Lecture 15: Functions and data-directed programming (ii)
Last time

We spent some time with databases and examined how we could access relational databases from within R

Our primary example came from the Reality Mining project at MIT -- We also made extensive use of SQLite
Today

We are going to look at the details of how functions are evaluated in R -- We'll talk about its scoping rules and introduce a class of objects called environments.

We'll then return to data-directed programming with S3 classes and show a simple example or two.

But first...
Functional programming

On the next few slides we review the idea of functional programming, as distinct from object-oriented programming, say

Ultimately, it will help us understand R a little better (I hope) -- But for the moment, let’s not try to make the connection with R and instead examine what this programming paradigm has to offer...
Functional programming

The functional approach to programming very different from the models or paradigms we have seen so far with imperative languages and object-oriented languages.

Functional programming takes inspiration from the work of Princeton mathematician Alonzo Church on the lambda calculus.

While the lambda calculus is a bit off topic for today, suffice it to say that in a functional model, we equate a program (a description of a specific computation) with a mathematical function; that is, a rule that associates to each set of inputs a unique output, or, in classical notation $y = f(x)$. 
Functional programming

With a programming language, we distinguish **function definition** and **function application**; the former tells us how a function is to be computed using **formal parameters**, while the latter is a call to a declared function that uses **actual values**.

In mathematics this distinction is often a little clouded; when we write the definition \( f(x) = x \times x \), we think of \( x \) as a formal parameter; but when we want to find \( x \) such that \( f(x) = 2 \), we think of \( x \) as a variable.

In mathematics, **variables always stand for actual values** -- While in the programming languages we have seen so far, variables can be both memory locations as well as values.

For example, in mathematical notation \( x = x + 1 \) doesn’t make much sense (whereas in Python or R we would mean that we increment \( x \)).
Functional programming

With functional programming, you don’t have the notion of a variable, except as a name for a value -- The idea that a variable is a piece of memory that can be updated is not allowed

With this restriction, we further eliminate (re)assignment as an available operation (by this we mean that we can give a name to a value but we cannot change that value later) -- Functions can be defined without repeatedly modifying some internal “state”

And without reassignment, we eliminate the possibility of a loop -- By its very definition, a loop has to have a control variable that is reassigned as the loop executes
Functional programming

OK so this might sound crazy at first, but it turns out that any computation can be described using function calls alone.

Below we define a simple R function that “reverses” a string, but does so without assignment and without an explicit loop -- Instead, the function uses recursion, in effect maintaining state through the arguments passed to successive calls to itself.

```r
# recall that paste takes two strings and, well, pastes them together using sep as a separator -- in python we'd just use simple addition to join two strings

> flip <- function(x){
    if(x == "") return(x)
    else return(paste(flip(substring(x,2)),substring(x,1,1),sep=""))
}

> flip("i'll be damned!")
[1] "!denmad eb ll'i"
```
Functional programming

It’s worth mentioning a couple other key principles of this coding paradigm -- One direct implication of adopting a mathematical interpretation of functions is that **the only result of applying a function is its return value** (the output)

Put another way, **functions cannot produce “side effects”** (by contrast, recall that in Python we could change the value of mutable objects passed to functions)

In addition, because mathematical functions depend only on the values assigned to their formal parameters (the inputs), **we don’t have to worry about “external influences”** when applying a function (by contrast, recall that Python had a set of rules for searching outside a function definition for names outside the local scope)
Functional programming

Because of these last two properties, **functional programs are easy to test** --
It’s almost a problem in experimental design (a fact that should warm the hearts
of our 201a students!)

It is also felt that **functional programs are easier to debug** because they
don’t depend on a lot of external, possibly ephemeral, conditions that could be
hard to reproduce

Finally, functional programs can be reasoned about mathematically, that is, **you can prove things about your code**, like whether or not two different programs
represent the same function -- this makes it possible to create systems that
analyze your code for correctness and even propose a range of input values for
testing
Functional programming

One final note, to deliver on these promises, a functional programming language must be able to manipulate functions in arbitrary ways -- That is, functions must be general language objects

In particular, **functions must be viewed as values themselves**, which can be computed by other functions and which can also be parameters to functions
Functional programming and R

The view of functional programming on the previous slides is often referred to as “pure” -- Many languages that are functional in style break from this strict doctrine (we’ve seen many times in this class how some of the most powerful tools strike a balance between strict formalism and ease of use)

R, for example, allows for both local and non-local assignments, and some functions have wide-ranging side effects

Still, functions play an important and deep role in R -- The style of programming advocated by Chambers and others makes liberal use of functions for reasons like readability and testing

With that in mind, we’ll now shift gears and consider functions -- Now that you have written a few, we can talk a bit about what goes on “under the hood”...
Function objects

Last time we saw that we **create our own functions** by evaluating an expression of the form

\[
\text{function ( formal arguments ) body}
\]

The resulting object has class `function` and it consists of three pieces:

1. A function’s **formal arguments** are contained in a comma-separated list of names followed optionally by the corresponding default expression (glued together with an ‘=’ which is not to be confused with the assignment operator)

2. The **body of a function** can be any complete R expression, but is typically a sequence of expressions (a block surrounded by `{}`’s)

3. The **environment of a function** is important in determining what objects are visible in a call to this function; when a function is created by evaluating its expression (it’s definition), the current environment is recorded as the function’s environment
# Median Absolute Deviation

# Description:

Compute the median absolute deviation, i.e., the (lo-/hi-) median of the absolute deviations from the median, and (by default) adjust by a factor for asymptotically normal consistency.

