Much like the overhead bins on this plane, the Internet is full.

Check back in a bit; your turn is coming up soon.
There's no waiting on r.mobile. Switch to mobile to start surfing now.

Gogo Unlimited Subscribers may sign in at any time.
Enjoy free Wi-Fi on your flight
Happy Holidays from Google Chrome

Enter a valid email address to get started

cocteau@gmail.com

Please send me email offers and news from Gogo®

Connect to Internet

By hitting the button above, you agree to Gogo terms of use.
Your homework

What did you learn from the data?

What questions do we have for the company that provided them to us?

What would you have done differently?
I think I want us to have a go at culling these data for ourselves -- It's a relatively straightforward task to pull data with a handful of search terms from Twitter

Suppose we have a file called interesting.txt that has a single line like “track=harry,potter” -- We can then call the following Unix command to start puddling up data

curl -d @interesting.txt http://stream.twitter.com/1/statuses/filter.json -uUser:Psswd
Your new homework

So, with that as a start, I’d like us to recreate some of the preliminary effort that went into assembling the portion of the data we were handed -- Clearly culling JSON data is not an issue, so what do we have to think about?

Obviously, we’ll need to determine which tweets are really relevant for the movie we’re after and which aren’t (say an English housewife commenting on her husband Harry having a potter in the garden is not what we’re after)

In the two weeks that remain, we are going to again, build a system to do something -- This time, we’ll pull data from Twitter about at least one movie that opens this weekend (so you have some work to do before you disappear for Thanksgiving) as well as some other data to relate it to (instantaneous box office numbers?)
An aside about time...
The time fields have all been brought into R as factors -- Remember that the convenience functions like \texttt{read.csv} try to coerce the data they read into logical, integer, numeric, complex and factor -- But we also have the ability to deal with dates and times.

The timing of events is an important kind of statistical data and there are a few default classes for representing this information -- Reviewing a couple will let us revisit the data-directed programming ideas introduced in previous lectures...
Date-time Conversion Functions to and from Character

Description:

Functions to convert between character representations and objects of classes "POSIXlt" and "POSIXct" representing calendar dates and times.

Usage:

```r
## S3 method for class 'POSIXct':
format(x, format = "", tz = "", usetz = FALSE, ...)
## S3 method for class 'POSIXlt':
format(x, format = "", usetz = FALSE, ...)
## S3 method for class 'POSIXt':
as.character(x, ...)
strptime(x, format="", tz = "", usetz = FALSE, ...)
strptime(x, format, tz = "")
```
The details of the formats are system-specific, but the following are defined by the ISO C / POSIX standard for 'strftime' and are likely to be widely available. A _conversion specification_ is introduced by '\%', usually followed by a single letter or 'O' or 'E' and then a single letter. Any character in the format string not part of a conversion specification is interpreted literally (and '%%' gives '%'). Widely implemented conversion specifications include

'\%a' Abbreviated weekday name in the current locale. (Also matches full name on input.)

'\%A' Full weekday name in the current locale. (Also matches abbreviated name on input.)

'\%b' Abbreviated month name in the current locale. (Also matches full name on input.)

'\%B' Full month name in the current locale. (Also matches abbreviated name on input.)

'\%c' Date and time, locale-specific.

'\%d' Day of the month as decimal number (01-31).

'\%H' Hours as decimal number (00-23).
'\%H' Hours as decimal number (00-23).

'\%I' Hours as decimal number (01-12).

'\%j' Day of year as decimal number (001-366).

'\%m' Month as decimal number (01-12).

'\%M' Minute as decimal number (00-59).

'\%p' AM/PM indicator in the locale. Used in conjunction with '\%I' and *not* with '\%H'. An empty string in some locales.

'\%S' Second as decimal number (00-61), allowing for up to two leap-seconds (but POSIX-compliant OSes will ignore leap seconds).

'\%U' Week of the year as decimal number (00-53) using Sunday as the first day 1 of the week (and typically with the first Sunday of the year as day 1 of week 1). The US convention.

'\%w' Weekday as decimal number (0-6, Sunday is 0).

'\%W' Week of the year as decimal number (00-53) using Monday as the first day of week (and typically with the first Monday of the year as day 1 of week 1). The UK convention.
'%x' Date, locale-specific.

'%X' Time, locale-specific.

'%y' Year without century (00-99). If you use this on input, which century you get is system-specific. So don't! Often values up to 68 (or 69) are prefixed by 20 and 69 (or 70) to 99 by 19.

'%Y' Year with century.

'%z' (output only.) Offset from Greenwich, so '-0800' is 8 hours west of Greenwich.

'%Z' (output only.) Time zone as a character string (empty if not available).

Where leading zeros are shown they will be used on output but are optional on input.

Also defined in the current standards but less widely implemented (e.g. not for output on Windows) are

'%C' Century (00-99): the integer part of the year divided by 100.

'%D' Locale-specific date format such as '%m/%d/%y': ISO C99 says it should be that exact format.
'\%e' Day of the month as decimal number (1-31), with a leading space for a single-digit number.

'\%F' Equivalent to \%Y-%m-%d (the ISO 8601 date format).

'\%g' The last two digits of the week-based year (see '%V').
   (Typically accepted but ignored on input.)

'\%G' The week-based year (see '%V') as a decimal number.
   (Typically accepted but ignored on input.)

'\%h' Equivalent to '%b'.

'\%k' The 24-hour clock time with single digits preceded by a blank.

'\%l' The 12-hour clock time with single digits preceded by a blank.

'\%n' Newline on output, arbitrary whitespace on input.

'\%r' The 12-hour clock time (using the locale's AM or PM).

'\%R' Equivalent to '%H:%M'.

'\%t' Tab on output, arbitrary whitespace on input.
'T' Equivalent to '%H:%M:%S'.

'\%u' Weekday as a decimal number (1-7, Monday is 1).

'\%V' Week of the year as decimal number (00-53) as defined in ISO 8601. If the week (starting on Monday) containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. (Typically accepted but ignored on input.)

