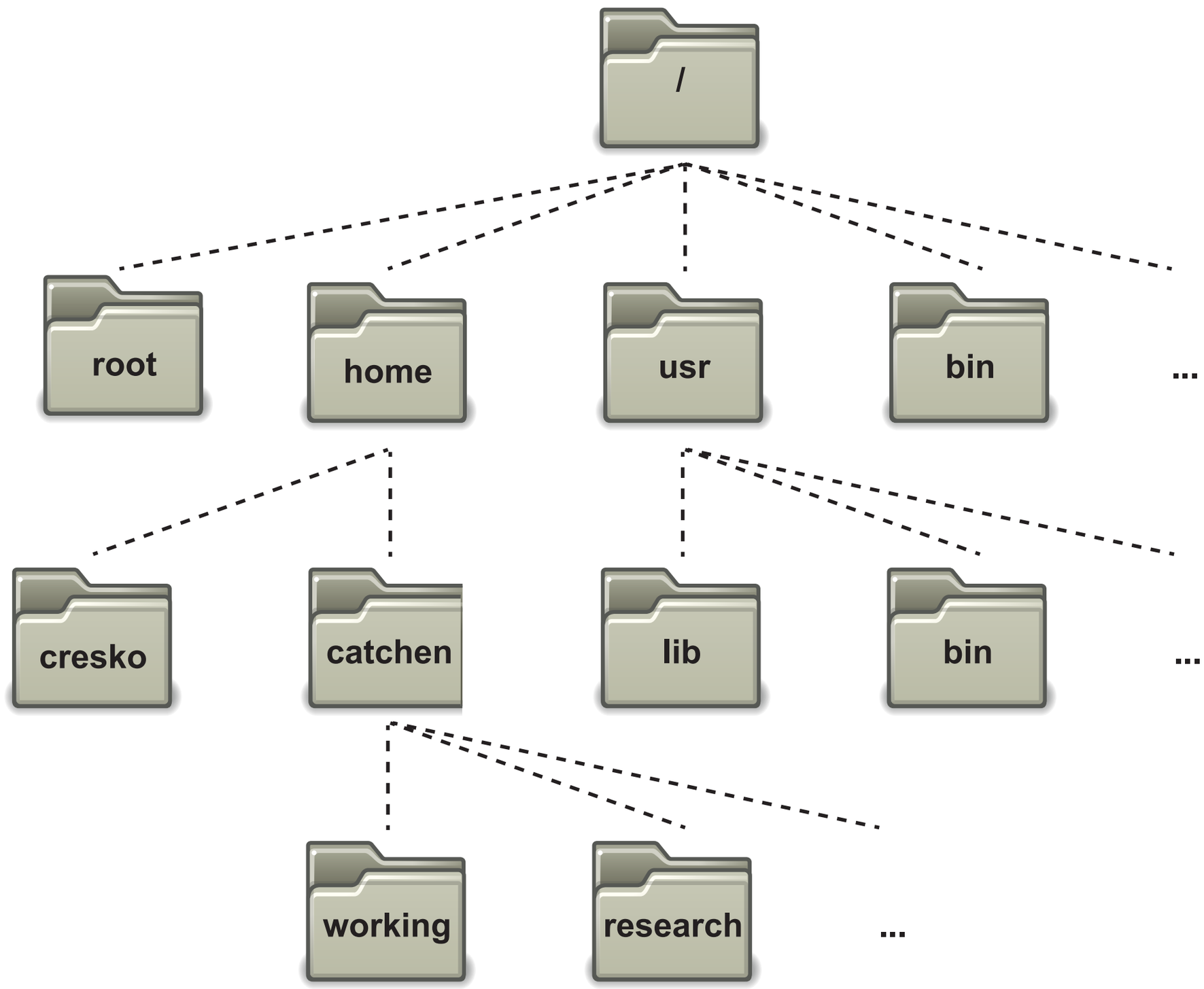
Basic UNIX commands

Note: not all of these are actually part of UNIX itself, and you may not find them on all UNIX machines. But they can all be used on **turing** in essentially the same way, by typing the command and hitting return. Note that some of these commands are different on non-Solaris machines - see SunOS differences.
If you've made a typo, the easiest thing to do is hit **CTRL-u** to cancel the whole line. But you can also edit the command line (see the guide to More UNIX).
UNIX is case-sensitive.