> mad
function (x, center = median(x), constant = 1.4826, na.rm = FALSE, low = FALSE, high = FALSE)
{
  if (na.rm)
    x <- x[!is.na(x)]
  n <- length(x)
  constant * if ((low || high) && n%%2 == 0) {
    if (low && high)
      stop("'low' and 'high' cannot be both TRUE")
    n2 <- n%/2 + as.integer(high)
    sort(abs(x - center), partial = n2)[n2]
  }
  else median(abs(x - center))
}
<environment: namespace:stats>
> body(mad)
{
  if (na.rm)
    x <- x[!is.na(x)]
  n <- length(x)
  constant * if ((low || high) && n%%2 == 0) {
    if (low && high)
      stop("'low' and 'high' cannot be both TRUE")
    n2 <- n%/%2 + as.integer(high)
    sort(abs(x - center), partial = n2)[n2]
  }
  else median(abs(x - center))
}

> formals(mad)
$x
$center
median(x)

$constant
[1] 1.4826

$na.rm
[1] FALSE

$low
[1] FALSE

$high
[1] FALSE

> environment(mad)
<environment: namespace:stats>
Aside

When we define a function, it is often the case that users can provide a subset of the arguments -- We might define sensible defaults in the form of actual values or other expressions (in the case of mad, the default for center is median(x))

When we don’t provide a default, its entry will be a special object (everything has to be an object, after all) that represents its missing status; it is an object with content “” that is treated as missing when passed to any function, except missing() itself

missing() a function that returns TRUE or FALSE depending whether its argument is, well, missing (this is often used in R functions so that an author can adapt to the kind of data a user provided)
Function objects

Notice that the environment for `mad` is the namespace for the package `stats` while `paste` and `substring` are associated with the package `base` -- Environments for function objects are important in that they determine **what other objects are visible from within a call to the function**

```
> environment(substring)
<environment: namespace:base>

> environment(paste)
<environment: namespace:base>
```

We will return to this idea in a moment -- Notice, however, if we create a function during an R session, it’s **environment is that of our workspace** (`.GlobalEnv`), the top level global environment
# when we define our own functions on the command line, their environment
# is that of our workspace

> f <- function(x) x^2

> environment(f)
<environment: R_GlobalEnv>

> environment(mad)
<environment: namespace:stats>

> environment(paste)
<environment: namespace:base>

> search()
[1] ".GlobalEnv"   "package:polspline" "package:stats"
[10] "package:base"
Scoping rules

With each language we encounter in this class, we will see that it is essential to be able to refer to objects that we create (or make use of objects and methods that others have created) -- The procedures languages follow for matching names to objects are known as scoping rules

We have seen R’s scoping rules in action already...
# the command search() presents our search path; when we type a name into
# R, it runs along the path, looking in each “environment” (here, the packages you
# have attached) until it finds an object associated with the name you requested

> search()
[1] "\.GlobalEnv"        "package:polspline" "package:stats"
[10] "package:base"

# let’s look for pi

> pi
[1] 3.141593

> find("pi")
[1] "package:base"

# and let’s create our own...

> pi <- 10

> find("pi")
[1] ".GlobalEnv"    "package:base"

> pi
[1] 10

# another way to get the value of pi we want...

> base::pi
[1] 3.141593

> rm(pi)
Scoping rules

Essentially the same problem exists when you are executing the body of a function and \textit{R encounters a name that is not one of the formal arguments and has not been defined in the body of the function itself}

Consider the following...
# first, a simple function that does nothing particularly interesting...
# x and y are the formal arguments, z is a local variable

> test <- function(x,y)
  {
    z <- 3
    return(x*y+z)
  }

> test(2,3)
[1] 9

# now, rewrite the function, but do not define z in the body

> test <- function(x,y)
  {
    return(x*y+z)
  }

> test(2,3)
Error in test(2, 3) : object 'z' not found

# there is no object z for it to find; for example, there’s no z in our workspace

> ls(pattern="^z")
character(0)

# ... but if we make z...

> z <- 5

> test(2,3)
[1] 11

> ls(pattern="^z")
[1] "z"
Scoping rules

In R, the fundamental reference to an object consists of a pair -- The combination of a name (a character string) and an environment (a context) in which the name is evaluated.

An environment is a class of object in R that consists of pairs of names and objects -- It also contains a reference to another environment, its parent and this simple structure creates a search hierarchy that R follows to names to objects.

As you work through an R session, you will encounter a number of environments (well, they will be tapped into existence for you, whether you are directly aware of their presence or not).

Your workspace or the so-called global environment is one example; an environment is created when you evaluate a function (as we will see shortly); and environments associated with packages contain objects exported to the session or, in the package’s namespace, the objects visible to functions in the package.
Environments

Whether we are hunting for \texttt{pi} at the command line, or \texttt{z} within a function body, \texttt{R} uses environments to specify the search process.

The chain of environments for an R session depends on what packages and other environments are attached; now we see that the function \texttt{search()} returns the names of these environments in a search list.
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"

> ev <- parent.env(.GlobalEnv)
> environmentName(ev)
[1] "package:stats"

> class(ev)
[1] "environment"

> ev2 <- parent.env(ev)
> environmentName(ev2)
[1] "package:graphics"

> ev2
<environment: package:graphics>
attr("name")
[1] "package:graphics"
attr("path")

> library(MASS)
> search()
[1] ".GlobalEnv"        "package:MASS"      "package:stats"
[10] "package:base"
> ev <- parent.env(.GlobalEnv); environmentName(ev)
[1] "package:MASS"
Environments

Notice that **loading a package changes the order of your search path** and there might be conflicts with names

Both `mgcv` and `gam` both have a function `gam()`, for example, and the version R will find depends on which of `library(mgcv)` and `library(gam)` was called last -- **It is possible to refer uniquely to the function you want** with the operator `::`, or `mgcv::gam()` and `gam::gam()`

(We will see how to rectify all this when we learn how to create our own packages -- combining functions and data into a form that can be readily shared with others)
Environments and functions

Recall that a function definition consists of three pieces: A function's formal arguments, the expressions that comprise its body, and an environment.