For output (and possibly input) there are also '\%O[\dHImMUVwWy]' which may emit numbers in an alternative locale-dependent format (e.g. roman numerals), and '\%E[\CcYyXx]' which can use an alternative 'era' (e.g. a different religious calendar). Which of these are supported is OS-dependent.
The classes mentioned here refer to POSIX (pronounced poz-icks), an acronym that stands for “Portable Operating System Interface [for Unix],” and refers to a body of standards relating to various aspects of the Unix operation system.

The POSIX standards were developed by the IEEE (another, probably closer-to-home acronym standing for the Institute of Electrical and Electronics Engineers) and, in more formal terms, define an API for the various flavors of Unix.

We’ve talked before about all the things an operating system has to do, and this standard covers things like IO, threading and, well, the representation of dates and times (some of our beloved utilities like grep and cut are also part of the standard, as well as one we didn’t cover, join -- any takers?)
# ok, let’s put this all to work...

```r
movies <- read.table("/data/movies/test_list.txt", as.is=T)
names(movies) <- c("movie","timestamp")
head(movies)

<table>
<thead>
<tr>
<th>movie</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saw_3D</td>
<td>2010-11-11 02:30:23</td>
</tr>
<tr>
<td>Saw_3D</td>
<td>2010-11-11 02:31:55</td>
</tr>
<tr>
<td>Saw_3D</td>
<td>2010-11-11 02:31:53</td>
</tr>
<tr>
<td>Saw_3D</td>
<td>2010-11-11 02:31:06</td>
</tr>
<tr>
<td>Saw_3D</td>
<td>2010-11-11 00:59:53</td>
</tr>
<tr>
<td>Saw_3D</td>
<td>2010-11-11 02:19:43</td>
</tr>
</tbody>
</table>

> tail(movies)

<table>
<thead>
<tr>
<th>movie</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>160470</td>
<td>Secretariat 2010-10-03 20:38:20</td>
</tr>
<tr>
<td>160471</td>
<td>Secretariat 2010-10-03 20:35:41</td>
</tr>
<tr>
<td>160472</td>
<td>Secretariat 2010-10-03 20:37:18</td>
</tr>
<tr>
<td>160473</td>
<td>Secretariat 2010-10-03 20:37:03</td>
</tr>
<tr>
<td>160474</td>
<td>Secretariat 2010-10-03 20:34:25</td>
</tr>
<tr>
<td>160475</td>
<td>Secretariat 2010-10-03 23:58:56</td>
</tr>
</tbody>
</table>
```
> head(movies$timestamp)

# The format of these dates is integer year (with century)-integer month-integer day
# and the times are integer hours (0-24):integer minutes:integer seconds

> tms <- strptime(movies$timestamp, "%Y-%m-%d %H:%M:%S", tz="GMT")
> head(tms)

# more on the two classes in a second (no pun intended)
> class(tms)
[1] "POSIXt" "POSIXlt"

> weekdays(tms[1:10])
[1] "Thursday" "Thursday" "Thursday" "Thursday" "Thursday" "Thursday"
[7] "Thursday" "Thursday" "Thursday" "Thursday"

> table(weekdays(tms))[c(2,6,7,5,1,3,4)]
        Monday       Tuesday     Wednesday    Thursday       Friday      Saturday     Sunday
22380  22951  22693       20980  22539       24640      24292
Dates and Times in R

For time, we have two classes, `POSIXlt` and `POSIXct` -- The first represents time in a “broken down” list format with integer values for the month, day, year, etc. -- While the second records time in the number of seconds that have elapsed since the so-called Unix epoch (January 1, 1970) (the names refer to the original POSIX standards for “local_time” and “calendar_time”)

`POSIXt` is a (virtual) class that inherits from both `POSIXlt` and `POSIXct` -- Virtual classes are used to represent objects that share a common set of behaviors but may have different internal representations

Let's see how this plays out...
```r
> round(tms[1:10], units="hours")
[1] "2010-11-11 03:00:00 GMT"  "2010-11-11 03:00:00 GMT"
[3] "2010-11-11 03:00:00 GMT"  "2010-11-11 03:00:00 GMT"
[5] "2010-11-11 01:00:00 GMT"  "2010-11-11 02:00:00 GMT"
[7] "2010-11-11 06:00:00 GMT"  "2010-11-11 05:00:00 GMT"
[9] "2010-11-11 06:00:00 GMT"  "2010-11-11 05:00:00 GMT"

POSIXt is a “virtual class” and inherits from both classes... we’ll see more on this later but a virtual class collects a group of objects that share a common behavior, but may not have the same representation.

> class(tms)
[1] "POSIXt"  "POSIXlt"
> is.list(tms)
[1] TRUE
> names(tms)
[1] "sec"  "min"  "hour"  "mday"  "mon"  "year"  "wday"  "yday"  "isdst"
> tms$hour[1:10]
    [1] 2 2 2 2 0 2 5 5 5 5
> head(tms)
```
# now look at the "calendar time" or POSIXct representation...

> ctms <- as.POSIXct(tms)

> class(ctms)
[1] "POSIXt"  "POSIXct"

# note that appt_start now inherits from POSIXct and not POSIXlt; it's also
# no longer a list of time "pieces"...

> is.list(ctms)
[1] FALSE

# instead, it is stored as seconds since the UNIX epoch, January 1, 1970; we can
# get these values directly by calling as.numeric() or as.integer()

> as.numeric(ctms)[1:10]
[1] 1289442623 1289442715 1289442713 1289442666 1289437193 1289441983
[7] 1289453403 1289452998 1289453430 1289453369

# if you call as.numeric() on the POSIXlt class you get, as you might expect, POSIXct

> as.numeric(tms)[1:10]
[9] "2010-11-11 05:30:30 GMT" "2010-11-11 05:29:29 GMT"

> class(as.numeric(tms)[1:10])
[1] "POSIXt"  "POSIXct"
> head(ctms)

> as.numeric(ctms[1])
[1] 1289442623

# another way to create dates...
> d <- ISOdatetime(2010,11,11,2,30,23,tz="GMT")
> class(d)
[1] "POSIXt" "POSIXct"
> as.numeric(d)
[1] 1289442623

# before the UNIX epoch earns you negative seconds...
> d <- ISOdatetime(1009,11,11,2,30,23,tz="GMT")
> as.numeric(d)
[1] -30299088577
... some uses

