Last time

We went over classes in R again and exhibited a simple implementation of the Shazam algorithm -- Our goal was not to “solve” the problem for you, but to provide you with a lower bound and an existence proof

You are not expected to make use of this code, but instead we hoped that you’d improve on it! You are also not expected to program in R (we’re just starting after all) but R is good for getting acquainted with the suite of new concepts and tools this project has thrown at you

Also, that humble piece of code was not a complete working system -- At best it computes some small part of a larger chain of tasks that you all mapped out with your previous assignment
shazam <- function(snd){

spec <- specgram(snd$sound, 64*8000/1000, 8000, 64*8000/1000)
map <- find_constellation(abs(spec$S))

mfreqs = matrix(1:length(spec$f), ncol=length(spec$t), nrow=length(spec$f))
mtimes = matrix(1:length(spec$t), ncol=length(spec$t), nrow=length(spec$f), byrow=T)

const <- cbind(mtimes[map], mfreqs[map])

hashes <- NULL

for(i in 1:nrow(const)){

# we are going to process each song in overlapping 15 second chunks and
# want to stop building constellation/target segments at 10 seconds

if(const[i,1] < 314){

    candidates <- const[const[,1]>const[i,1],]
    candidates <- candidates[order(idist(const[i,], candidates)),]

    for(j in 1:min(c(10, nrow(candidates)))){

        hashes = rbind(hashes, c(const[i,],
                                   ihash(const[i,2], candidates[j,2], candidates[j,1]-const[i,1])))
    }
}

return(hashes)
}
> sinshaz <- shazam(sinatra[(10*8000+1):(25*8000)])
> jzshaz <- shazam(jz[(50*8000+1):(65*8000)])

# test to see if we have any hashes that match between jay-z and sinatra...

> sum(jzshaz[,3] %in% sinshaz[,3])
[1] 4

# now try an unknown sample recorded by my laptop

> unknown <- loadSample("longsample.wav")
> sunknown <- unknown[(30*8000+1):(45*8000)]
> ushaz <- shazam(sunknown)

> sum(ushaz[,3] %in% sinshaz[,3])
[1] 10
> dim(ushaz)
[1] 1890 3

> sum(ushaz[,3] %in% jzshaz[,3])
[1] 86

> plot(ushaz[,1:2],pch=19,col=5,cex=0.5,xlab="time",ylab="frequency")
> points(ushaz[,1:2][ushaz[,3] %in% jzshaz[,3],])

> sunknown <- unknown[(35*8000+1):(50*8000)]
> ushaz <- shazam(sunknown)
> sum(ushaz[,3] %in% jzshaz[,3])
[1] 483

> plot(ushaz[,1:2],pch=19,col=5,cex=0.5,xlab="time",ylab="frequency")
> points(ushaz[,1:2][ushaz[,3] %in% jzshaz[,3],])
Finally

We can make the plots that we scan for peaks in the signal -- First we plot the time of the hashcode matches for the Jay-Z sample versus those for the unknown sample

Then, recalling that strings of matches should line up on a line with unit slope (a time offset should appear as a shift in intercept) we can scan a plot for diagonal features -- Or we can take a histogram of the differences in times which should cluster around the value of the time offset

```r
> sunknow <- unknown[(5*8000+1):(20*8000)]
> ushaz <- shazam(sunknown)

# restricted range on the x-axis so that a line with unit slope looks like it
> mm <- match(jzbig[,3],ushaz[,3],
              xlab="jz-time",ylab="sample-time",xlim=c(500,800))

> hist(jzbig[,1][!is.na(mm)]-ushaz[,1][mm[!is.na(mm)]],
       breaks=100,xlab="time differences",main="seconds 5-20 of unknown",cex=0.5)
```
seconds 25–40 of unknown
seconds 35–50 of unknown
Today

We are going to start with a review of some of the mechanics behind SSH, specifically public-key authentication -- It was something we should have mentioned a loooong time ago

We are forced to bring it up today because we will also talk about code sharing via GitHub, another dangling thread

We'll then finish looking at the last few introductory data types in R, namely lists and dataframes and factors...
Public-key authentication

So far, our work with SSH has been mostly based on password-based authentication -- That is, each time you invoke the command, the machine you are communicating with requests a password.