When a function is created, it gets a reference to the environment in which its defining expression was evaluated -- So, if we define a function at the R prompt, it's environment is our workspace or the global environment.
Calls to R functions

So we’re now in a position to talk about how functions are evaluated in R; there are basically three steps

1. The **argument expressions in the call** (the actual arguments) are **matched to the formal arguments** in the function definition

2. A **new environment is created** and the data you provided in the call are **copied to this new workspace** (with defaults for the remaining arguments) -- the parent environment is the **environment of the function object**

3. The **body of the function is evaluated in the new environment** and the **result returned** as the value of the function call
Calls to R functions

Remember, that assignments (like the one mentioned in step 2) create copies of objects -- In this way, we can be sure that whatever we pass to a function is not changed by that function (as opposed to what we will experience in Python)

Also, recall that the expressions you pass as actual arguments are evaluated in this special environment; they are evaluated the first time they are needed (this is so-called lazy evaluation)

This is why we can define center=median(x) meaningfully as a default value in the function definition of mad
Calls to R functions: Scoping

R uses what is commonly referred to as static scoping or “lexical scoping”; the term “lexical” coming from the fact that if you were to sit down with the code, you could see (read) where each variable gets its value (This is in contrast to so-called dynamic scoping, which we won’t really get into this quarter)

When R evaluates the expressions in the function’s body, it uses values associated with names in its evaluation environment -- If the function body references a name not contained in this environment, it starts looking...

The hunt begins with the parent environment of the function call, or rather the environment associated with the function object -- In the simplest case, you create a function in your workspace, and this becomes its parent environment and R will search there for names

If it can’t be found in the function object’s environment, R starts searching, moving up the chain of parent environments we explored earlier
Environments and function calls

Typically, once a value is returned, the environment created to evaluate a function is “garbage collected” -- That is, when there are no more references pointing to it, R will delete it from memory

We can use the process R follows to evaluate functions in interesting ways...
> power <- function(lambda)
  {
    g <- function(x) x^lambda
    return(g)
  }

# the environment associated with square() refers to where it was defined; in
# this case it was defined in a call to power()

> square <- power(2)
> cube <- power(3)

> square(1:5)
[1]  1  4  9 16 25

> cube(1:5)
[1]   1   8  27  64 125

> environment(square)
<environment: 0x16ecb34>

> environment(cube)
<environment: 0x16ec8b0>
Environments and function calls

Here we used **a function to create another function**; in this case, the environment of the new function (**square**) is the evaluation environment of the function we invoked (**power**)

We can see that the two functions **square** and **cube** have different environments by printing them out

The value of **lambda** used by each is stored in its environment -- Just like objects are stored in your own workspace
> square
function(x) x^lambda
<environment: 0x16ecb34>

> cube
function(x) x^lambda
<environment: 0x16ec8b0>

> parent.env(environment(square))
<environment: R_GlobalEnv>

> parent.env(environment(cube))
<environment: R_GlobalEnv>

> objects(environment(square))
[1] "g" "lambda"

> objects(environment(cube))
[1] "g" "lambda"

> get("lambda",env=environment(square))
[1] 2

> get("lambda",env=environment(cube))
[1] 3
> make.cdf <- function(x)
  {
    n <- length(x)
    out <- function(q) return(mean(x<=q))
    return(out)
  }

> x <- rnorm(100)

> mycdf <- make.cdf(x)

> mycdf
function(q) return(mean(x<=q))
<environment: 0x1739e0c>

> mycdf(-1)
[1] 0.16

> mycdf(0.5)
[1] 0.66

> get("n",env=environment(mycdf))
[1] 100
Final word on environments

As an object, we’ve said that an environment is a collection of symbol-value pairs, with an additional piece of information describing its parent.

Aside from these function gymnastics, you can create environments directly and use them a bit like lists -- They are unlike lists in R in two important ways.

First, you can only access the values by name and not by “number” as there is no sense of linear order here -- Second environments and their contents are not copied if they are passed as arguments to functions.

This last point is crucial because it breaks all the rules we have learned so far about side effects and function evaluation -- The mechanism provides for the pass-by-reference semantics we met Python and should be used with great caution.
Data-directed programming

I will use the term “data-directed programming” for the programming structures we will talk about today; in most books on R, this material is found under the heading of “Object-oriented programming”

In “classical” object-oriented programming, we see a division between how data are represented and the operations that can sensibly be performed on them -- This often leads to code that is easier to write and maintain

As its name suggests, object-oriented programming deals in objects, typically physical things that are represented in data by classes and methods or functions are written to manipulate them in various ways
Data-directed v. object-oriented programming

Many object-oriented languages like Python and Java are (as Gentleman describes them) “class-centric”, meaning they classes define objects and are “repositories for the methods that act on” them.

In R, on the other hand, separates the class information from the creation of so-called generic functions and (again, quoting Gentleman) can be thought of as a “function-centric” system.
Data-directed v. object-oriented programming

There are two built-in class and method systems in R (S3 and S4) and a number of others that are available via packages

These constructions were relatively late additions to the language and hence the two versions; S3 evolved out of a significant effort to introduce modeling functions (\texttt{lm, glm, gam, loess}) into the language in the early 1990s, while S4 was, well, a more reliable second attempt in the late 1990s

S3 lacks formal specification of classes, and is really about function dispatch, about generic functions and polymorphism (we will call this data-directed programming); S4 introduces formal class definitions and a complete system for inheritance (we'll return to this after we have seen a bit of Python)
Data-directed programming in R

To clarify things a bit, a class specifies how objects are to be represented in computer code, or, in short, the properties an object must have if it is of a given class -- An object is the instance of one and only one class and we say that objects differ depending on their “state”

New classes can extend old ones in a process known as inheritance; the extension might involve new data, for example, or maybe combining two existing classes -- If class A extends class B, we say that A is a superclass of B and that B is a subclass of A

A method is a function that is invoked depending on the class of one or more of its arguments, a process we have been referring to as dispatch; think back to the way in which plot() functioned differently
We’ll start with the relatively simple S3 class construction; primarily because there’s not much to it and because it is a big part of the modeling tools you have been using in other classes.

There is no formal mechanism for representing instances of a class; they are, however, typically lists, where named elements of the list represent “slots”.

You can access the class of an S3 object with the function `class()`; this can be used to both determine as well as set the class of an object (except for special cases involving implicit classes, this is the same as creating an attribute called `class` with value the string with the class name); finally `is.object()` tests to see if an R object has a `class` attribute.
As a start

Here’s a site I’ve been a little obsessed with -- It advertises the positions of planes, allowing you to track flights, examine all the traffic into or out of a given airport or examine the fleet of a given carrier

From what I can see, the data are not available via an API and so we have to do a little scraping -- For grins, here is what that scraping looks like in R rather than Python with BeautifulSoup (or htmlib5)
Live Flight Tracking.