Now, suppose we want to compute the time between tweets about different movies, maybe looking at an instantaneous intensity -- Or perhaps we want to “register” each movie according to its release date so that we can directly compare peak and fade-off for different films

Both of these suggest we might want to be able to do arithmetic with times!
```r
> info <- read.csv("/data/movies/info.csv")
> names(info)
[1] "Name" "Opening.Day"
[7] "Keywords" "Filtered.for.Movies."
[9] "X" "X..of.total.keyword.mentions"
> head(info$Opening.Day)

# POSIXct is good for adding to data frames, say -- POSIXlt is itself a list
# structure and not well suited for this
> info$tm <- as.numeric(strptime(as.character(info$Opening.Day), "%m/%d/%Y", tz="GMT"))

# now subset the data, looking only at one movie
> sn <- movies[movies$movie=="The_Social_Network",]
> dim(sn)
[1] 40444 2
> sn$time <- as.numeric(strptime(sn$timestamp, "%Y-%m-%d %H:%M:%S", tz="GMT"))
> head(sn)
          movie                   timestamp                   time
43274 The_Social_Network 2010-11-11 01:21:50 2010-11-11 01:21:50
43275 The_Social_Network 2010-11-11 01:21:07 2010-11-11 01:21:07
43276 The_Social_Network 2010-11-11 01:19:27 2010-11-11 01:19:27
43277 The_Social_Network 2010-11-11 01:19:17 2010-11-11 01:19:17
43278 The_Social_Network 2010-11-11 01:18:50 2010-11-11 01:18:50
```
```r
> sn$time[1]
[1] "2010-09-18 02:44:02 GMT"

> sn$time[1]-sn$time[2]
Time difference of -49.68333 mins

> sn$time[1]
[1] "2010-09-18 02:44:02 GMT"

> sn$time[2]
[1] "2010-09-18 03:33:43 GMT"

> sn$time[1]-sn$time[2]
Time difference of -49.68333 mins

> sn$time[1]+60*60
[1] "2010-09-18 03:44:02 GMT"

> sn$time[1]+60*60*24
[1] "2010-09-19 02:44:02 GMT"

> mean(sn$time)
[1] "2010-10-18 03:56:23 GMT"

> methods("mean")
[1] mean.data.frame mean.Date mean.default mean.difftime
[5] mean.POSIXct mean.POSIXlt
```
# the differences...
> sn <- sn[order(sn$time),]
> head(diff(sn$time))
Time differences in secs
[1]  2981   273  4322  5779  1771  1809

> d <- diff(range(sn$time))

# this has its own class...
> class(d)
[1] "difftime"

# which means there must be something going on here - diff returns a special
# class when operating on a time object. as you might expect...

> diff
function (x, ...)  
UseMethod("diff")
<environment: namespace:base>

> methods("diff")
[1] diff.Date  diff.default diff.POSIXt  diff.ts

# we can also call a differencing function directly...
> difftime(max(sn$time),min(sn$time),units="hours")
Time difference of 1452.232 hours
An aside about data analysis...

The following slides were presented last Wednesday to a group that’s working on the Participatory Sensing curriculum for high school students -- The question entertained during that session was, “When teaching data analysis, should we use R or Python?”

Comments?
Background

Both Python and R are languages that are “on the rise” in the sense that they both have very active communities of users and are becoming widely recognized as tools that people “should know”

From a practical matter, this means that both languages are being actively extended by large groups of developers -- New tools are being added to both languages at an incredible rate and there is often no work required by a user who is interested in working with a new data type (say)

In my class, for example, we just finished implementing the Shazam algorithm which meant the students needed to work with audio signals -- Both languages have the ability to read mp3 and wave files, both provide some basic editing and subsetting capabilities and both provide at least some graphical representation of the signals
Background

You will often see references like “R is another **Swiss Army Knife** of numerical and statistical routines for hacking through the big data sets -- collections big enough that it might be better called a Swiss Army Machete” and that Python is popular among scientists because of their need to “improvise when trying to interpret results, so they are drawn to **dynamic languages** which allow them to work very quickly and see results almost immediately.”

Comments like these underscore the fact that **Python is a general-purpose language** that is often used to “glue” systems together, while **R finds is real strength in its data structures and methods** for dealing with data -- In truth, one could use R as one my use Python and vice versa, although there’s a price to be paid in complexity
7 programming languages on the rise

From Ruby to Erlang, once niche programming language are gaining converts in today’s enterprise

By Peter Wayner | InfoWorld

In the world of enterprise programming, the mainstream is broad and deep. Code is written predominantly in one of a few major languages. For some shops, this means Java; for others, it’s C# or PHP. Sometimes, enterprise coders will dabble in C++ or another common language used for high-performance tasks such as game programming, all of which turn around and speak SQL to the database.

Programmers looking for work in enterprise shops would be foolish not to learn the languages that underlie this paradigm, yet a surprising number of niche languages are fast beginning to thrive in the enterprise. Look beyond the mainstays, and you’ll find several languages that are beginning to provide solutions to increasingly common problems, as well as old-guard niche languages that continue to occupy redoubts. All offer capabilities compelling enough to justify learning a new way to juggle brackets, braces, and other punctuation marks.

[ Keep up on key application development insights with the Fatal Exception blog and Developer World newsletter. | See how the latest Python IDEs and PHP tools fared in our recent InfoWorld Test Center reviews. ]

While the following seven niche languages offer features that can’t be found in the dominant languages, many rely on the dominant languages to exist. Some run on top of the Java Virtual Machine, essentially taking advantage of the Java team’s engineering. And when Microsoft built C#, it explicitly aimed to make the virtual machine open to other languages. That detail may help make deployment easier, but it doesn’t matter much to the programmer at creation time.
Names You Need to Know in 2011: R Data Analysis Software

Simply put by one of its staunchest advocates, “R is the most powerful statistical computing language on the planet; there is no statistical equation that cannot be calculated in R.”

Beyond “just” a language, R is a toolset, a community, and a lot of free software.

“Everyone can, with open source R,” Norman Nie says in Quentin Hardy’s article, “afford to know exactly the value of their house, their automobile,” their current business and prospects. Nie has built a successful business providing services and support for R. (Thanks to community member Johnkolchak for this correction.)