As an alternative, public-key authentication relies on a generated public/private keypair -- The public key is used to encrypt data which can only be read by a person who has access to the associated private half of the key.
Key-based authentication

In short, having access to the public half of a key allows you to send secret information to anyone with the private half, but also lets you verify that a person does in fact have access to the private half.

As a user, you can generate a keypair and then place the public half of the key on a remote system (like homework.stat202a.org) -- That remote system is then able authenticate you, or prove that you are really you, and allow you to login just by having you demonstrate that you have access to the private half of the keypair.

Of course this means that your “identity” has been mapped to the fact that you have access to your private key -- This means you really need to keep your private key private!
Public-key authentication

To make this less abstract, let’s create a keypair on our local Mac’s -- You will be prompted to name the keypair and to assign a passphrase.

The passphrase is used like a password and it will mean that whenever the system tries to decrypt a message using this private key, you will be prompted to enter the phrase again (In some situations we want to ship files around in a trusted system without passwords and we could leave the passphrase blank).
% ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/cocteau/.ssh/id_rsa):
/Users/cocteau/.ssh/git_rsa
Enter passphrase (empty for no passphrase): on a winter’s night a traveller
Enter same passphrase again: on a winter’s night a traveller
Your identification has been saved in /Users/cocteau/.ssh/git_rsa.
Your public key has been saved in /Users/cocteau/.ssh/git_rsa.pub.
The key fingerprint is:
The key's randomart image is:
+--[ RSA 2048]-----+
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
Public-key authentication

We have named this keypair after Git because we will need a keypair to update files on GitHub shortly -- In principle you could make many pairs, each for a different machine you need to login to regularly (although many people simply use the same pair and deposit the public key on each machine they routinely access)

Notice from the previous slide that the files are not globally readable -- Who has access to each? (On the next page we provide a snippet from the help pages on ssh that describe what permissions each file needs to have)
~/.ssh/
This directory is the default location for all user-specific configuration and authentication information. There is no general requirement to keep the entire contents of this directory secret, but the recommended permissions are read/write/execute for the user, and not accessible by others.

~/.ssh/authorized_keys
Lists the public keys (RSA/DSA) that can be used for logging in as this user. The format of this file is described in the sshd(8) manual page. This file is not highly sensitive, but the recommended permissions are read/write for the user, and not accessible by others.

~/.ssh/config
This is the per-user configuration file. The file format and configuration options are described in ssh_config(5). Because of the potential for abuse, this file must have strict permissions: read/write for the user, and not accessible by others.

~/.ssh/identity
~/.ssh/id_dsa
~/.ssh/id_rsa
Contains the private key for authentication. These files contain sensitive data and should be readable by the user but not accessible by others (read/write/execute). ssh will simply ignore a private key file if it is accessible by others. It is possible to specify a passphrase when generating the key which will be used to encrypt the sensitive part of this file using 3DES.

~/.ssh/identity.pub
~/.ssh/id_dsa.pub
~/.ssh/id_rsa.pub
Contains the public key for authentication. These files are not sensitive and can (but need not) be readable by anyone.
Managing identities

As we mentioned, you can have different keypairs for different machines -- To manage which private key to use when decrypting a message from a remote server, you can either use a command line option to ssh or you can set up a config file (that should only be readable/writeable by you)

```
% vi .ssh/config
% cat .ssh/config
Host github.com
  IdentityFile ~/.ssh/git_rsa

% chmod 600 .ssh/config
% ls -l
drwx------  6 cocteau  staff   204 Nov 8 10:40 .
drwxr-xr-x+ 252 cocteau  staff  8568 Nov 8 10:40 ..
-rwx------  1 cocteau  staff   46 Nov 8 10:40 config
-rw-------  1 cocteau  staff  1675 Nov 8 10:36 git_rsa
-rw-r--r--  1 cocteau  staff  410 Nov 8 10:36 git_rsa.pub
-rw-r--r--  1 cocteau  staff  4335 Nov 2 08:49 known_hosts
```
Public-key authentication

For those of you on a Windows machine, Putty provides you with a GUI to accomplish the same tasks -- This site provides a pretty simple guide through the process

http://www.howtoforge.com/ssh_key_based_logins_putty

Now, let's use this key for something!
Sharing code

Previously we had discussed the use of Git both as a tool for keeping track of the changes you make to a body of code as well as a platform for sharing code with others.