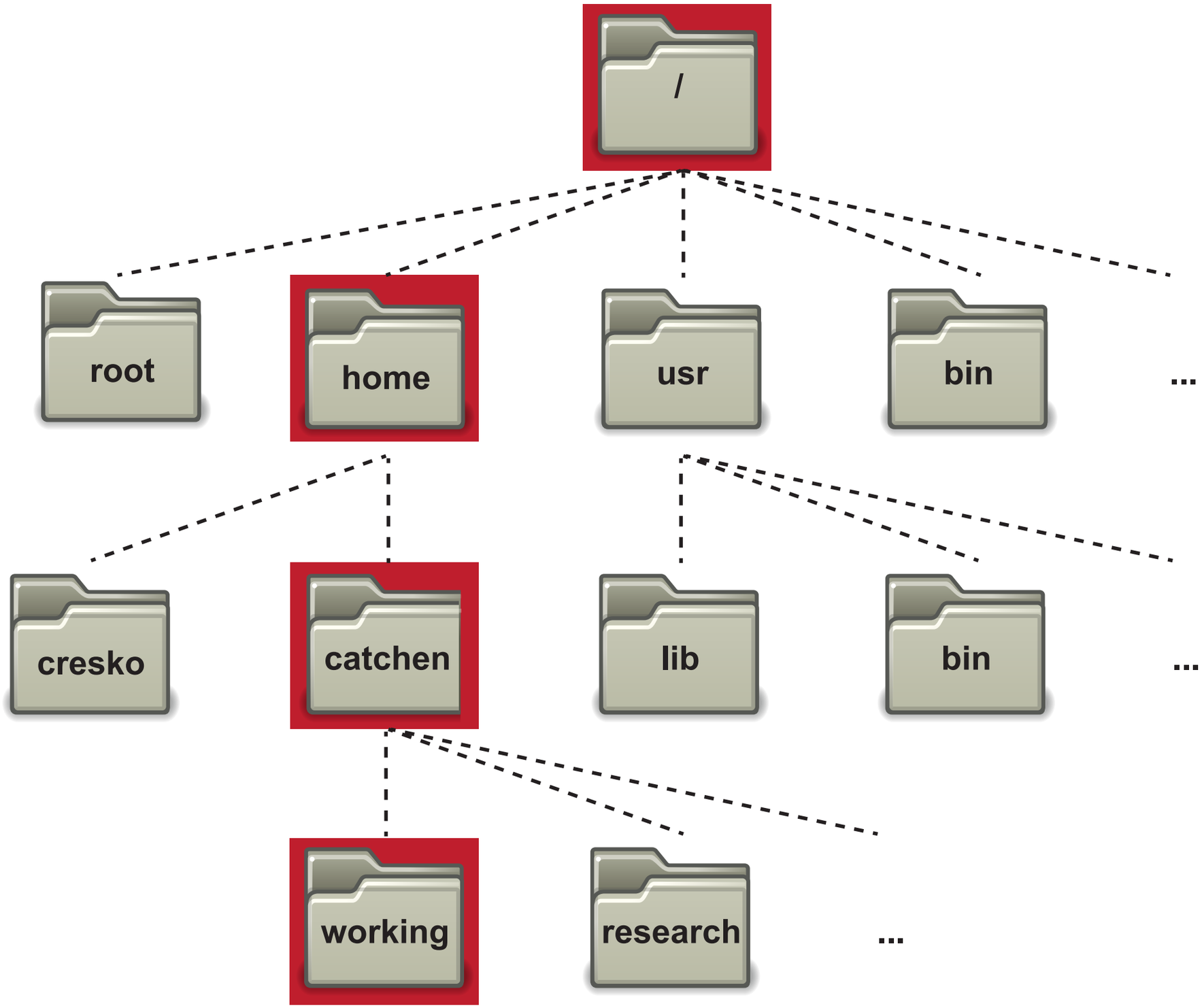
Files

- **ls** --- lists your files
  ls -l --- lists your files in 'long format', which contains lots of useful information, e.g. the exact size of the file, who owns the file and who has the right to look at it, and when it was last modified.
  ls -a --- lists all files, including the ones whose filenames begin in a dot, which you do not always want to see.
  There are many more options, for example to list files by size, by date, recursively etc.
- **more *filename*** --- shows the first part of a file, just as much as will fit on one screen. Just hit the space bar to see more or **q** to quit. You can use ***/pattern*** to search for a pattern.
- **emacs *filename*** --- is an editor that lets you create and edit a file. See the emacs page.
- **mv *filename1 filename2*** --- moves a file (i.e. gives it a different name, or moves it into a different directory (see below)
- **cp *filename1 filename2*** --- copies a file
- **rm *filename*** --- removes a file. It is wise to use the option rm -i, which will ask you for confirmation before actually deleting anything. You can make this your default by making an alias in your .cshrc file.
- **diff *filename1 filename2*** --- compares files, and shows where they differ
- **wc *filename*** --- tells you how many lines, words, and characters there are in a file
- **chmod *options filename*** --- lets you change the read, write, and execute permissions on your files. The default is that only you can look at them and change them, but you may sometimes want to change these permissions. For example, **chmod o+r *filename*** will make the file readable for everyone, and **chmod o-r *filename*** will make it unreadable for others again. Note that for someone to be able to actually look at the file the directories it is in need to be at least executable. See help protection for more details.
- File Compression
  - **gzip *filename*** --- compresses files, so that they take up much less space. Usually text files compress to about half their original size, but it depends very much on the size of the file and the nature of the contents. There are other tools for this purpose, too (e.g. **compress**), but gzip usually gives the highest compression rate. Gzip produces files with the ending '.gz' appended to the original filename.
  - **gunzip *filename*** --- uncompresses files compressed by gzip