FlightAware > Live Flight Tracker > V Australia #1

VAU1
(all flights) (photos)
V Australia "V-OZ" Australia

Upload photo
now

Origin
Sydney (YSSY / SYD) - track or info

Destination
Los Angeles Intl (KLAX) - track or info

Other flights between these airports
SY O505 ALLOC GORDO 2421S 16300E NISAS 1900S 17000E 12000 18000W IDEMJ 0500S 17300W DULCE 0700N 16000W TAKIE 1400N 15000W 2100N 14000W 2900N 13000W FOOTS R578 FICKY C1177 SXC LAX

Date
Monday, November 15 2010

Duration
13 hours 43 minutes

Status
Landed 42 minutes ago (track log & graph)

Distance
Direct: 7,502 sm Flown: 7,614 sm

Scheduled 7-day Average Actual/Estimated

Departure
01:15PM AEDT 12:21PM AEDT 01:33PM AEDT

Arrival
07:32AM PST 07:01AM PST 08:17AM PST

Speed
500 kts

Altitude
37,000 feet

At least part of this flight occurs outside of FlightAware's primary service area. Learn more about FlightAware's coverage and service areas here.

© 2010 FlightAware.com
Weather: 15-Nov-2010 04:30AM

© 2010 FlightAware.com
## Vehicle Tracking
Save Money & Time With GPS Tracking Get Your Free Live Demo Online Here

Teletrac.net/FreeLiveDemo

<table>
<thead>
<tr>
<th>Time</th>
<th>Position USA: Eastern</th>
<th>Orientation</th>
<th>Groundspeed KTS</th>
<th>Altitude Feet</th>
<th>Rate</th>
<th>Reporting Facility</th>
</tr>
</thead>
<tbody>
<tr>
<td>05:27AM</td>
<td>8.52 -158.05</td>
<td>52° Northeast</td>
<td>514</td>
<td>35,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>05:41AM</td>
<td>9.68 -156.52</td>
<td>52° Northeast</td>
<td>514</td>
<td>35,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>05:55AM</td>
<td>10.88 -154.92</td>
<td>52° Northeast</td>
<td>522</td>
<td>35,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>05:59AM</td>
<td>11.27 -154.42</td>
<td>57° Northeast</td>
<td>510</td>
<td>35,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>06:08AM</td>
<td>11.98 -153.28</td>
<td>57° Northeast</td>
<td>511</td>
<td>35,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>06:22AM</td>
<td>13.03 -151.58</td>
<td>58° Northeast</td>
<td>525</td>
<td>35,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>06:35AM</td>
<td>14.00 -150.00</td>
<td>51° Northeast</td>
<td>511</td>
<td>35,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>06:36AM</td>
<td>14.12 -149.85</td>
<td>52° Northeast</td>
<td>512</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>06:50AM</td>
<td>15.30 -148.25</td>
<td>53° Northeast</td>
<td>510</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>07:04AM</td>
<td>16.45 -146.67</td>
<td>53° Northeast</td>
<td>509</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>07:32AM</td>
<td>18.75 -143.38</td>
<td>54° Northeast</td>
<td>515</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>07:45AM</td>
<td>19.90 -141.67</td>
<td>55° Northeast</td>
<td>520</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>07:59AM</td>
<td>21.00 -140.00</td>
<td>46° Northeast</td>
<td>495</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>08:13AM</td>
<td>22.42 -138.37</td>
<td>47° Northeast</td>
<td>483</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>08:27AM</td>
<td>23.77 -136.78</td>
<td>48° Northeast</td>
<td>481</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>08:41AM</td>
<td>25.03 -135.22</td>
<td>48° Northeast</td>
<td>477</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>08:55AM</td>
<td>26.25 -133.70</td>
<td>49° Northeast</td>
<td>462</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>09:09AM</td>
<td>27.37 -132.23</td>
<td>50° Northeast</td>
<td>451</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>09:23AM</td>
<td>28.48 -130.72</td>
<td>49° Northeast</td>
<td>466</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>09:29AM</td>
<td>29.00 -130.00</td>
<td>67° East</td>
<td>458</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>09:36AM</td>
<td>29.37 -128.98</td>
<td>68° East</td>
<td>456</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>09:50AM</td>
<td>30.00 -127.13</td>
<td>69° East</td>
<td>456</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>10:04AM</td>
<td>30.62 -125.25</td>
<td>70° East</td>
<td>464</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>10:19AM</td>
<td>31.21 -123.22</td>
<td>66° East</td>
<td>473</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>10:20AM</td>
<td>31.26 -123.08</td>
<td>62° Northeast</td>
<td>461</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>10:21AM</td>
<td>31.32 -122.95</td>
<td>59° Northeast</td>
<td>455</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>10:22AM</td>
<td>31.38 -122.83</td>
<td>61° Northeast</td>
<td>448</td>
<td>37,000</td>
<td></td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>10:23AM</td>
<td>31.45 -122.70</td>
<td>63° Northeast</td>
<td>447</td>
<td>37,000</td>
<td>-60</td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>10:24AM</td>
<td>31.51 -122.56</td>
<td>63° Northeast</td>
<td>452</td>
<td>38,900</td>
<td>-60</td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>10:34AM</td>
<td>31.64 -122.35</td>
<td>63° Northeast</td>
<td>452</td>
<td>38,900</td>
<td>-60</td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>10:35AM</td>
<td>31.70 -122.22</td>
<td>63° Northeast</td>
<td>452</td>
<td>38,900</td>
<td>-60</td>
<td>Oakland Oceanic (ODAPS)</td>
</tr>
<tr>
<td>Time</td>
<td>Position</td>
<td>Orientation</td>
<td>Groundspeed</td>
<td>Altitude</td>
<td>Reporting Facility</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>-------------</td>
<td>-------------</td>
<td>----------</td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>11:14 AM</td>
<td>118.47°</td>
<td>-118.47°</td>
<td>33.87°</td>
<td>229</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:15 AM</td>
<td>118.48°</td>
<td>-118.48°</td>
<td>33.82°</td>
<td>243</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5,800</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1,860</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As a start