We mentioned at the time platforms like GitHub that provide you with a number of added features that help make sharing a little smoother, and could let you painlessly involve the help of many (possibly unknown to you) other developers.

For the moment, we’ll use the “free” version of GitHub, but I am more than happy to fund groups that prefer private repositories...
Secure source code hosting and collaborative development – GitHub

450,000 people hosting over 1,378,000 git repositories

jQuery, reddit, Sparkle, curl, Ruby on Rails, node.js, ClickToFlash, Erlang/OTP, CakePHP, Redis, and many more

GitHub

\verb+git+

Git is an extremely fast, efficient, distributed version control system ideal for the collaborative development of software.

\verb+git\,hub+

GitHub is the best way to collaborate with others. Fork, send pull requests and manage all your public and private git repositories.

Plans, Pricing and Signup

Unlimited public repositories are free!

Free public repositories, collaborator management, issue tracking, wikis, downloads, code review, graphs and much more...

Reliable code hosting

We spend all day and night making sure

30 seconds to give people access to

Team management

Code review

Open source collaboration

Participant in the most important open
Signing up

The sign-up process is relatively straightforward and does not ask for a lot of personal information -- The next few screen shots show you what to expect...
Sign up for GitHub

$0/mo
You are signing up for the free plan
The cost for this plan is $0 per month. You can cancel, downgrade, or upgrade at any time.

Create your free personal account

Username
cocteau

Email Address
cocteau@stat.ucla.edu
We promise we won’t share your email with anyone.

Password

Confirm Password

By clicking on “Create an account” below, you are agreeing to the Terms of Service and the Privacy Policy.

You’re joining the smartest companies in the world

• Email support
• Upgrade, downgrade or cancel at any time
• Secure, reliable, always-available repository hosting
Welcome to GitHub! What's next? (just now)

Create a Repository
Tell us about yourself
Browse Interesting Repos
Learn more about Git and GitHub

Your Repositories (0)

You don't have any repositories yet!
Create your first repository or learn more about Git and GitHub
The basic model

The GitHub interface tells you what to do to create a new (public) repository -- At some point, however, it dips back down to command line instructions to set up your own **private local version of the material** you’ll create

When working with a site like GitHub, your workflow will typically involve making changes to your local files and then periodically “pushing” them back to your public repository

Your personal repo  You push  Your public repo
Create a New Repository

Project Name
music tag

Description (optional)

Homepage URL (optional)

Who has access to this repository? (You can change this later)
- Anyone (learn more about public repos)
- Upgrade your plan to create more private repositories!

Note
If you intend to push a copy of a repository that is already hosted on GitHub, please fork it instead.

Create Repository
Global setup:
Download and install Git
`git config --global user.name "Your Name"`
`git config --global user.email cocteau@stat.ucla.edu`

Next steps:
`mkdir next_homework`
`cd next_homework`
`git init`
`touch README`
`git add README`
`git commit -m 'first commit'`
`git remote add origin git@github.com:cocteau/next_homework.git`
`git push origin master`
SSH keys

The first step involves setting up an SSH key pair that will be used to authenticate sessions you initiate (mostly `scp` or secure copy commands) -- The next couple of screens show you what to expect...
You must have at least one SSH public key to push your git repo to GitHub.

Add another public key

You must have at least one SSH public key to push your git repo to GitHub.