In UNIX everything is a file organized in a hierarchy



Paths



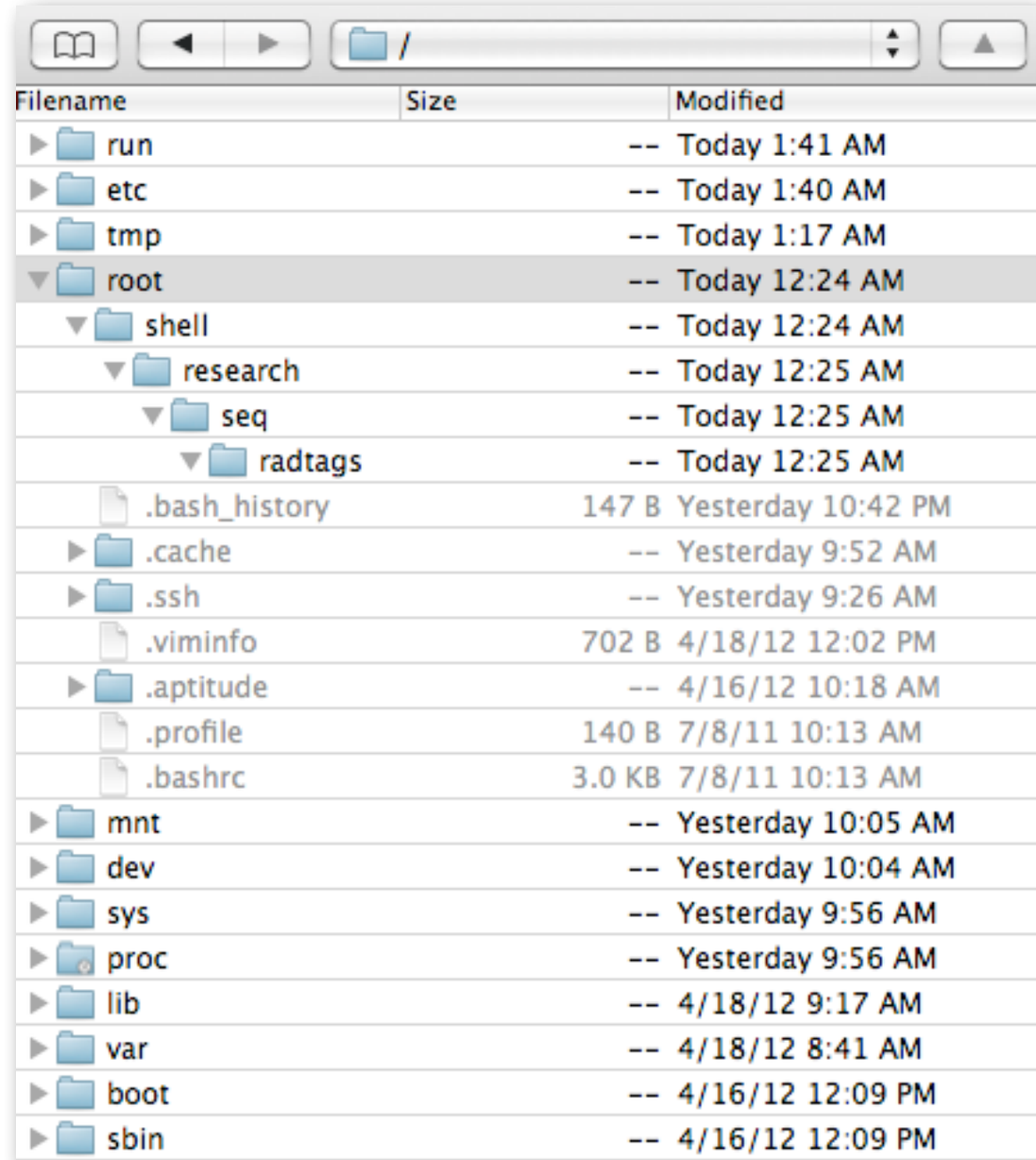
/home/catchen/working

Paths, cont

This shell view of the nested directories shell, research, seq, and radtags…..

```
root@ubuntu:~# mkdir shell
root@ubuntu:~# cd shell
root@ubuntu:~/shell# mkdir research
root@ubuntu:~/shell# ls
research
root@ubuntu:~/shell# cd research
root@ubuntu:~/shell/research# mkdir seq
root@ubuntu:~/shell/research# ls
seq
root@ubuntu:~/shell/research# cd seq
root@ubuntu:~/shell/research/seq# mkdir radtags
root@ubuntu:~/shell/research/seq# cd radtags/
root@ubuntu:~/shell/research/seq/radtags# ls
root@ubuntu:~/shell/research/seq/radtags# ls -la
total 8
drwxr-xr-x 2 root root 4096 2012-06-25 00:25 .
drwxr-xr-x 3 root root 4096 2012-06-25 00:25 ..
root@ubuntu:~/shell/research/seq/radtags# pwd
/root/shell/research/seq/radtags
root@ubuntu:~/shell/research/seq/radtags#
```