The following code makes use of the XML and RCurl packages and does a brutal kind of scrape through the structures in the HTML file -- In this case it’s pretty easy (albeit a total hack)

The XML package uses a language called XPath to specify objects in an XML document -- Here we are after all nodes “//” that have an attribute “@” href, for example

The code is in /data/flights/flight_tracker.R on our homework machine...
grab_flights <- function(where="KLAX"){

  flight_tracks <- list()

  airport <- paste("http://flightaware.com/live/airport/",
                   where,"/departures",sep="")

  x <- htmlParse(getURL(airport))
  hflights <- unlist(x["//@href"])
  flights <- hflights[grep("/live/flight/",hflights)]
  flights <- flights[-grep("random",flights)]

  for(f in flights){

    y <- htmlParse(getURL(paste("http://flightaware.com",f,sep="")))
    htracks <- unlist(y["//@href"])
    tracks <- htracks[grep("/live/flight/.*tracklog$",htracks)]
    z <- htmlParse(getURL(paste("http://flightaware.com/",tracks[1],sep="")))

    log <- xmlToDataFrame(
      z["/tr[@class='smallrow1'] | //tr[@class='smallrow2']"],
      nodes=z["/tr[@class='smallrow1'] | //tr[@class='smallrow2']"],
      colClasses= c("character","character","numeric","character","numeric",
                     "character","character","character","character","character",
                     "numeric","character","numeric","character","character",
                     "character","character","character","character","character")
    
    if(nrow(log)>0){
      log <- log[,c(3,5)]
      names(log) <- c("lat","lon")
      log <- as.matrix(log)
      class(log) <- "flight_track"
      attr(log,"flight") <- gsub(".*([A-Z0-9]+)\,"\"l",f)
      flight_tracks <- c(flight_tracks,list(log))
    }
  }

  return(flight_tracks)
}
```r
> fl <- out[[1]]
> class(fl)
[1] "flight_track"

> methods(class="flight_track")
none found.

> fl
     lat     lon
[1,] 33.93 -118.45
[2,] 33.93 -118.52
attr(,"class")
[1] "flight_track"
attr(,"flight")
[1] "DAL2365"

# let's make a print method... or rather a function called print.flight_track
> print.flight_track <- function(x){

    cat(paste("Flight",attributes(x)$flight,"is currently at",
                   x[nrow(x),1],"latitude and",x[nrow(x),2],"longitude\n"))

}

# and try it out!
> fl
Flight DAL2365 is currently at 33.93 latitude and -118.52 longitude
```
# now make a new kind of method...

> flight_name <- function(x,...) UseMethod("flight_name")
> flight_name.flight_track <- function(x) attributes(x)$flight

> flight_name(fl)
[1] "DAL2365"

> print.flight_track <- function(x){

    cat(paste("Flight",flight_name(x),"is currently at",
            x[nrow(x),1],"latitude and",x[nrow(x),2],"longitude\n"))
}

> fl
Flight DAL2365 is currently at 33.93 latitude and -118.52 longitude

# and now we’ve got two methods!

> methods(class="flight_track")
[1] flight_name.flight_track print.flight_track
summary.flight_track <- function(x, rad_of_earth=3963){

  point_distance <- function(lat1,lon1,lat2,lon2,r){

    # Calculates the distance between two lon/lat points. Assumes
    # degrees east and north are positive values. The radius of the
    # earth is given as 3,963 miles

    cos_between_points <-acos(cos(lat1*pi/180)*
                            cos(lat2*pi/180)*cos((lon2-lon1)*pi/180)+
                            sin(lat1*pi/180)*sin(lat2*pi/180))

    return(r*cos_between_points)
  }

  trip <- 0

  if(nrow(x) > 1){

    for(i in 2:nrow(x)){

      trip <- trip +
      point_distance(x[i,1],x[i,2],x[i-1,1],x[i-1,2],rad_of_earth)
    }

  }

  paste("Flight",flight_name(x),"has traveled",round(trip,2),"miles.")
}
> sapply(out, summary)

[1] "Flight DAL2365 has traveled 4.02 miles."
[2] "Flight UAL81 has traveled 4.59 miles."
[3] "Flight EJA931 has traveled 17.51 miles."
[4] "Flight SKW434L has traveled 30.7 miles."
[5] "Flight SKW692F has traveled 42.14 miles."
[7] "Flight SKW445M has traveled 54.67 miles."
[8] "Flight COA1703 has traveled 48.22 miles."
[9] "Flight SKW112B has traveled 72.64 miles."
[10] "Flight DAL1197 has traveled 87.43 miles."
[12] "Flight EGF3190 has traveled 90.18 miles."
[13] "Flight UAL324 has traveled 93.08 miles."
[14] "Flight SWA653 has traveled 115.37 miles."
[15] "Flight SKW6246 has traveled 55.01 miles."
[16] "Flight UAL558 has traveled 126.8 miles."
[17] "Flight AAL244 has traveled 142.17 miles."

> lines.flight_track <- function(x, ...) lines(x[, 2:1], ...)
> plot.flight_track <- function(x, database = "usa", regions = ".", ...){

    map(database, regions)
    lines(x, ...) # uses default method for lines, passes color
}

> plot(out[[3]], col = 5, database = "state", regions = "california")
> for(i in 1:length(out)) lines(out[[i]][, 2:1], col = 5)
As a start

It seems reasonable that we might want to distinguish between flights that are in progress and those that have finished -- We can create a second class, say, `complete_flight_track` that “extends” the `flight_track` class

Programmatically, this kind of hierarchy is accomplished through the class attribute of an object -- The class can in fact be a vector of class names...
> flended <- out[[17]]

# let's assume this flight has landed...
> class(flended) = c("completed_flight_track", "flight_track")

# and compare it to one that is still "in flight"

> fl <- out[[15]]

# check inheritance...

> inherits(fl,"flight_track")
[1] TRUE
> inherits(flended,"flight_track")
[1] TRUE

> inherits(fl,"completed_flight_track")
[1] FALSE
> inherits(flended,"completed_flight_track")
[1] TRUE
S3 classes and methods

In the previous example, we assigned a vector of classes to the object `flended`; this is how S3 structures its classes.