Ross Ihaka and Robert Gentleman, then both at Auckland University in New Zealand, created the R Project informally around 1990. The R Core Team, currently at 19 members, is responsible for the development of the basic R software.
Observations

Part of our goal is to show how time, location, images and text are computable, in addition to the classically statistical data types -- This should frame the way we consider each language

In particular, we want to examine the coding effort required to “work with” each kind of data -- That is, the tradeoff between “ideas” and language acquisition has to favor ideas
Observations

When performing data analysis, we are almost by definition invoking an iterative process that involves producing many different “views” (graphical or numerical) of a data set, possibly incorporating new data sources, and sharing (interactive) analyses with other.

This cycle of “exploratory computing” is different from standard programming -- We are not writing a single, monolithic program to implement a particular procedure, but instead rely on code to construct new “views”, expand or reduce a data set and so on.

With that in mind, our analysis usually starts with an interactive “shell” in which we type commands, have them interpreted and some action executed -- Both R and Python have shells of this kind.
Observations

Python, as a general purpose language, was not designed with data analysis in mind, but a series of projects (SciPy, NumPy, matplotlib and iPython) have all been building structures that are closer in spirit to R (well, in many cases to Matlab which is somewhat similar to R).

In fact, R is (in a pure sense) an implementation of the so-called S language -- That is, it was designed as an open source implementation of a program originally developed at AT&T Bell Laboratories.

There has been discussions with Enthought (the group managing SciPy and NumPy) to try to implement the S language in Python -- I bring this up as another example of how some consensus is forming around what one needs from a language to support data analysis and there are attempts to bring those structures into Python.
Executive Summary

1. There is some **inevitability to the look of a language that supports data analysis** -- Even within a general purpose language like Python, we are seeing moves to adapt its structures to models that have worked well in the past (S/R or Matlab)

2. R is by far **the most advanced language in this regard** and (as we will see later) the structure of the language makes certain kinds of coding simple -- These “easy paths” lead to tools that often have a common “look and feel”

3. Python, as a general purpose language, **can be made to look like R** in terms of its data structures and consistent coding standards -- But the PyLab movement is only partially complete and even our data types in Lesson 1 are not well supported

4. We would have to provide a **coding layer on top of Python**, extending the work of PyLab, to cover all the data types we see in Mobilize -- We will also have to build our own GUI
Supporting EDA

We’re now going to change gears and consider a crucial part of modern EDA, graphics -- The ability to make quick, reliable and informative plots is a mainstay of the practice of telling stories with data

R’s base graphics system is fairly old and sometimes feels that way -- The way in which you can specify colors or line types can be arcane

Today we’ll cover some aspects of the base graphics package in R and next time (because so many of you are likely to have a holiday on your mind), we’ll cover more “advanced systems” like ggplot2
Graphics in R

Over the last few lectures, we have produced several basic plots in R; and at this point you should be familiar with R’s general approach to graphics -- We call a series of graphics functions that either produce a complete plot, or add something to an existing plot.

These added components include adding text and lines and polygons to a plot and can almost be thought of as a set of graphical output primitives -- Many specialty graphics (like those in the package maps, say) make use of these components to build new plots.

We have been referring to this as a “painter’s model,” by which we mean that a plot emerges in steps, each command adding graphical components and obscuring any previous output it overlaps.
Graphics in R

R’s **graphics functions can be divided into a series of packages** that range from creating complete plots to exposing very low-level devices

**Graphics packages:** Specialized plots that are often built from graphics primitives like polygons or lines or text

**Graphics systems:** The so-called *traditional graphics system* consists of the functions in the package `graphics`, while a newer “re-think” of the graphics model can be found in the package `grid`

**A graphics engine:** This consists of the functions in the package `grDevices` and provides basic support for colors and fonts, and exposes several graphics `devices` for producing graphics output in various formats (say a JPEG, a PDF or a PostScript file; or a Mac OS X Quartz window)

**Graphics device packages:** And, as you might expect, there are contributed packages that support other devices (say, a GTK window, a Java Swing window or an SVG file)
> search()
[1] ".GlobalEnv"   "package:stats"   "package:graphics"

> ls("package:grDevices")
[1] "as.graphicsAnnot"   "bitmap"   "blues9"
[4] "bmp"   "boxplot.stats"   "cairo_pdf"
[7] "cairo_ps"   "check.options"   "chull"
[10] "CIDFont"   "cm"   "cm.colors"
[13] "col2rgb"   "colorConverter"   "colorRamp"
[16] "colorRampPalette"   "colors"   "colors"spaces"
[19] "colours"   "contourLines"   "convertColor"
[22] "densCols"   "dev.control"   "dev.copy"
[25] "dev.copy2eps"   "dev.copy2pdf"   "dev.cur"
[28] "dev.interactive"   "dev.list"   "dev.new"
[31] "dev.next"   "dev.off"   "dev.prev"
[34] "dev.print"   "dev.set"   "dev.size"
[37] "dev2bitmap"   "devAskNewPage"   "deviceIsInteractive"
... (output truncated) ...
[88] "X11"   "X11.options"   "X11Font"
[91] "X11Fonts"   "xfig"   "xy.coords"
[94] "xyTable"   "xyz.coords"
> ls("package:graphics")

[1] "abline"          "arrows"          "assocplot"       "axis"
[5] "Axis"            "axis.Date"       "axis.POSIXct"   "axTick"
[9] "barplot"         "barplot.default" "box"           "boxplot"
[13] "boxplot.default" "boxplot.matrix" "bxp"           "cdplot"
[17] "clip"            "close.screen"   "co.intervals"  "contour"
[21] "contour.default" "coplot"         "curve"         "dotchart"
[25] "erase.screen"    "filled.contour" "fourfoldplot"   "frame"
[29] "grconvertX"      "grconvertY"    "grid"          "hist"
[33] "hist.default"    "identify"       "image"         "image.default"
[37] "layout"          "layout.show"    "lcm"           "legend"
[41] "lines"           "lines.default"  "locator"       "matlines"
[45] "matplot"         "matpoints"      "mosaicplot"    "mtext"
[49] "pairs"           "pairs.default"  "panel.smooth"  "par"
[53] "persp"           "pie"            "piechart"      "plot"
[57] "plot.default"    "plot.design"    "plot.new"      "plot.window"
[61] "plot.xy"         "points"         "points.default" "polygon"
[65] "rect"            "rug"            "screen"        "segments"
[69] "smoothScatter"   "spineplot"      "split.screen"  "stars"
[73] "stem"            "strheight"      "stripchart"    "strwidth"
[77] "sunflowerplot"   "symbols"        "text"          "text.default"
[81] "title"           "xinch"          "xspline"       "xyinch"
[85] "yinch"
So far, we have been almost **exclusively using the traditional graphics system** -- When we pulled flight paths of aircraft leaving LAX, for example, \texttt{maps} (the library we used to generate the outline of the continental US) relies on \texttt{polygon} to outline countries and states and counties, a function in the traditional graphics system