**Title**

```
my-git-key
```

**Key**

```
uhGQVUC1ATROgBlL0DqDrn6F45618VD5xhQbZAvAOBZ2gDv9bC4S5z+yS0lL5JW1HBlUmBGRyyhlpJK938NzGNOlXZ2Sxt688en9jxMrUzh2m5bPz4VW
onzAgbQvMNjgrrW35MSY02GC6PeQiyyKpOR2kXpoC0mGvU4S3Y19rZKKrRJH
9h4KvazJ60WXNxEe7Hv1A5KQc2Z85RvVx1kmcc2ZRLxyNzFIdD1B12p+
9y1A4n0+61C6WbjmxyhPivD0rTrxqOCt5QzX5LWionhVcDISB679Xf7J
0b/1px0qi31Dym4Akmt6eh4qOukvQ6w== cocteau@MacsbookPro.local
```

Add key or cancel
Now, follow their instructions...

As promised, we’re back at the command line to create a basic repository on your local machine -- Here “local” could refer to our homework.stat202a machine or your laptop

We’ve seen many of the commands in a previous lecture, but they probably seemed pretty abstract at the time -- Here we are going to create a directory for our project and initialize it as a Git repository

Then we will create (touch) a README file and commit it to the local repository -- We then “point” this collection back to GitHub, setting its origin to be the public repository, and “push” our file there
Global setup:
Download and install Git
  git config --global user.name "Your Name"
  git config --global user.email cocteau@stat.ucla.edu

Next steps:
  mkdir next_homework
  cd next_homework
  git init
touch README
  git add README
  git commit -m 'first commit'
  git remote add origin git@github.com:cocteau/next_homework.git
  git push origin master
% git config --global user.name "cocteau"
% git config --global user.email cocteau@stat.ucla.edu

% mkdir music-tag
% cd music-tag

% git init
Initialized empty Git repository in /Users/cocteau/music-tag/.git/

% touch README
% git add README

% git commit -m 'first commit'
[master (root-commit) ee0002c] first commit
  0 files changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 README

% git remote add origin git@github.com:cocteau/music-tag.git
% git push origin master
The authenticity of host 'github.com (207.97.227.239)' can't be established.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,207.97.227.239' (RSA) to the list of known hosts.
Counting objects: 3, done.
Writing objects: 100% (3/3), 203 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:cocteau/music-tag.git
  * [new branch] master -> master
cocteau / music-tag

Source Commits Network Pull Requests (0) Fork Queue Issues (0) Wiki (0) Graphs

Switch Branches (1) ▾ Switch Tags (0) Branch List

click here to add a description
click here to add a homepage

SSH HTTP Git Read-Only git@github.com:cocteau/music-tag.git This URL has Read+Write access

first commit
cocteau (author)
just now

music-tag /

name age message

file README just now first commit [cocteau]

README
Adding more files

To add more files, say some of the exceedingly lame R code we created last time, we follow basically the same steps (add and then commit it to the local repository and then push the changes to GitHub)

Let’s see how that works...
```r
find_constellation <- function(mat){
    const <- matrix(TRUE, nrow=nrow(mat), ncol=ncol(mat))
    for(i in 1:16){
        strip <- mat[((i-1)*16+1):(i*16),]
        maxs <- apply(strip,2,max)
        mmaxs <- matrix(maxs,nrow=16,ncol=ncol(strip),byrow=T)
    }
```

```
cocteau / music-tag

Source
Commits
Network
Pull Requests (0)
Fork Queue
Issues (0)
Wiki (0)
Graphs
Branch: master

Switch Branches (1)  Switch Tags (0)  Branch List

click here to add a description
click here to add a homepage

SSH  HTTP  Git Read-Only  git@github.com:cocteau/music-tag.git

This URL has Read+Write access

second commit, added R code

commit 5b59e8cde93f7405cc27
tree 592b3f3952f9a6b281bc
parent ee002c4e12ed56f43dc

music-tag /

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>README</td>
<td>8 minutes ago</td>
<td>first commit [cocteau]</td>
</tr>
<tr>
<td>tag_code.R</td>
<td>just now</td>
<td>second commit, added R code [cocteau]</td>
</tr>
</tbody>
</table>

README
The basic model

Again, when working with a site like GitHub, your workflow will typically involve making changes to your local files and then periodically “pushing” them to your this site.

GitHub emphasizes shared development through its social networking functions, allowing you to build on the work of others, whether or not you've known them prior to obtaining their code (for a fee it also supports more private versions of code sharing).

Here, for example, is a recent project to make MongoDB accessible through R...
R client to interface with MongoDB — Read more

