Lecture 11: I want to wake up in the city that never sleeps
Lecture 11: The city never sleeps, better slip you an Ambien
Last time

We talked a bit about sharing code, version control systems and software licenses -- I realize now that some of the version control lecture was not well motivated and we’ll fix that up in Lecture 12

We also had a very long discussion of your homework -- We’ll return to that today!
Today

We want to present some of the basic (and oldest) objects in R, vectors and matrices -- We have a lot of slides that go over how we create vectors and matrices and how we manipulate them (compute with them, subset them, and so on)

We will then have a brief introduction to S3 classes in R and examine how objects can help us with our Shazam homework -- Really our goal today is to get you more comfortable with Shazam!
Beyond the Fringe Lecture Series
Richard Weinstein Lecture

Monday
November 1 | 6:30 pm

Frank Gehry
Architect, Los Angeles

Raised in Toronto, Canada, Frank Gehry moved with his family to Los Angeles in 1947. Mr. Gehry received his Bachelor of Architecture degree from the University of Southern California in 1954, and he studied City Planning at the Harvard University Graduate School of Design. In subsequent years, Mr. Gehry has built an architectural career that has spanned five decades and produced public and private buildings in America, Europe and Asia. His work has earned Mr. Gehry several of the most significant awards in the architectural field, including the Pritzker Prize. Notable projects include the Guggenheim Museum in Bilbao, Spain and the Walt Disney Concert Hall in Los Angeles, California.

The Weinstein Lecture will bring, from a variety of fields, those who have made significant creative contributions that are relevant to the study of Architecture and Urban Design.

Lectures and exhibitions are open to the public.

Perloff Hall is located on the UCLA Campus.
Perloff Hall, M-F: 9am – 5pm
Lectures take place at 6:30pm in Perloff Hall, Decafé
Parking is available in Lot 3 for $10
Check the website for confirmation of all programs at www.audi.ucla.edu
Info: 310.267.4704
The campus map is available at http://www.audi.ucla.edu/pdfs/ucla-map.pdf
Statistical Computing in the 1960’s

A number of statistical systems and programs already existed; BMD and P-Stat were in current use and GenStat was being developed.

These systems grew out of specific application areas and tended to offer pre-packaged analyses.

At the time, most statistics researchers would not be directly involved in analyzing data; programmers (read graduate students) would do the grubby work when necessary.

The systems like BMD and SAS, for example, and PStat, to some extent, and GenStat’s another good example, all grew up in environments where statisticians were required to do some fairly committed routine analysis. So BMD, of course comes from a biomedical field; SAS from several areas but medical, again; and GenStat comes from an agricultural background.

Now in all those situations, the statistics groups, amongst other duties, were expected to be doing kind of analysis to order. You know, the data would come along from an experiment, or a clinical trial, or other sources, and as part of the job of the statisticians to produce analysis. Now often the analysis that they produced were relatively predetermined, or at least that’s how it worked out.

Interview with John Chambers, 2002
The mid 1960’s at Bell Labs

Statistics Research at Bell Labs tackled large-scale data analysis projects with teams of researchers and “programmers”

Unlike much of the statistical computing of the day, this kind of work was not well suited to pre-packaged programs

Even then, AT&T was creating large-scale applications; data from TelStar, an early (1960s) communications satellite, involved tens of thousands of observations

Launched by NASA aboard a Delta rocket from Cape Canaveral on July 10, 1962, Telstar was the first privately sponsored space launch. A medium-altitude satellite, Telstar was placed in an elliptical orbit (completed once every 2 hours and 37 minutes), revolving at a 45 degree angle above the equator. Because of this, its transmission availability for transatlantic signals was only 20 minutes in each orbit.

Telstar relayed its first television pictures (of a flag outside its ground station in Andover, Maine) on the date of its launch. Almost two weeks later, on July 23, it relayed the first live transatlantic television signal. During that evening it also dealt with the first telephone call transmitted through space and successfully transmitted faxes, data, and both live and taped television, including the first live transmission of television across an ocean (to Pleumeur-Bodou, in France). John F. Kennedy, then President of the United States, gave a live transatlantic press conference via Telstar.
The mid 1960’s at Bell Labs

During this period, John Tukey was also starting to formulate the beginnings of Exploratory Data Analysis

“Exploratory analysis is detective work -- numerical detective work -- or counting detective work -- or graphical detective work”

EDA is an approach to data analysis that postpones the usual assumptions about what kind of model the data follow with the more direct approach of allowing the data itself to reveal its underlying structure and model. EDA is not a mere collection of techniques; EDA is a philosophy as to how we dissect a data set; what we look for; how we look; and how we interpret.