Specifically, it says that `flended` inherits its properties from both `completed_flight_track` and `flight_track`; but we think of `completed_flight_track` as “the” class of the object.

The reason for this structure will be clear when we talk about methods.
Frailties

Really, the couple of lines two slides back is all there is to instantiating an object of a given class; notice that **there is no checking of slots to guarantee that the object has everything it might need** to be what it claims to be.

In addition, there are no built-in constructors for S3 objects, as there were for objects in Python -- Typically you create a function (named as you like) that returns objects of the given class (that at some points assigns the `class` attribute appropriately).

With such a low barrier to entry, there is a price to pay when it comes to authoring methods in S3 -- Most of methods have to do a lot of checking.
Generic functions

As we have seen, the S3 class and method system is designed around the concept of a generic function; a generic function has different behaviors depending on the class of one or more of its arguments (this is known as polymorphism).

Generics perform a kind of method “dispatch” that, in turn, selects the appropriate method to be called; we’ve seen a number of S3 generics before...
# spotting a generic function...

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

> summary
function (object, ...)
UseMethod("summary")
<environment: namespace:base>

> residuals
function (object, ...)
UseMethod("residuals")
<environment: namespace:stats>

> fitted
function (object, ...)
UseMethod("fitted")
<environment: namespace:stats>
# spotting a generic... plot calls UseMethod providing the lead argument
# is not a function...

```r
> plot
function (x, y, ...)
{
  if (is.function(x) && is.null(attr(x, "class"))) {
    if (missing(y))
      y <- NULL
  hasylab <- function(...) !all(is.na(pmatch(names(list(...)),
          "ylab")))
  if (hasylab(...))
    plot.function(x, y, ...)
  else plot.function(x, y, ylab = paste(deparse(substitute(x)),
          "(x)")(x), ...)
  }
  else UseMethod("plot")
}
<environment: namespace:graphics>
```

# we can see what happens if we feed plot() a function...

```r
> plot(sqrt)
```
# which means there’s an initial screen in plot() for functions, leading
# to a call of plot.function -- a function that’s a little hard to find... the function is
# in the package graphics, but its name is not “exported” for users to call directly

> plot.function
Error: object 'plot.function' not found

> find(plot.function)
Error in stopifnot(is.character(what)) : object 'plot.function' not found

> getAnywhere("plot.function")   # this function returns even non-visible functions

A single object matching ‘plot.function’ was found
It was found in the following places
  namespace:graphics
with value

  function (x, y = 0, to = 1, from = y, xlim = NULL, ...)
  {
    if (is.null(xlim)) {
      if (is.null(from))
        from <- 0
    }
    else {
      if (is.null(from))
        from <- xlim[1L]
      if (missing(to))
        to <- xlim[2L]
    }
    curve(x, from, to, xlim = xlim, ...)
  }
<environment: namespace:graphics>
S3 classes and methods

The function `UseMethod` dispatches on the class of the object returned by `class()`; methods are simply ordinary functions that are identified by a special naming convention.

Specifically, methods are given names that are concatenations of the name of the generic method and the name of the class that they are intended to apply to, separated by a “.”

You can list the methods associated with a particular generic with a call to the function `methods()`.
> methods(class="factor")

[1] [.factor             [[.factor            <<-.factor
[4] all.equal.factor     as.character.factor  as.data.frame.factor
[7] as.Date.factor       as.list.factor       as.POSIXlt.factor
[10] as.vector.factor     format.factor        is.na<-.factor
[16] Ops.factor           plot.factor*         print.factor
[19] relevel.factor*      relist.factor*       reorder.factor*
[22] rep.factor           summary.factor       Summary.factor
[25] xtfrm.factor

Non-visible functions are asterisked

> methods(class="lm")

[1] add1.lm*           alias.lm*           anova.lm
[5] confint.lm*        cooks.distance.lm* deviance.lm*        dffits.lm*
[9] dfbetas.lm*        drop1.lm*          dummy.coef.lm*      effects.lm*
[13] extractAIC.lm*     family.lm*         formula.lm*         hatvalues.lm
[17] influence.lm*      kappa.lm           labels.lm*          logLik.lm*
[21] model.frame.lm     model.matrix.lm    plot.lm              predict.lm
[25] print.lm           proj.lm*           residuals.lm         rstandard.lm
[29] rstudent.lm        simulate.lm*       summary.lm
variable.names.lm*
[33] vcov.lm*

Non-visible functions are asterisked
> methods("summary")

[1] summary.aov summary.aovlist summary.connection
[4] summary.data.frame summary.Date summary.default
[7] summary.ecdf* summary.factor summary.glm
[10] summary.infl summary.lm summary.loess*
[16] summary.nls* summary.packageStatus* summary.POSIXct
[19] summary.POSIXlt summary.ppr* summary.prcomp*
[22] summary.princomp* summary.stepfun summary.stl*
[25] summary.table summary.tukeysmooth*

Non-visible functions are asterisked

> methods("residuals")

[1] residuals.default residuals.glm residuals.HoltWinters*
[4] residuals.isoreg* residuals.lm residuals.nls*
[7] residuals.smooth.spline* residuals.tukeyline*

Non-visible functions are asterisked

> methods("AIC")