```
show collections docs
```

RMongo/

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Rbuildignore</td>
<td>October 14, 2010</td>
<td>dbGetQuery only supports data.frame access stru... [tc]</td>
</tr>
<tr>
<td>.gitignore</td>
<td>October 01, 2010</td>
<td>initial import [tc]</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>4 days ago</td>
<td>show collections docs [tc]</td>
</tr>
<tr>
<td>NAMESPACE</td>
<td>4 days ago</td>
<td>show collections docs [tc]</td>
</tr>
<tr>
<td>R/</td>
<td>October 25, 2010</td>
<td>using control A as the separator character to a... [tc]</td>
</tr>
</tbody>
</table>
The basic model

By creating a fork (in our Git discussion, we referred to this as branching), you pull a copy of someone’s code for you to work with -- Again, you pull this ultimately ends by placing code in your own personal repository to work on, maybe using with other code you write.

If you want to contribute your changes to their branch, you can make a request via the GitHub interface -- In your groups, it means that your development loop might look like the following:

- Your personal repo
- You push
- Your public repo
- You pull
- Their public repo
- They pull
- They push
- Their personal repo
- They push
The basic model

So, suppose I want to add to the R-MongoDB interface being developed because, for example, at the moment it seems to lack some key features like authentication -- To do this I create my own personal repository of the code using the clone command

```
% git clone git@github.com:cocteau/RMongo.git
Cloning into RMongo...
remote: Counting objects: 263, done.
remote: Compressing objects: 100% (94/94), done.
remote: Total 263 (delta 96), reused 263 (delta 96)
Receiving objects: 100% (263/263), 2.86 MiB | 830 KiB/s, done.
Resolving deltas: 100% (96/96), done.
% cd RMongo/
% ls
DESCRIPTION  R  inst  src
NAMESPACE    README  man  tests```
% vi README
% head README

Trying my hand at making this look more like pymongo

RMongo
http://github.com/quid/RMongo
----
@tommychheng

This is a R package which lets you query MongoDB databases.

% git commit -a -m "my first edit"
[master f40cc4a] my first edit
  1 files changed, 2 insertions(+), 0 deletions(-)

% git push origin master
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 332 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@github.com:cocteau/RMongo.git
c84a6a9..f40cc4a  master -> master
R client to interface with MongoDB — Read more
click here to add a homepage

```
my first edit
cocteau (author)
just now
```

### RMongo /

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>message</th>
<th>history</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Rbuildignore</td>
<td>October 14, 2010</td>
<td>dbGetQuery only supports data.frame access stru... [tc]</td>
<td></td>
</tr>
<tr>
<td>.gitignore</td>
<td>October 01, 2010</td>
<td>initial import [tc]</td>
<td></td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>4 days ago</td>
<td>show collections docs [tc]</td>
<td></td>
</tr>
<tr>
<td>NAMESPACE</td>
<td>4 days ago</td>
<td>show collections docs [tc]</td>
<td></td>
</tr>
<tr>
<td>R/</td>
<td>October 25, 2010</td>
<td>using control A as the separator character to a... [tc]</td>
<td></td>
</tr>
<tr>
<td>README</td>
<td>just now</td>
<td>my first edit [cocteau]</td>
<td></td>
</tr>
<tr>
<td>inst/</td>
<td>October 25, 2010</td>
<td>using control A as the separator character to a... [tc]</td>
<td></td>
</tr>
</tbody>
</table>
You're asking **quid** to pull 1 commit into **quid:master** from **cocteau:master**

**People to be notified**
- agius
- bkudria
- mikejihbe
- tc

Send pull request
Social networking

Because GitHub has fully embraced the benefits of “social coding”, it’s not surprising that they both keep and present a variety of information about you and your coding activities.