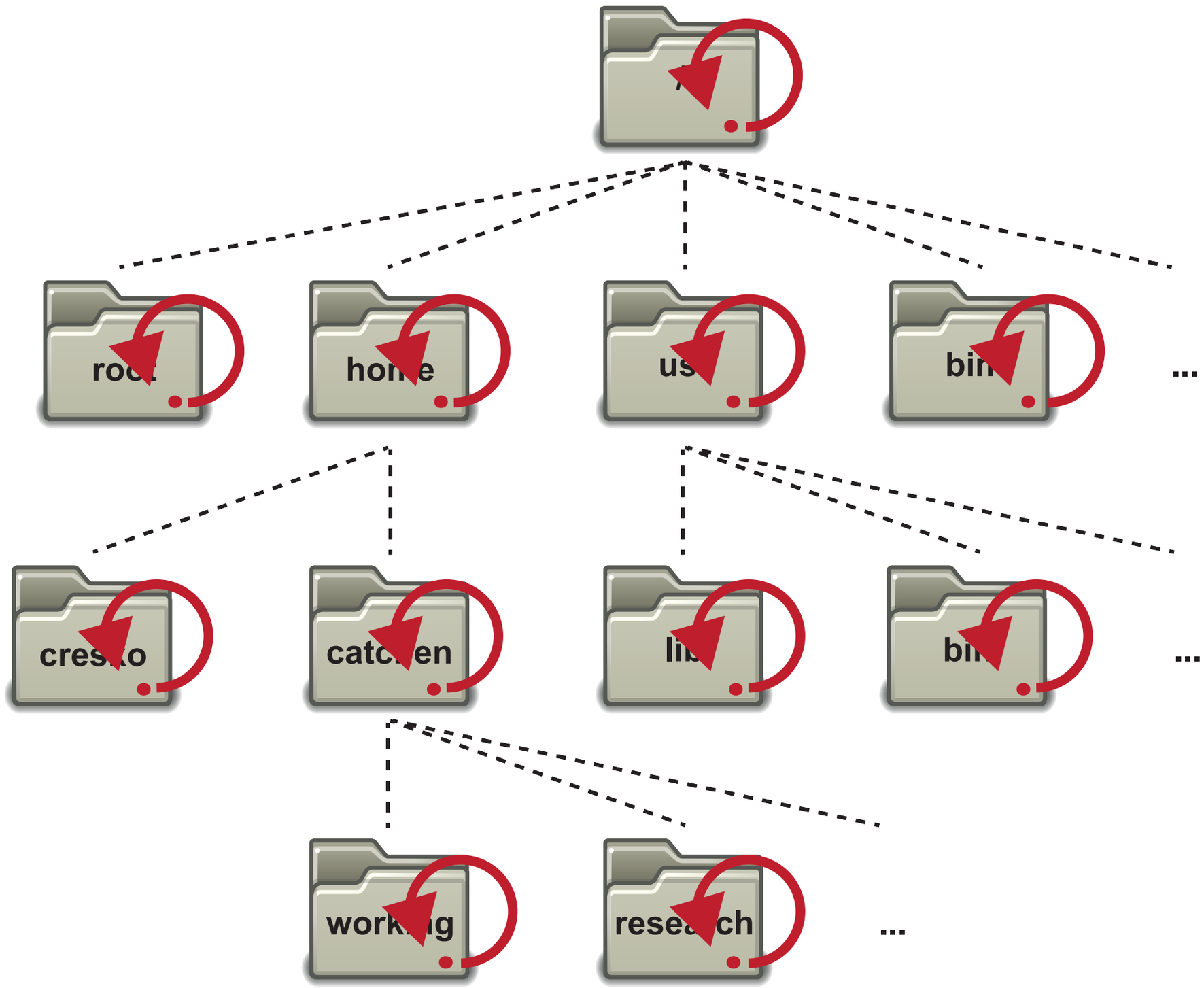
…. is equivalent to this GUI view of the same directories