[1] AIC.default* AIC.logLik*

Non-visible functions are asterisked
> methods("print")
[1] print.acf*  [50] print.DLLInfoList
text                   [51] print.DLLRegisteredRoutines
[2] print.anova          [52] print.dummy_coef*
[3] print.aov*          [53] print.dummy_coef_list*
[4] print.aovlist*      [54] print.ecdf*
[7] print.arima0*       [57] print.family
[8] print.AsIs           [58] print.formula
[10] print.browseVignettes* [60] print.getAnywhere*
[13] print.check_demo_index* [63] print.help_files_with_topic*
[14] print.check_dotInternal* [64] print.hexmode
[15] print.check_make_vars* [65] print.McMllMesters*
[16] print.check_package_code_syntax* [66] print.haease*
[17] print.check_package_datasets* [67] print.h.test*
[18] print.check_packagedepends* [68] print.inf*
[19] print.check_package_description* [69] print.integrate
[20] print.check_package_description_encoding* [70] print.isoreg*
[21] print.check_package_license* [71] print.kmeans*
[22] print.check_packages_used* [72] print.Latex*
[23] print.check_Rd_files_in_Rd_db* [73] print.libraryIQR
[24] print.check_Rd_xrefs* [74] print.listof
[25] print.check_T_and_F* [75] print.lm
[26] print.check_vignette_index* [76] print.loadings*
[27] print.checkDocFiles* [77] print.loess*
[28] print.checkDocStyle* [78] print.logLik
[29] print.checkFF* [79] print.la_strF*
[30] print.checkReplaceFuns* [80] print.medpolish*
[31] print.checkS3methods* [81] print.MethodsFunction*
[32] print.checkPin* [82] print.mtable*
[33] print.checkVignettes* [83] print.NativeRoutineList
[34] print.citation* [84] print.nls*
[35] print.citationList* [85] print.noquote
[36] print.codoc* [86] print.numeric_version
[37] print.codocClasses* [87] print.object_size*
[38] print.codocData* [88] print.objectmode
[39] print.colorClasses* [89] print.packageDescription*
[40] print.condition* [90] print.packageInfo
[41] print.connection* [91] print.packageIQR*
[42] print.data.frame* [92] print.packagedstatus*
[43] print.Date* [93] print.pairwise.htest*
[44] print.default* [94] print.POSIXct
[45] print.dendrogram* [95] print.POSIXlt
[46] print.density* [96] print.power.htest*
[47] print.dftime* [97] print.ppr*
[48] print.dist* [98] print.prcomp*
[49] print.DLLInfo* [99] print.princomp*
[100] print.proc_time* [101] print.prcomp*
[102] print.recordedplot* [103] print.numeric_version
[104] print.RGBcolorConverter* [105] print.rle
[106] print.roman* [107] print.getAnywhere*
[108] print.simple.list* [109] print.hclust*
[110] print.sessionInfo* [111] print.srcfile
[112] print.srcref* [113] print.stepfun*
[114] print.subdir_tests* [115] print.StructTS*
[116] print.summary.aov*
Non-visible functions are asterisked
[117] print.summary.aovlist*
[118] print.summary.ecdf*
[119] print.summary.ecdf*
[120] print.summary.glm*
[121] print.summary.lm*
[122] print.summary.loess*
[123] print.summary.manova*
[124] print.summary.mle*
[125] print.summary.ppr*
[126] print.summary.prcomp*
[127] print.summary.princomp*
[128] print.summary.tables*
> methods("["]
[1] [.acf* [.AsIs [.data.frame
[4] [.Date [.difftime [.factor
[7] [.formula* [.getAnywhere* [.hexmode
[10] [.listof [.noquote [.numeric_version
[13] [.octmode [.POSIXct [.POSIXlt
[16] [.roman* [.simple.list [.terms*
[19] [.ts* [.tskernel* [.XMLInternalDocument*
[22] [.XMLInternalNode* [.XMLNode*

Non-visible functions are asterisked

> methods("[[")
[1] [[.data.frame [[.Date [[.dendrogram*
[4] [[.factor [[.numeric_version [[.POSIXct
[7] [[.tclArray* [[.XMLDocumentContent* [[.XMLHashTreeNode*
[10] [[.XMLInternalDocument* [[.XMLInternalNode* [[.XMLNode*

Non-visible functions are asterisked

> methods("[<-")
[1] [<-.CURLOptions* [<-.data.frame [<-.Date [<-.factor
[5] [<-.POSIXct [<-.POSIXLt [<-.ts* [<-.XMLNode*

Non-visible functions are asterisked
Generics are usually **simple functions with just two arguments**, one named \texttt{x} and one “...”; the first argument is typically used to trigger method dispatch (recall that “...” is the way we pass arguments to a function that don’t have to be named explicitly but can be used in the body of the function)

If you don’t include the catch-all “...”, then no method can have a formal argument that is not also a formal argument of the generic; it’s good practice to have all methods include the same formal arguments as the generic and for generics to include the “...” so methods added later can use other arguments

The downside, of course, is that if you mistype the name of a formal argument it will be swept up into the “...” without a warning
S3 classes and methods

When the class attribute of an object contains **more than one string**, **R will run down the list in order**, looking for `generic.classname` for each entry, picking the first it finds; if it can’t find one it will call the associated default method (if it exists, otherwise it produces an error)

For example, the lists on the previous slide all include defaults (`summary.default`, `residuals.default` and `AIC.default`)

Notice that as R runs through the class vector, it is really **running up the inheritance tree**, checking super-classes for an appropriate method -- Sometimes we want to make **explicit use of a method from a super-class**; with S3 we can achieve this kind of action by a call to `NextMethod()`
> flight_name <- function(x,...) UseMethod("flight_name")

> flight_name.flight_track <- function(x) attributes(x)$flight

> flight_name.completed_flight_track <- function(x){
    nm <- NextMethod()
    paste(nm,"(completed)")
}

> flight_name.default <- function(x) cat("please provide a flight track\n")

> class(fl)
[1] "flight_track"
> flight_name(fl)
[1] "SKW6246"

> class(flended)
[1] "completed_flight_track" "flight_track"
> flight_name(flended)
[1] "AAL244 (completed)"

> flight_name(runif(100))
please provide a flight track
Aside: Implicit classes

As we said before, the **earliest versions of the S language were developed prior to widespread adoption of object-oriented programming** principles; as a result, some of the basic classes in R do not make use of the `class` attribute and are referred to as **implicit classes**

For example, functions are implicitly of class `function`, while matrices and arrays are implicitly of classes `matrix` and `array`, respectively.

As a result, `is.object()` will return FALSE when applied to objects having an implicit class; `UseMethod` dispatches, however, on even implicit classes (depending only on the result of a call to `class()`).
Case study: Semiconductor manufacturing

This example helps illustrate how the organization of computations can both ease the burden of simply looking at data and can in fact cascade producing new avenues for methodology and analysis.