Today, we will see an example of \texttt{grid} in action -- It offers the graphics designer **a much wider range of possibilities** for creating new plots at “the cost of having to learn a few additional concepts”
Graphical output formats

When we work with R interactively (typing commands into the interpreter), *graphics output appears in a separate window* (the exact type depending on the computer you’re using) -- We can also **catch the output into a file**, saving it as a PDF or JPEG for use in other applications.

There are **a variety of formats for representing graphical output**, and R selects between them using the concept of **an output device** -- We explicitly **open a device** (specifying, perhaps, a filename), invoke one or more **graphics commands**, and then **close the device**.

R also allows you to **manage many open devices at once** (although, as a practical matter, you will typically have a small number -- 1 if a file, 2 or 3 if windows -- open at one time).
# during an interactive session, this will automatically open a window on your computer...

```r
> hist(sn$time,"hours")
> hist(sn$time,"days")
> hist(sn$time,"weeks")
> hist(sn$time,"months")
```

# we can then save the plot to a file, perhaps as a PDF or a JPEG

```r
> dev.print(device=pdf,file="months.pdf",height=8,width=8)
> dev.print(device=jpeg,file="months.jpg",height=8,width=8)
```

# you could also open a PDF or JPEG file directly... in this case all of the graphics output will be directed to the file and not the screen -- we use these functions when executing in batch mode (commands in a file)

```r
> pdf("days.pdf")
> hist(sn$time,"days")
> dev.off()
```

# The last command closes the file and you can now drop it into a word or pages document, say, or post it to the web -- dev.off is just one of many functions to manage graphics devices...
Histogram of sn\$time

Aug
Sep
Oct
Nov
# Advanced: managing several graphics devices at one time...
# first, shut down all the graphics devices -- dev.off() can be used for individual
# devices, graphics.off() shuts them all off
> graphics.off()

# dev.list produces a list of active devices; at this point we have none
> dev.list()
NULL

# now instantiate a couple -- quartz() opens a graphics window under mac os,
# for the windows operating system, use windows() instead
> pdf("out.pdf")
> quartz()
> dev.list()
    pdf quartz
       2 3

# and one more...
> jpeg("a_plot.jpg")
> dev.list()
    pdf quartz jpeg
       2 3 4

# identifying the current device...
> dev.cur()
jpeg
   4
# set the current device (dev.set and dev.off return the integer value
# of the now active device)
> dev.set(3)
quartz
 3

# invoke a graphics function directed at device 2, the PDF file...
> plot(1:10)

# close the jpeg file, making device 3, the quartz window, the current device
> dev.off(4)
quartz
 3

# and, as expected, there are just two left...
> dev.list()
   pdf quartz
      2      3

# copy the "display list" from the current device, the quartz window, to a new pdf file
> dev.copy(pdf,file="out2.pdf")
> dev.off()
Display lists

Associated with each open device is a **display list**, a record of the graphical output that has been directed to it -- This information is used to redraw the screen, for example, when you resize your graphics window; it is also used when you invoked `dev.print()` which we used to save the plot you created in a window.

In the last command on the previous page, you see that **these lists can also be explicitly copied from the active device to another (possibly new) device**

For the most part, during an interactive session you will likely have only **one or two windows open at a time**, and you will call `dev.print()` to produce graphics for a paper, say -- **In batch mode** (executing a commands from a file), you will make use of the commands like `pdf()` to generate graphical output.
The traditional graphics model

Many of the high-level plotting routines we’ve seen so far are (to flex our newfound vocabulary) generic; and many of the new plots one might encounter are simply new methods for these generics

```r
> hist
function (x, ...)
UseMethod("hist")
<environment: namespace:graphics>

> barplot
function (height, ...)
UseMethod("barplot")
<environment: namespace:graphics>

> boxplot
function (x, ...)
UseMethod("boxplot")
```
The traditional graphics model

As you have seen, specifying the characteristics of a high-level plot can be an interesting exercise -- the size of the text, the font, the colors one might use, all of these things need to be specified in some way

In R, this means that most functions that produce graphical output rely on a series of arguments to alter the look of the final plot

To simplify things somewhat, almost all of the high-level plotting routines respond to a set of so-called standard arguments -- these include axes-oriented features like xlab/ylab (labels on the x- and y-axes), xlim/ylim, main, xaxs/yaxs (how are the ranges for the x- and y-axes chosen), lty (the type of line used), pch (the plotting character used), and col (the color used)
Data symbols

R provides **26 different symbols for you to choose from** (although with some effort you can create your own!) which are set using the `pch=` argument to an agreeable high-level plotting routine.