Our brief encounter with GitHub makes us look most industrious (although crushingly unpopular).
cocteau's Profile - GitHub

Member Since: Nov 08, 2010

2 public repos 0 followers

Following 0 coders and watching 3 repositories view all →

Public Repositories (2)

- **RMongo**
  - Forked from quid/RMongo
  - R client to interface with MongoDB
  - Last updated 5 minutes ago

- **music-tag**
  - Last updated 22 minutes ago

Public Activity

- cocteau pushed to master at cocteau/RMongo 5 minutes ago
- cocteau forked quid/RMongo 8 minutes ago
- cocteau pushed to master at cocteau/music-tag 22 minutes ago
- cocteau created branch master at cocteau/music-tag 30 minutes ago

f40cc40: my first edit
5b59e8c: second commit, added r code
New branch is at cocteau/music-tag/tree/master
Switching gears...

We’re now going to cover a few of the remaining basic data types in R -- I wanted our test case to be factual.com but the JSON format they publish is not the simplest to swallow directly into R

Therefore, we’re going to fall back on an old favorite...
A new Twitter experience is coming! Learn more about it... #NewTwitter

The best way to discover what's new in your world.

Top Tweets

SkillzVa If she's amazing, she won't be easy. If she's easy, she won't be amazing. If she's worth it, you won't give up. If you give up, you're not worthy.

about 1 hour ago

LittleBeadles http://t.co/mLhf3L via @youtube help get the numbers up. Leaving today for Nashville. Thanks 2 all of your support!

32 minutes ago
The Twitter API

We have talked previously about APIs that advertise data in a kind of service -- Next time we'll talk a bit more explicitly about the styles of APIs that have emerged but for the moment we're mostly interested in getting some data to work with

So to be very practical, using the Twitter API is not more than making a request for a web address...
# read in 20 of the latest status updates from twitter
# create a connection to the site and then transform the data into a “data frame”

> api_call <- url("http://api.twitter.com/1/statuses/public_timeline.xml")
> df <- xmlToDataFrame(readLines(api_call))

> class(df)
[1] "data.frame"

> dim(df)
[1] 20 16

> names(df)
[1] "created_at" "id"  
[3] "text" "source"  
[5] "truncated" "favorited"  
[7] "in_reply_to_status_id" "in_reply_to_user_id"  
[9] "in_reply_to_screen_name" "retweet_count"  
[11] "retweeted" "user"  
[13] "geo" "coordinates"  
[15] "place" "contributors"

# what kinds of methods do we know about for data frames?
Data structures in R

Continuing our explanation of the data types in R, we now move on to lists and data frames.

vectors: ordered collections of primitive elements

matrices and arrays: rectangular collections of primitive elements having dimension 2,...

lists: a kind of generic vector, where elements can be of mixed (not-necessarily primitive) type

data frames: two-dimensional data tables

factors: categorical variables
Lists

Lists can be used to store items that are not all of the same type; in fact the types that can be socked away are more general than what you could put in a vector or a matrix.

There are three ways to pull data from a list; the usual [ operator, a [[ double version, and by selecting specific named entries with $\text{name}$.
> x <- list(a=50, b=2, cat=3:5, lightbulb="some characters")
> class(x)
[1] "list"

> is.vector(x)  # we can think of it as a vector
[1] TRUE

> is.matrix(x)
[1] FALSE

> length(x)    # how long?
[1] 4

> x            # have a look at it
$a
[1] 50

$b
[1] 2

cat
[1] 3 4 5

$lightbulb
[1] "some characters"
> x[1:2]  # single ['s return another list but with just the entries selected
  $a
  [1] 50

$b
  [1] 2

> x[[1]]  # double [[']s return the actual value stored in the list (just one)
  [1] 50

> x$cat  # ... as does "$" with the name of the entry you want
  [1] 3 4 5

> names(x)
  [1] "a" "b" "cat" "lightbulb"

> x <- list(a=50,2,cat=3:5,lightbulb="some characters")  # drop name "b"
> names(x)
> names(x)
  [1] "a" "" "cat" "lightbulb"

> x[1:2]
  $a
  [1] 50

    [[2]]
  [1] 2
Data frames

Some of the syntax on the previous slides should have warmed you up to the notation we saw when we scanned in the USGS data; that’s because data frames are a special kind of list, special in the sense that all the elements are vectors, each with the same length.