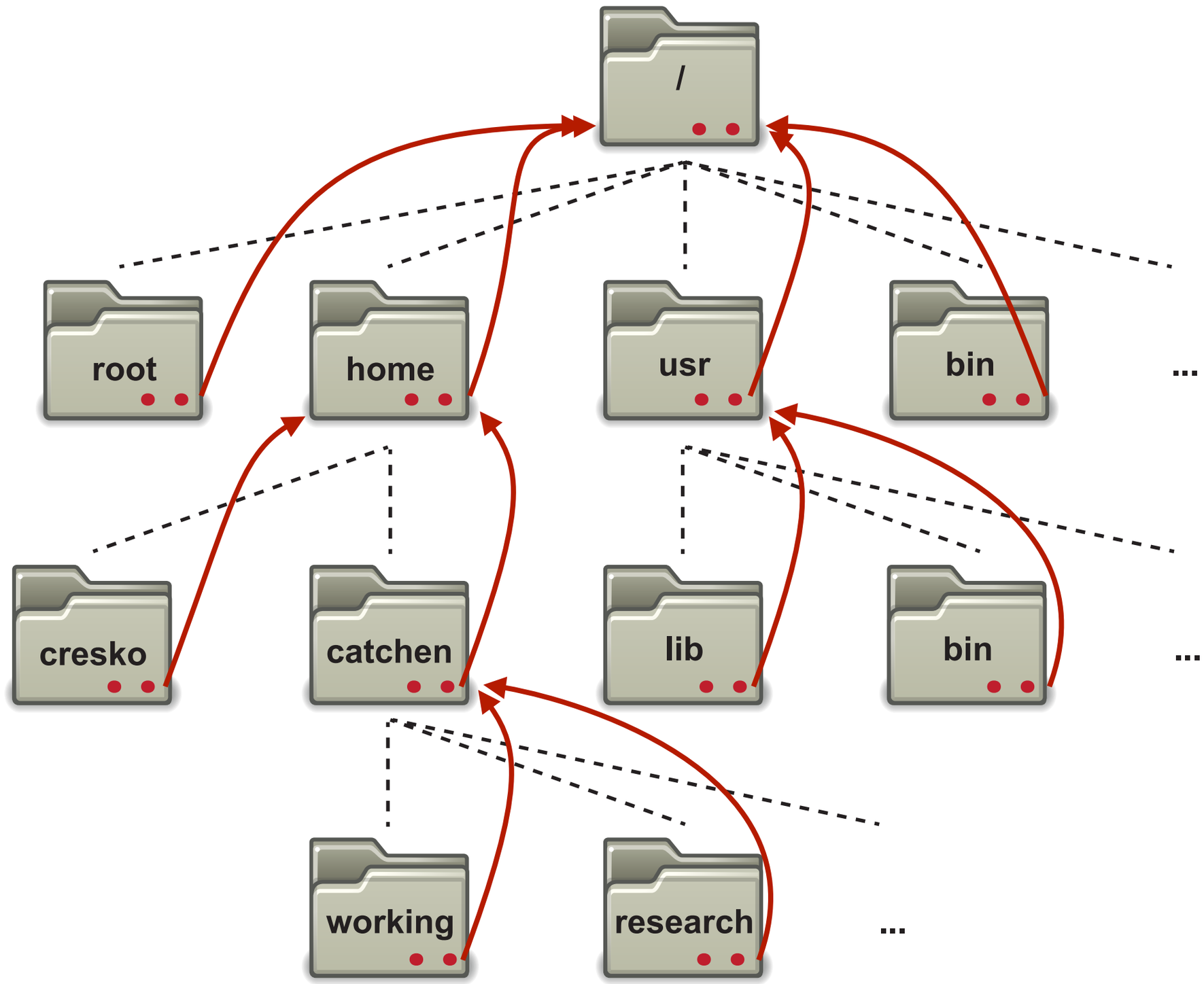
And the `radtags` directory is uniquely identified by its path:

`/root/shell/research/seq/radtags`

Special files -- ‘*dot*’



Special files -- '*dot dot*'