Numbers **21 though 25 can be assigned outline colors** setting the `fg=` and `bg=` arguments (foreground and background colors -- more later)

```r
# simple scatter plot with different data symbols
> plot(0:25,0:25,pch=0:25)

# and add a vertical line at 20.5 to indicate which # symbols can be assigned outline colors
> abline(v=20.5)
```

Note that **the specification of plotting characters is vectorized**; in this case, it means we specify a different plotting symbol for each point by supplying a vector of symbol indices -- Remember the recycle rule if the length of the vector you supply to `pch` is not the same as the number of points you are plotting!
Data symbols

You can also use single characters as data symbols in a plot...

```r
# simple scatter plot with letters as data symbols
> plot(0:25,0:25,pch=letters)

# equivalently, we can simply add points to an existing
# plot; start with an "empty" or null plot

> plot(c(0,25),c(0,25),type="n")

# ... and add points
> points(0:25,0:25,pch=0:25)
```
Lines

R also supports **pre-defined line types** which you can specify with names like “solid” or “dashed,” or with an integer index `lty` (in a spirit similar to `pch`) -- Line widths are specified with a numeric value `lwd` (where the width associated with `lwd=1` depends on the output device)

It is also possible to customize things like the **way lines join and end**, and particular **patterns for dashed lines**; this is getting a bit advanced so we’ll leave it at that

```r
# again make an empty plot, this time leaving
# off the axes
> plot(c(0,6),c(0,6),type="n",axes=F,
     xlab="",ylab="lty index")

# draw some horizontal lines of different types (note
# that lty 0 is blank!)

> abline(h=0:6,lty=0:6)

# add axes (what does las do?)
> axis(2,las=1)

Again, notice that the line type specification is vectorized; six different horizontal lines and six different line types
Similarly, you have a certain degree of control over the **font that R uses** and the **size of the characters**; for the most part you will be playing with the size, with an argument `cex` (for a relative character expansion; values > 1 mean expansion, < 1 mean reduction) or an argument `ps` (specifying the absolute size in “points”)

You can also control the “**justification**” of a string (the numeric parameter `adj`), you can **rotate the text arbitrarily** (the numeric parameter `srt`), or **rotate just single characters**; again, more detail than you need now, but it’s good to see how infinitely controllable these things are
Colors

We have seen a couple ways to specify colors in R; in some cases, we have specified them by name ("cyan", "magenta", "red", "blue") and you can see all the named colors with a call to the command `colors()`.

You can also refer to colors using their RGB values; the function `rgb()` will create strings that specify colors. `rgb(0.5, 0, 0.5)` returns the string "#800080" which represents a kind of purple (in two slides we show how to go between the name and the RGB specification).

Finally, sets of colors can be specified with `heat.colors()`, `topo.colors()`, `terrain.colors()` where you specify the number of colors you’d like.

The parameter `col` is typically used to control the color of the object you’re plotting; this goes for text and lines and points and even the bars of a barplot...
colors()

[1] "white"                "aliceblue"            "antiquewhite"           "antiquewhite1"
[4] "antiquewhite2"        "aquamarine"            "aquamarine1"            "aquamarine2"
[7] "aquamarine3"          "azure"                 "azure1"                 "azure2"
[10] "azure3"               "azuresky"             "beige"                  "bisque"
[13] "bisque1"              "bisque2"               "bisque3"                "bisque4"
[16] "blanchedalmond"       "blue"                  "blue1"                  "blue2"
[19] "blue3"                "blue4"                  "brown"                  "brown1"
[22] "brown2"               "brown3"                "brown4"                 "burlap"
[25] "burlywood"            "burlywood1"            "burlywood2"             "burlywood3"
[28] "burlywood4"           "cadetblue"             "cadetblue1"             "cadetblue2"
[31] "cadetblue3"           "cadetblue4"            "chartreuse"             "chartreuse1"
[34] "chartreuse2"          "chartreuse3"           "chartreuse4"            "chocolate"
[37] "chocolate1"           "chocolate2"            "chocolate3"             "chocolate4"
[40] "chocolate5"           "chocolate6"            "chocolate7"             "chocolate8"
[43] "chocolate9"           "chocolate10"           "chocolate11"            "chocolate12"
[46] "chocolate13"          "chocolate14"           "chocolate15"            "chocolate16"
[49] "chocolate17"          "chocolate18"           "chocolate19"            "chocolate20"
[52] "chocolate21"          "chocolate22"           "chocolate23"            "chocolate24"
[55] "chocolate25"          "chocolate26"           "chocolate27"            "chocolate28"
[58] "chocolate29"          "chocolate30"           "chocolate31"            "chocolate32"
[61] "chocolate33"          "chocolate34"           "chocolate35"            "chocolate36"
[64] "chocolate37"          "chocolate38"           "chocolate39"            "chocolate40"
[67] "chocolate41"          "chocolate42"           "chocolate43"            "chocolate44"
[70] "chocolate45"          "chocolate46"           "chocolate47"            "chocolate48"
[73] "chocolate49"          "chocolate50"           "chocolate51"            "chocolate52"
[76] "chocolate53"          "chocolate54"           "chocolate55"            "chocolate56"
[79] "chocolate57"          "chocolate58"           "chocolate59"            "chocolate60"
[82] "chocolate61"          "chocolate62"           "chocolate63"            "chocolate64"
[85] "chocolate65"          "chocolate66"           "chocolate67"            "chocolate68"
[88] "chocolate69"          "chocolate70"           "chocolate71"            "chocolate72"
[91] "chocolate73"          "chocolate74"           "chocolate75"            "chocolate76"
[94] "chocolate77"          "chocolate78"           "chocolate79"            "chocolate80"
[97] "chocolate81"          "chocolate82"           "chocolate83"            "chocolate84"
[100] "chocolate85"         "chocolate86"           "chocolate87"            "chocolate88"
[103] "chocolate89"         "chocolate90"           "chocolate91"            "chocolate92"
[106] "chocolate93"         "chocolate94"           "chocolate95"            "chocolate96"
[109] "chocolate97"         "chocolate98"           "chocolate99"            "chocolate100"
[112] "chocolate101"        "chocolate102"          "chocolate103"           "chocolate104"
[115] "chocolate105"        "chocolate106"          "chocolate107"           "chocolate108"
[118] "chocolate109"        "chocolate110"          "chocolate111"           "chocolate112"
[121] "chocolate113"        "chocolate114"          "chocolate115"           "chocolate116"
[124] "chocolate117"        "chocolate118"          "chocolate119"           "chocolate120"
[127] "chocolate121"        "chocolate122"          "chocolate123"           "chocolate124"
[130] "chocolate125"        "chocolate126"          "chocolate127"           "chocolate128"
[133] "chocolate129"        "chocolate130"          "chocolate131"           "chocolate132"
[136] "chocolate133"        "chocolate134"          "chocolate135"           "chocolate136"
[139] "chocolate137"        "chocolate138"          "chocolate139"           "chocolate140"
[142] "chocolate141"        "chocolate142"          "chocolate143"           "chocolate144"
[145] "chocolate145"        "chocolate146"          "chocolate147"           "chocolate148"
[148] "chocolate149"        "chocolate150"          "chocolate151"           "chocolate152"
[151] "chocolate153"        "chocolate154"          "chocolate155"           "chocolate156"
[154] "chocolate157"        "chocolate158"          "chocolate159"           "chocolate160"
[157] "chocolate161"        "chocolate162"          "chocolate163"           "chocolate164"
[160] "chocolate165"        "chocolate166"          "chocolate167"           "chocolate168"
[163] "chocolate169"        "chocolate170"          "chocolate171"           "chocolate172"
# first, the colors R knows by name...
> cc <- colors()
> length(cc)
[1] 657

# here’s a good range...
> cc[20:30]
[1] "bisque1" "bisque2" "bisque3" "bisque4"
[5] "black" "blanchedalmond" "blue" "blue1"
[9] "blue2" "blue3" "blue4"

# examine the rgb value of these...
> col2rgb(cc[20:30])