We think of these elements as column vectors and even index them as we would a matrix; unlike a matrix, however, the columns in a data frame do not have to be the same kind of thing (each separate column can be of a different type).

Again, you can think of a data frame as R’s answer to a two-dimensional spreadsheet; from the standpoint of statistical applications, it was created to handle the (very common) setting in which experimental data are stored as a table (rows indicating runs or experimental units, columns indicating a common set of measurements taken at each run).
> is.vector(df)
[1] FALSE
> is.matrix(df)
[1] FALSE
> is.list(df)
[1] TRUE

# what kinds of things are the columns? how do we find out?

> names(df)
[1] "created_at"              "id"
[3] "text"                    "source"
[5] "truncated"               "favorited"
[7] "in_reply_to_status_id"   "in_reply_to_user_id"
[9] "in_reply_to_screen_name" "retweet_count"
[11] "retweeted"               "user"
[13] "geo"                     "coordinates"
[15] "place"                   "contributors"

# we can make some interesting assignments as well...
> names(df)[2] <- "index"

> names(df)
[1] "created_at"              "index"
[3] "text"                    "source"
[5] "truncated"               "favorited"
[7] "in_reply_to_status_id"   "in_reply_to_user_id"
[9] "in_reply_to_screen_name" "retweet_count"
[11] "retweeted"               "user"
[13] "geo"                     "coordinates"
[15] "place"                   "contributors"
Data frames

As we said previously, data frames are a bit like matrices except that the columns can be of different types; the subsetting rules for a data frame are essentially the same as those for matrices.

So simple row and column selections via like [,3] and [5:10,] make sense (and again, you can use indices or names or a logical mask, just as you would for a matrix).

Note that whenever you ask for more than one column, you end up with a data frame; single columns are no longer data frames but are vectors (or factors, depending on the underlying type of data in the column).
Data frames

Because data frames are lists and not vectors, however, **some of the tricks we played with matrix subsetting don’t work**

For example, a single subscript like \texttt{z[1:1000]} when \texttt{z} is a matrix, will force R to stretch \texttt{z} into a vector by appending all its columns; but for a data frame, the single subscript operates like it would for a list and so \texttt{z[1:2]} is the same as asking for the two columns \texttt{z[,1:2]}

Because a data frame is also a list, **you can extract columns using the \$name construction**; again, this returns a single column and so the result is no longer a data frame but a vector (or a factor)
# the last few variables are largely empty...
> levels(df$geo)
[1] ""

> df$geo
[1]
Levels:

# so we might as well drop them (showing off subsetting by exclusion)

> df <- df[, -c(13:16)]

> names(df)
[1] "created_at"          "index"
[3] "text"                 "source"
[5] "truncated"             "favorited"
[7] "in_reply_to_status_id" "in_reply_to_user_id"
[9] "in_reply_to_screen_name" "retweet_count"
[11] "retweeted"            "user"

# or we can make a new data frame with retweeted count and the text of the tweets,
# showing off subsetting by position :)

> newdf <- df[, c(10, 3)]

> class(newdf)
[1] "data.frame"
Data frames

The function `read.csv` is a “wrapper” for `read.table`; that is, it provides a set of good default values that anticipate what a typical comma-separated file looks like.

`read.table`, however, is a wrapper for an even lower-level function called `scan`; this function is better at reading very large data tables.

`read.table` and its progeny are convenient because they try to be smart about creating R variables from the columns in the file; that is, they assign classes to columns in the data frame.