Caution: This example is really really really old now; I bring it up only because it nicely illustrates my points and what can be achieved with simple S3 classes.
Case study: Semiconductor manufacturing

Fabrication requires **hundreds of processing steps** and can last up to **6 weeks**; hundreds of IC’s are fabricated simultaneously on a wafer; wafers are processed in lots.

Heavy competition reduces the time between design and manufacture; **lots of experimentation is done do identify design/manufacturing issues** and improve yields.
Semiconductor manufacturing

At the end of the line, wafers are subjected to a series of probe tests - Testing of chips is done in a sequential fashion; each chip is marked with the first test it fails, or is labeled good.

When we first started working with the engineers, most of the plotting was done with ASCII glyphs; different letters represented different failure modes (ultimately this was our print method)

Therefore, our first task was to create some kind of plotting mechanism to make sense of the data.
Semiconductor manufacturing

Here is a simple example of our plotting method in action; colored squares represent failed chips, white squares are good chips

```r
# file with 25 wafer lot
> w = scan.wafers(file)

# double bracket gives single wafer
> plot(w[[1]])

# makes lower row
> barplot(w[[1]])
```
Semiconductor manufacturing

Technically, a wafer object is just a matrix; each row represents a set of probe test results (a lot of 25 wafers each with 300 chips would be a 25x300 matrix)

There are extra slots that record the kind of device, the failure modes, and the geometry (map) that binds column to x-y location
Semiconductor manufacturing

Subsetting with “[]” returns separate wafers

\[ w[ [ \text{yield}(w) < 0.5 ] ] \]
\[ w[[1:5]] \]

Subsetting with “[” works within the map (the chip layout)

\[ w[1:10,1:10] \]
\[ w[\text{edge}(w,2)] \]
Semiconductor manufacturing

The percentage of good chips on a wafer is referred to as its yield; engineers often divide defects into those attributable to special and random causes.
Process problems leave behind **telltale spatial signatures** in inline and post-production data; these patterns can vary both within and between lots.
Semiconductor manufacturing

The manufacturing process is **bathing in complex data**; from thickness measurements to particle scans to various voltage tests taken on some or all of the wafers in some or all of the lots.
Semiconductor manufacturing

Our ultimate goal was **quality and process improvement**; if we can track patterns back to manufacturing steps quickly, we can **correct developing problems**

With so many kinds of data, this exercise requires being able to manipulate these various structured data sources easily and in some sense “correlate” them to each other and to process information
Creating a binary map

> w1 = scan.wafers(file)
> w = as.numeric(w1)
> plot(mean(w))

Defining arithmetic operations meant that we could create spatial averages of groups of wafers

\[
\frac{1}{n} \left( \text{wafer}_1 + \text{wafer}_2 + \cdots + \text{wafer}_n \right)
\]
Probe data

Sometimes a simple mean (average across the lot) can determine problems that hit all the wafers in a lot
Probe data

Often, experiments are run to determine optimal settings for various steps in the process; typically, analysis is performed on wafer yields.

Here we have two factors (coating thickness and diffusion time) with an unequal number of observations.
Probe data

Here's a common display of these data; what recipe do you recommend?
Probe data

We would then follow this up with a linear model to **quantify the impact of the processing strategies on yield**; notice that `fit` is an object of class `lm` and the appropriate summary method is being invoked.

```r
> fit <- lm( yield ( w ) ~ diffusion * coating )
> summary( fit )

            Coefficient Value Std.Err. T-val
(Intercept)       0.269   0.020    13.76
coating           0.116   0.028     4.23
diffusion.L      -0.225   0.032    -6.87
diffusion.Q       0.003   0.034      0.09
coating diffusion.L  0.040   0.046     0.86
coating diffusion.Q-0.087   0.050   -1.76
```
Probe data

Now, let’s use our wafer classes; the idea is that these data are inherently spatial and that we can hopefully glean more information by not reducing them to yields.
Probe data

We can add spatial information by using the **average wafer maps as plotting characters**; this is a call to

```r
> plot(levs,ylds)  # then add lines()
> points(w,at=cbind(levs,ylds))
```

![Graph showing the mean of yield (w) vs diffusion time with different coatings (thin and thick)]
Probe data

And what about our linear model?

If we are trying to determine spatial effects, how might we generalize the fit?

What kind of display suggests itself?
Probe data

To work with these data as easily as the yields vectors, we might envision something like this

```r
> fit = lm(smooth(w) ~ coating*diffusion)
> out = summary(fit)
> plot(out)
```
<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
<th>Std.Err.</th>
<th>T-val</th>
<th>p-val</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Intercept)</td>
<td>-0.35</td>
<td>0.44</td>
<td>0.013</td>
<td>0.100</td>
</tr>
<tr>
<td>coating</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>diffusion.L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>diffusion.Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coatingdiffusion.L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coatingdiffusion.Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Residual degrees of freedom: 32

Root MSE: 0.077
R-squared: 0.240
Probe data

Ok, so that was an extreme example, but being able to manipulate wafers simply leads to some immediate spatial updates of common plots

> boxplot(w)
Probe data

Because wafers are fancy matrices, we can also use some of R's built-in tools for data analysis -- for example, given our interest in spatial patterning and identifying groups of wafers exhibiting similar patterns, clustering seems like an obvious tool...

Let's see how we can extend hierarchical clustering...
Sample lot exhibiting spatial patterning

Lot composite
My point?

Statistical computing is not just about creating new methodology; we also have to consider how to structure data and enable a natural kind of exploratory analysis.

The point of this quarter of Statistics 202 is that you learn the basics of the languages you will use regularly so that you are never left just imagining various analyses; you have the background to implement and do so effectively.
S4 classes and methods

As we have seen, the S3 class mechanism is extremely loose; it really is a way to organize computations around characteristics of objects.

A few years after S3 classes were established through the modeling language (linear models, generalized linear models, generalized additive models, tree methods), John Chambers developed a more sophisticated class mechanism for R.

S4 classes have well-defined structures, include methods for initialization, and the dispatch mechanism has been greatly extended... we will return to these ideas a little later in the quarter.