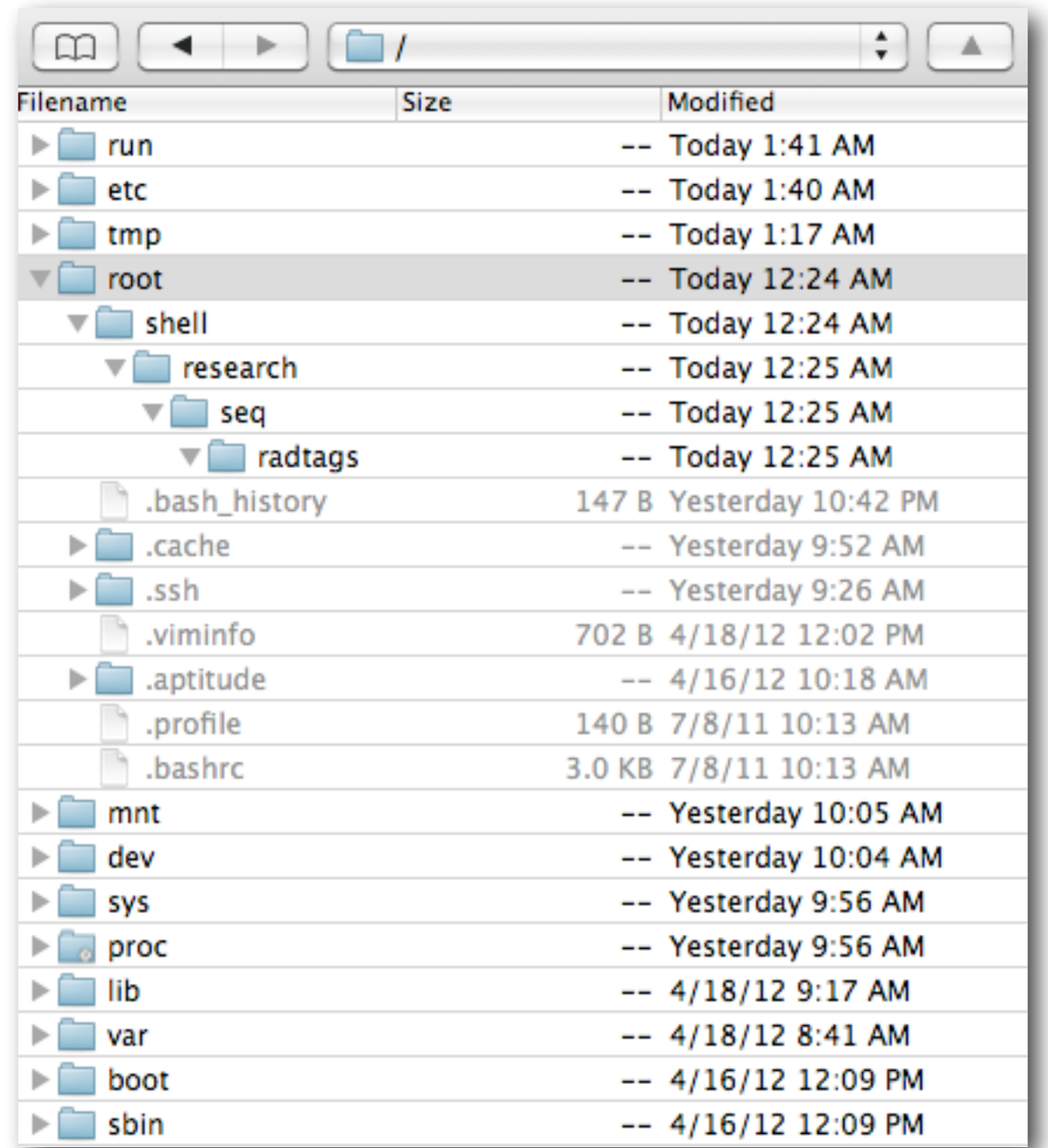


Relative and absolute paths

`/root/shell/research/seq/radtags`

Create a series of directories under shell:

```
root@ubuntu:~# mkdir shell
root@ubuntu:~# cd shell
root@ubuntu:~/shell# mkdir research
root@ubuntu:~/shell# ls
research
root@ubuntu:~/shell# cd research
root@ubuntu:~/shell/research# mkdir seq
root@ubuntu:~/shell/research# ls
seq
root@ubuntu:~/shell/research# cd seq
root@ubuntu:~/shell/research/seq# mkdir radtags
root@ubuntu:~/shell/research/seq# cd radtags/
root@ubuntu:~/shell/research/seq/radtags# ls
root@ubuntu:~/shell/research/seq/radtags# ls -la
total 8
drwxr-xr-x 2 root root 4096 2012-06-25 00:25 .
drwxr-xr-x 3 root root 4096 2012-06-25 00:25 ..
root@ubuntu:~/shell/research/seq/radtags# pwd
/root/shell/research/seq/radtags
root@ubuntu:~/shell/research/seq/radtags#
```




```
% ls .                    % ls ..                    % ls ../../

% cd ~/                   % cd shell/research        % pwd
```

Why use the command line?

- The commands work almost identically across platforms
- You can even use them on a large computer cluster like Talapas
- It is incredibly powerful particularly for repeated actions
- It allows you to do thousands of ‘clicks’ with single commands

Getting used to the power of the command line

- Get a **terminal** application up and running on your computer
- Make sure that you can display the **path** to your current directory
- **List** the files in the current directory
- Use **arguments** to your list command to show **all** files in **long** form
- Move through your directories by **changing** your location them
- **Make** a new directory and put an existing text file in it
- Use the command '**less**' to see the contents of the file
- Create a new file in the directory using '**nano**'
- **Copy** the files that you made into a new directory
- **Delete** the files and then the directory

Beginning R Tutorial

Why use R?

- Good general scripting tool for statistics and mathematics
- Powerful and flexible and free
- Runs on all computer platforms
- New enhancements coming out all the time
- Superb data management & graphics capabilities
- Reproducibility - can keep your scripts to see exactly what was done
- Can embed your R analyses in dynamic, polished files using R markdown
- You can write your own functions
- Lots of online help available
- Can use a nice GUI front end such as Rstudio

AN INTRODUCTION TO GETTING STARTED WITH THE BASICS IN R

Anything that follows '#' symbol (aka hash) is just for humans

You can add notes to yourself in your scripts
A little later we will run a better way to embed code in R Markdown scripts
You can run commands through the terminal, console or **scripts**

Let's start with a simple command that shows the
immediate results but the data are not stored.

```
4*4
```

```
(4+3)*2^2
```

A better way to do this is to assign variables
Variables are assigned values using the "<-" operator.
Variable names must begin with a letter, but other than that, just
About anything goes. Do keep in mind that R is case sensitive.

```
x <- 2
x*3
y <- x * 3
y-2
```

These are no good

```
3y <- 3
3*y <- 3
```

Arithmetic operations can be performed easily on functions
as well as numbers.
Try the following, and then your own.

Note that the last of these - 'log' - is a built in function
of R, and therefore the object of the function needs
to be put in parentheses

These parentheses will be important, and we'll come back to
them later when we add arguments after the object in the
parentheses

```
x+2
x^2
log(x)
```

the outcome of calculations can be assigned to new variables as
well, and the results can be checked using the 'print' command

```
y <- 67
 print(y)
```

```
x <- 124
z <- (x*y)^2
 print(z)
```

STRINGS

Variables and operations can be performed on characters as well.
Note that characters need to be set off by quotation marks to
differentiate them from numbers.

The c stands for 'concatenate'.
note that we are using the same variable names as we did
previously, which means that we're overwriting our previous
assignment.