R attempts to coerce the column into logical, integer, numeric and then complex; if that all fails, it will store the data as a factor.
Factors

... and on that note, our running Twitter example has provided us with one more data type: we have seen integers and floats and logical values, but on the previous slides, city has been read as class factor

A factor is a class of object that represents categorial data; the existence of this class has a lot to do with the fact that R (or S) is designed to support statistical calculations

Factors have a dual existence: Their levels are represented as character strings (what you see if you print a factor), but they can be translated into integer values for modeling...
```r
> y <- sample(letters[1:5], 20, rep=T)

> f <- as.factor(y)
> f
[1] a b a b d a b d e e d a d d c e c a c e
Levels: a b c d e

> class(f)
[1] "factor"
> levels(f)
[1] "a" "b" "c" "d" "e"

> f2 = f[1:10]     # subsetting returns a factor with the same levels

> class(f2)
[1] "factor"
> levels(f2)
[1] "a" "b" "c" "d" "e"

> le <- list(vowel=c("a","e"),consonant=c("b","c","d"))
> levels(f) <- le

> f
Levels: vowel consonant
```
Factors

A subclass of factors are the so-called **ordered factors**; remember categorical data can be unordered (like the vowel and consonant example) or ordered (as in low, medium and high)

The command `ordered()` will assign an order to the levels (using a vector you provide); this will be made clear in how the factor prints and in **how it is used in models**
> f <- ordered(f, levels=c("vowel", "consonant"))

> class(f)
[1] "ordered" "factor"

> f
[1] vowel  consonant vowel  consonant consonant vowel  consonant consonant vowel  consonant
[8] consonant vowel  vowel  consonant vowel  consonant consonant consonant
Levels: vowel < consonant

> attributes(f) # information stored as attributes
$levels
[1] "vowel" "consonant"

$class
[1] "ordered" "factor"
Factors

A factor's dual existence (storing integers representing the level number and a separate vector of vector labels) means that they can reduce the amount of space required for storage.

In the Twitter data we have a few factors, the most important in some sense being the text of the tweet itself -- It might be better stored as a character vector...
class(df$created_at)
[1] "factor"

df[,3]
[1] Omw home I already want to take a nap but I have this orientation. All faith in god (:  
[2] Thanks miss. RT @G0ldiielocks @RickeyahAllen happy birthday beautiful!  
[4] Dicho lo anterior, prefiero las manifestaciones espontáneas que las reguladas @elprimerresorte!  
[6] lemme do sum homewrk .. ..  
[8] Eindelyk upstairs na gedwongen JT te moeten kyken, wat erg (L)  
[12] RT @danygutierrezrv: RT @oscarZmr: RT @MarianaaVazquez: Realmente me afecta el poder ver mis cac  
[14] RT @danygutierrezrv: RT @oscarZmr: RT @MarianaaVazquez: Realmente me afecta el poder ver mis cac  
[16] Ignore me..really  
[17] This nigga coco so fake ive been down here for a min now he finally wanna text somebody ... nig  
[18] Ch-ch-check yourself before you wreeeeckkk yaself!
20 Levels: Before You Buy A Rebecca Minkoff MAB Mini Satchel Consider Thes ...

# maybe store the tweets as characters rather than as a factor...
> df[,3] <- as.character(df[,3])
> class(la$city)
[1] "factor"

> df$created_at
8 Levels: Mon Nov 08 21:52:22 +0000 2010 ... Mon Nov 08 21:52:30 +0000 2010

> summary(df$created_at)

Mon Nov 08 21:52:22 +0000 2010 Mon Nov 08 21:52:23 +0000 2010
               2               2
Mon Nov 08 21:52:24 +0000 2010 Mon Nov 08 21:52:25 +0000 2010
               2               2
Mon Nov 08 21:52:26 +0000 2010 Mon Nov 08 21:52:27 +0000 2010
               4               3
Mon Nov 08 21:52:29 +0000 2010 Mon Nov 08 21:52:30 +0000 2010
               2               3
# turning the times into something a bit more useful...
# see the help for `strptime` to make out the string "%a %b %d %H:%M:%S"

> tms <- strptime(as.character(df$created_at), "%a %b %d %H:%M:%S")

> class(tms)
[1] "POSIXt"  "POSIXlt"

# this time class is basically a list
> is.list(tms)
[1] TRUE

> names(tms)
[1] "sec" "min" "hour" "mday" "mon" "year" "wday" "yday" "isdst"

> tms$hour

> tms$sec