```
red  255  238  205  139    0  255    0    0    0     0     0
green 228  213  183  125    0  235    0    0    0     0     0
blue  196  183  158  107    0  205  255  255  238   205   139
```

> barplot(1:5,col=heat.colors(5))
> heat.colors(5)
[1] "#FF0000FF" "#FF5500FF" "#FFAA00FF" "#FFFF00FF" "#FFFF80FF"

> barplot(1:5,col=topo.colors(5))
> topo.colors(5)
[1] "#4C00FFFF" "#004CFFFF" "#00E5FFFF" "#00FF4DFF" "#FFFF00FF"
Colors

The default color sets that R provides can be somewhat limiting -- With the RColorBrewer package, **we can define a pallet of colors**, a range of colors for various purposes; **sequential palettes** (for ordered data that move from low to high, say) or perhaps a **qualitative palette** that is used for unordered categories and the colors are chosen to be easily differentiable

The sequential palettes have hues like shades of Blues or Greys or Oranges, or faces between two colors, RdPu (red to purple) or YIGn (yellow to green) ...
# load the RColorBrewer library...
> library(RColorBrewer)

# and have a look at the palettes available when you want, say, six different colors

> display.brewer.all(6)
> display.brewer.all(10)

# we can then select one we like...
> pal = brewer.pal(6,"Blues")

# notice that it’s stored as hexadecimal RGB values -- the same format as returned
# by the function rgb()

> pal
# [1] "#EFF3FF" "#C6DBEF" "#9ECAE1" "#6BAED6" "#3182BD" "#08519C"

# finally, we can use them in a plot...
Polygons

We have seen how to add lines and points to a plot, and we’ve seen that the arguments controlling their size and color and symbol or line type are vectorized; so that if we provide a vector of values R will (re)cycle through them.

The same is true for polygons; the maps package, for example, deals in polygons -- You can see where one polygon ends and the next one starts by looking for a row of NA coordinates.

If you provide graphical arguments like line type or line width or color to the function polygon, it will also (re)cycle through them, assigning one for each polygon.
The geometry of a plot

The basic plot routine can mark the data with points (a scatterplot, `type="p"`), or lines (`type="l"`) or both (`type="b"`) or vertical bars (`type="h"` for heights) or (`type="s"` for stairs) or, well, do nothing at all (`type="n"`) -- See the help entry for `plot` for a complete list.

Each of the high-level plotting functions (`plot`, `hist`, `barplot`, etc.) is highly customizable (allowing you to position the tick marks and labels as you like, for example) and we can annotate them as we like, adding `points()` and `lines()` and `polygons()` and `rect()` and `text()` to a plot.

These high-level plots share a basic geometry and every page is divided into three main regions: the **outer margins**, the **current figure region**, and the **current plot region**.
The geometry of a plot

R allows you to have more than one plot on a page; old school S allowed you to carve a page into a grid of plots specified by a graphical parameters `mfrow` or `mfcol` (the former adds plots by row, the latter by column -- we’ll talk about how to do this in a second) -- On the next page we give an example with `mfrow=c(3,2)`, meaning 3 rows and 2 columns

An alternative introduced in R is to use a command `layout()` -- this function provides all the functionality of the S method, and allows for arbitrarily sized grid regions...
> lmat <- matrix(1:4,ncol=2)
> lmat

layout(lmat)

# show the position of all four plots
layout.show(4)

# same as mfrow=c(3,2)

lmat <- matrix(1:6,ncol=2,byrow=T)
layout(lmat)
layout.show(6)

layout(matrix(c(1,2),ncol=1),height=c(2,1))
layout.show(2)

# a zero means a non-figure region
> layout(rbind(c(1,0,2),c(0,3,0),c(4,0,5)),heights=c(2,1,2))
> layout.show(5)
The state of a traditional graphics device

In the previous example, we alluded to the parameters `mfrow` and `mcol`; these graphical parameters are part of a larger structure that’s used to describe the state of a graphics device.

As you create a plot, you either implicitly or explicitly refer to its state; the state of a traditional graphics device is represented by a number of parameters that record the size of the display, default settings for plotting characters and so on -- We can have access to these parameters through the command `par()`

The manual entry for this command lists all of the variables that represent the current graphics state; similarly, you can use this command to make a change.

For example, by default R assumes you want a single figure on each page; it is possible to instead create a grid of figure regions on a single page...
Set or Query Graphical Parameters

Description:

'par' can be used to set or query graphical parameters. Parameters can be set by specifying them as arguments to 'par' in 'tag = value' form, or by passing them as a list of tagged values.

Usage:

```r
par(..., no.readonly = FALSE)

<highlevel plot> (...) <tag> = <value>)
```

Arguments:

...: arguments in 'tag = value' form, or a list of tagged values. The tags must come from the graphical parameters described below.

no.readonly: logical; if 'TRUE' and there are no other arguments, only
# par() returns a list of the parameter values that represent the current state of
# the graphics system

> opar <- par()

> length(opar)
[1] 70

> opar[1:5]
$xlog
[1] FALSE

$ylog
[1] FALSE

$adj
[1] 0.5

$ann
[1] TRUE

$ask
[1] FALSE

# mfrow is a vector that specifies a grid of figures, the first is nrow the second ncol
> opar$mfrow
[1] 1 1

# specify a 2x1 grid of figures...
> par(mfrow=c(2,1))
The state of a traditional graphics device

In general, you can use \texttt{par()} to \textbf{adjust the defaults for a particular graphics device} -- For example, suppose you prefer your scatterplots with filled circles (symbol number 20); we can reset the default with the command

\begin{verbatim}
# old value
> par()$pch
[1] 1

# assign a new value
> par(pch=20)

Again, this call applies to the current graphics device
\end{verbatim}
Deconstructing a plot

We have described a process whereby we start with a standard plot and either customize it or annotate it in some way; as we have seen before, you can completely deconstruct a plot and invoke lower-level primitives...