A good rule of thumb is to use new names for each variable, and
make them short but still descriptive

```
x <- “I Love"
 print (x)
y <- “Biostatistics”
 print (y)
z <- c(x,y)
 print (z)
```

The variable z is now what is called a list of character values.

Sometimes we would like to treat the characters as if they were
units for subsequent calculations.

These are called factors, and we can redefine our character
variables as factors.

This might seem a bit strange, but it’s important for statistical
analyses where we might want to see the mean or variance for two
different treatments.

```
z_factor <- as.factor(z)
print (z_factor)
```

Note that factor levels are reported alphabetically

```
# In general R thinks in terms of vectors (a list of characters,
# factors or numerical values) and it will benefit any R user to
# try to write programs with that in mind, as it will simplify most
# things.

# Vectors can be assigned directly using the 'c()' function and
then entering the exact values.
```

```r
x <- c(2,3,4,2,1,2,4,5,10,8,9)
 print(x)
```

BASIC STATISTICS OF DATA

```
# Many functions exist to operate on vectors.
# Combine these with your previous variable to see what happens.
# Also, try to find other functions (e.g. standard deviation).


mean(x)
median(x)
var(x)
sum(x)
length(x)
sample(x, replace = T)
```

```
# Getting Help on any function is very easy - just type a question
# mark and the name of the function.

# There are functions for just about anything within R and it is
# easy enough to write your own functions if none already exist to
# do what you want to do.

# In general, function calls have a simple structure: a function
# name, a set of parentheses and an optional set of parameters to
# send to the function.

# Help pages exist for all functions that, at a minimum, explain
# what parameters exist for the function.
# Help can be accessed a few ways - try them :
```

```
help(mean)
?mean
example(mean)
demo(mean)

#Note, not all functions have demos

help.search("mean")
apropos(“mea”)
args(mean)
```

```r
# Creating vector of new data by entering it by hand can be a
# drag. However, it is also very easy to use functions such as
# 'seq' and 'sample'.
# Try the examples below Can you figure out what the three
# arguments in the parentheses mean?
# Try varying the arguments to see what happens. Don't go too
# crazy with the last one or your computer might slow way down.

seq_1 <- seq(0.0, 10.0, by = 0.1)
 print(seq_1)
seq_2 <- seq(10.0, 0.0, by = -0.1)
 print(seq_2)
seq_square <- (seq_2)*(seq_2)
 print(seq_square)
seq_square_new <- (seq_2)^2
 print(seq_square_new)
```

```
# Here is a way to create your own data sets that are random
# samples.
# Again, play around with the arguments in the parentheses to see
# what happens.


x <- rnorm (10000, 0, 10)
y <- sample (1:10000, 10000, replace = T)
xy <- cbind(x,y)

plot(x,y)
plot(xy)
hist(x)
```

```
# You've probably figured out that y from the last example is
# drawing numbers with equal probability.
# What if you want to draw from a distribution?
# Again, play around with the arguments in the parentheses to see
what happens.


x <-rnorm(1000, 0, 100)
print (x)
hist(x, xlim = c(-50,50))
hist(x, xlim = c(-500,500))
curve(5000*dnorm(x, 0, 100), xlim = c(-50,50), add=TRUE)

# dnorm() generates the probability density, which can be plotted
# using the curve() function.

# Note that is curve is added to the plot using add=TRUE
```

VISUALIZING DATA

```r
# So far you've been visualizing just the list of output numbers
# Except for the last example where I snuck in a ‘hist’ function.
# You can also visualize all of the variables that you've created using the
# ’plot' function (as well as a number of more sophisticated plotting
# functions).
# Each of these is called a ‘high level’ plotting function, which sets the
# stage
# ‘Low level’ plotting functions will tweak the plots and make them
# beautiful
# What do you think that each of the arguments means for the plot function?
# A cool thing about R is that the options for the arguments make sense.
# Try adjusting an argument and see if it works



seq_1 <- seq(0.0, 10.0, by = 0.1)
plot (seq_1, xlab="space", ylab ="series 1", type = "p", col = "red")
```

```r
# You can also combine plots together into a single figure.


seq_1 <- seq(0.0, 10.0, by = 0.1)
  print(seq_1)
seq_2 <- seq(10.0, 0.0, by = -0.1)
  print(seq_2)

par(mfrow=c(2,2))
  plot (seq_1, xlab="time", ylab ="p in population 1", type = "p", col = 'red')
  plot (seq_2, xlab="time", ylab ="p in population 2", type = "p", col = 'green')
  plot (seq_square, xlab="time", ylab ="p2 in population 2", type = "p", col = 'blue')
  plot (seq_square_new, xlab="time", ylab ="p in population 1", type = "l", col = 'yellow')


# The first line of the lower script tells R that you are going to create a composite figure
# that has two rows and two columns. Can you tell how?
# Now, modify the cody to add two more variables and add one more row of two panels.
```

```r
# As above for the normal distribution, data can be generated by being sampled from
# nearly any distribution and then visualized.

# Below I'm having you use the 'histogram' function. What does it do?

# Binomial Distribution
# function rbinom takes three parameters
#  1. The number of observations to generate
#  2. The number of trials for each observation
#  3. Probability of a success

b <- rbinom(n=100, size=20, prob=0.5)
hist(b)


#10 successes (out of 20 trials) is the most frequent outcome

# This kind of statement can be run in one line as well, which is sometimes easier.

hist(rbinom(n=100, size=20, prob=0.5))
```

DATA FRAMES = DATA SETS IN R

```
# As you have seen, in R you can generate your own random data set drawn from
# nearly any distribution very easily. Often we will want to use collected data.
# Now, let’s make a dummy dataset to get used to dealing with data frames
# Set up three variables (habitat, temp and elevation) as vectors

habitat <- factor(c("mixed", "wet", "wet", "wet", "dry", "dry", "dry","mixed"))
temp <- c(3.4, 3.4, 8.4, 3, 5.6, 8.1, 8.3, 4.5)
elevation <- c(0, 9.2, 3.8, 5, 5.6, 4.1, 7.1, 5.3)




# create a data frame where vectors become columns

mydata <- data.frame(habitat, temp, elevation)




# append row names to the data frame

row.names(mydata) <- c("Reedy Lake", "Pearcadale", "Warneet", "Cranbourne",
"Lysterfield", "Red Hill", "Devilbend", "Olinda")
```

```
# A strength of R is being able to import data from an external source
# Create the same table that you did above in a spreadsheet like Excel.
# Export it to comma separated and tab separated text files for importing into R.
# The first will read in a comma-delimited file, whereas the second is a tab-delimited.
# In both cases the header and row.names arguments indicate that there is a header row
# and row label column. Not that the name of the file by itself will have R look in the
# CWD, whereas a full path can also be used.


YourFile <- read.table('yourfile.csv', header=T, row.names=1, sep=',')
YourFile <- read.table('yourfile.txt', header=T, row.names=1, sep='\t')


# Exporting a data frame as a comma-delimited or tab-delimited file


write.table(YourFile, "yourfile.csv", quote=F, row.names=T, sep=",")
write.table(YourFile, "yourfile.txt", quote=F, row.names=T, sep="\t")
```

INDEXING IN DATA FRAMES

```
# Next up - indexing just a subset of the data
# This is a very important idea in R, that you can analyze just a subset of the data.
# This is analyzing only the data in the file you made that has the factor value 'mixed'.

print (YourFile[,2])
print (YourFile$temp)
print (YourFile[2,])
plot (YourFile$temp, YourFile$elevation)


# You can perform operations on particular levels of a factor
# Calculating the mean of the 'mixed' and 'gipps' levels of habitat. Note that the first
# argument is the numerical column vector, and the second is the factor column vector.
# The third is the operation. Reversing the first two does not work (the one below).

tapply(YourFile$temp, YourFile$habitat, mean)
tapply(YourFile$temp, YourFile$habitat, var)
```

OK, SOME REAL TRANSCRIPTOMIC DATA

Initial examination of RNA-seq data

- Examine the text file: **`GacuRNAseq.csv`**
- How many many rows and columns are there?
- How many different variables are there?
- What are the general types of variables?
- Now let's read the data file into R and analyze it

```
# This exercise will help you get used to reading in and manipulating genomic data files
# First off, remember to set your working directory to find your file correctly

RNAseq_Data <- read.table(‘GacuRNAseq.csv', header=T, sep=',')

# OK, now let's look at the data

print (RNAseq_Data)

# Whoa, that’s a lot of data
# This is a little easier

head (RNAseq_Data)
tail (RNAseq_Data)

# How many columns are there? How many variables?
# Write down what you think are the characteristics of each
# How do we look at a subset of the data?
# Can you see how we can do the same thing different ways?

print (RNAseq_Data[,2])
print (RNAseq_Data[1,])
print (RNAseq_Data[1,2])
print (RNAseq_Data$ENSGACG00000000010)
print (RNAseq_Data$ENSGACG00000000010>45.0)
```

OK, let's try some summary stats and figures

```
summary1 <- summary(RNAseq_Data $ENSGACG00000000003)
print (summary1)

hist(RNAseq_Data $ENSGACG00000000003)

boxplot(RNAseq_Data$ENSGACG00000000003)
boxplot(RNAseq_Data$ENSGACG00000000003~RNAseq_Data$Population)

plot(RNAseq_Data $ENSGACG00000000003, RNAseq_Data$ENSGACG00000000003)


boxplot(RNAseq_Data $ENSGACG00000000003~RNAseq_Data$Treatment, col = "red", ylab
= “Expression Level", xlab = "Treatment level", border ="orange", main =
"Boxplot of variation in gene expression across microbiota treatments")
```

The Tidyverse https://www.tidyverse.org

Tidyverse

Packages Articles Learn Help Contribute

R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying philosophy and common APIs.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

```
# Hadley Wickham and others have written R packages to modify data

# These packages do many of the same things as base functions in R

# However, they are specifically designed to do them faster and more easily

# Wickham also wrote the package GGPlot2 for elegant graphics creations

# GG stands for ‘Grammar of Graphics’
```

```
install.packages("tidyverse")

library(tidyverse)
```