```r
# bottom up
> u <- rnorm(10)
> v <- rnorm(10)

# a new window...
> plot.new()

# ... adjust the coordinate system to our data ranges
> plot.window(range(u),range(v))

# ... make the default scatterplot
> plot.xy(u,v,type="p")
> box()                  # add a box
> axis(1)                # and the side 1 axis (bottom)
> axis(2)                # and the side 2 axis (left)
```

Murrell's text on R Graphics is an excellent resource (his Chapter 3 goes over all of the detail in how one interacts with the traditional graphics system)
Another graphics system

While we have been using the base graphics package so far, a fair bit of work has gone into updating this model

The grid package by Paul Murrell is a good example; it uses the idea of a viewport that lets you place small plots anywhere on the page -- This package has also been used to construct the lattice package we will see later in the quarter

The example here involves ridership on the NYC subway; about 6 months ago I came across an overlay of sparkline plots maps of NYC -- Here is an example tracking ridership on the L-train
the I train
# ridership data for L-train stops in Brooklyn

```r
> ride <- read.csv(url("http://www.stat.ucla.edu/~cocteau/ride.csv"))
```

# each column is a year, each row is a different station -- the rows are
# arranged according to their distance to manhattan (Bedford being closest,
# Rockaway Pky farthest)

```r
> ride$name

[1] Bedford Ave  Lorimer St   Graham Ave   Grand St     Montrose Ave
[6] Morgan Ave   Jefferson St DeKalb Ave   Myrtle Ave   Halsey St
[16] Livonia Ave  New Lots Ave E 105th St   Rockaway Pky
19 Levels: Atlantic Ave Bedford Ave Broadway Jct Bushwick Ave ... Wilson Ave
```

```r
> r = as.matrix(ride[, -1])
```

# matplot will create 19 time series on the same plot...

```r
> matplot(1928:2005, t(ride[1:6,]), type="b", xlab="year", ylab="number of riders")
```

# notice ridership has picked up more slowly deeper into Brooklyn

```r
> lstations <- read.csv(url("http://www.stat.ucla.edu/~cocteau/lstations.csv"))
```

```r
> head(lstations)

   name         x        y
1  Bedford Ave -73.95654 40.71708
2    Lorimer St -73.94937 40.71407
3   Graham Ave -73.94440 40.71453
4      Grand St -73.94069 40.71195
5 Montrose Ave -73.93985 40.70765
6  Morgan Ave -73.93188 40.70634
```

```r
> ride2 <- ride[, -1]
```
The Grid Graphics System

Like the traditional system, we have the concept of a graphical device; output appears on the device obscuring what came before (the painter’s model) -- There are also primitive functions that draw lines and points and place text on a graph.

In Grid, there are no pre-defined regions for graphical output; instead, there is a fairly powerful way to specify output regions using the notion of a viewport -- A viewport is a drawing context that consists of both a geometric context (a coordinate system for locating and sizing output, say) and a graphical context (the state as defined by graphical parameters specified in `gpar`).

Viewports are just another kind of object in R; we instantiate them and then “push” them onto a plot -- When we push a viewport, it becomes the active drawing region; this results in a chain of viewports that you can navigate.
> library(grid)

# open a new page... the setting for a new plot
> grid.newpage()

# open a new viewport -- an “old school” plotting window; and then set
# the ranges of the axes

> pvp = plotViewport()
> dvp = dataViewport(range(lstations$x),range(lstations$y))

# add pvp to the viewport tree and then add dvp to it; dvp is now the current
# drawing context

> pushViewport(pvp)
> pushViewport(dvp)

# primitive drawing routines to trace out the path of the L-train and label
# the stations; again, these are going to dvp

> grid.lines(lstations$x,lstations$y, default.units="native",
            gp=gpar(col=gray(0.95),lwd=3))

> grid.text(lstations$name,lstations$x-0.007,lstations$y,
          default.units="native",just="right",
          gp=gpar(cex=0.7,col=gray(0.7)))
# finally, make the small sparkline plots... these are each a separate viewport
# positioned within dvp...

> for(i in 1:nrow(ride)){

    spvp = viewport(x=lstations$x[i], y=lstations$y[i], just="bottom",
                    default.units="native", width=0.01, height=0.005)
    sdvp = dataViewport(c(1, ncol(ride2)), range(ride2))
    pushViewport(spvp)
    pushViewport(sdvp)

    # the sparkline
    grid.lines(1:ncol(ride2), ride2[i,], default.units="native")

    # the lower horizontal line
    grid.lines(c(1, ncol(ride2)), rep(min(ride2, 2)), default.units="native",
               gp=gpar(col=gray(0.8)))

    # the red dot
    grid.points(ncol(ride2), ride2[i, ncol(ride2)], default.units="native",
                gp=gpar(col="red", cex=0.5), pch=19)

    # move back to the dvp view port by removing the two we added...
    popViewport(2)
}
Where to?

Murrell’s text on R Graphics includes an extensive discussion of the Grid package in Chapter 5

One of the graphics packages in R, lattice, is written using the Grid system -- We will see lattice a little later in the quarter...
Next time

We’ll talk about ggp1ot2, a package developed by Hadley Wickham at Rice University -- It’s built around the ideas from Lee Wilkinson’s Grammar of Graphics text

We’ll also introduce lattice, a package that was originally designed for S by Bill Cleveland when he was at Bell Labs and has been rewritten for R

We’ll also cover polygons and spatial data objects in R -- We’ll experiment with a little mapped data and show how to make simple color overlays (and see a legend, a topic we should have covered today)