Most EDA techniques are graphical in nature with a few quantitative techniques. The reason for the heavy reliance on graphics is that by its very nature the main role of EDA is to open-mindedly explore, and graphics gives the analysts unparalleled power to do so, enticing the data to reveal its structural secrets, and being always ready to gain some new, often unsuspected, insight into the data. In combination with the natural pattern-recognition capabilities that we all possess, graphics provides, of course, unparalleled power to carry this out.

Taken from www.itl.nist.gov
The mid 1960’s at Bell Labs

“Today, software and hardware together provide far more powerful factories than most statisticians realize, factories that many of today's most able young people find exciting and worth learning about on their own. Their interest can help us greatly, if statistics starts to make much more nearly adequate use of the computer. However, if we fail to expand our uses, their interest in computers can cost us many of our best recruits, and set us back many years.”

The technical tools of Statistics, Nov 1964
The mid 1960’s at Bell Labs

In previous decades, computers had matured significantly, but the access a user might have to these systems was limited; recall that in 1964, Bell Labs partnered with MIT and GE to create Multics (for Multiplexed Information and Computing Service)

“Such systems must run continuously and reliably 7 days a week, 24 hours a day in a way similar to telephone or power systems, and must be capable of meeting wide service demands: from multiple man-machine interaction to the sequential processing of absentee-user jobs; from the use of the system with dedicated languages and subsystems to the programming of the system itself”
The mid 1960’s at Bell Labs

While Bell Labs dropped out of the project in 1969, it did spark a lot of interest among researchers throughout the lab; John Chambers had just joined the lab and was starting to think about larger computing issues.

He and a small group of statisticians began to consider how this kind of computing platform might benefit the practice of statistics.

*What should the computer do for us?*
What should the computer do for us?

Let’s take a moment and answer that question for ourselves; you now have a fair bit of experience collecting, organizing and analyzing data with simple shell tools and Python scripts.

What operations would a computing environment designed for statisticians make easy? What kinds of data types should be “built-in”? How should one interact with it?

How do your answers change if you think instead about a someone looking at data, someone who might not be a statistician?
The mid 1960’s at Bell Labs

What should the computer do for us?

In answering this question, the Bell Labs group considered the necessary components of a system that would “more naturally express what we do and allow us to interactively program analyses”

Tukey produced a memo that outlined the basic concepts; memos were the emails of the mid 1960s

Adapted from Chambers (2000)
Follow the arrows clockwise from the Mind and Intuition block. Tukey’s notion is that data analysts have an arsenal of operations applicable to data, which they describe to themselves and to each other in a combination of mathematics and (English) words, for which he coins the term Analysh. These descriptions can be made into algorithms (my term, not his) -- specific computational methods, but not yet realized for an actual computer (hence the conceptual computer). Then a further mapping implements the algorithm, and running it produces output for the data analyst. The output, of course, stimulates further ideas and the cycle continues. (The facilitating transducers I interpret to mean software that allows information to be translated back and forth between internal machine form and forms that humans can write or look at -- a transducer, in general, converts energy from one form to another. So parsers and formatting software would be examples.)

Taken from Chambers (2000)
The mid 1960’s at Bell Labs

This group met again in the summer of 1965; Martin Wilk opened the meeting with the following

“What do we want? We want to have easy, flexible, availability of basic or higher level operations, with convenient data manipulation, bookkeeping and IO capacity. We want to be able easily to modify data, output formats, small and large programs and to do this and more with a standard language adapted to statistical usage.”

Unfortunately, the plans developed by this group remained just that, plans; it wasn’t until a decade later that a second group met, and on May 5, 1976, work started on what would become “S”
XABC: FORTRAN subroutine to provide interface between ABC and/or utility programs.

**XABC** (INSTR, OUTSTR)

Input **INSTR** →

```
| "X" |
| "Y" |
```

Argument Names or Blank

Pointers/Values

**OUTSTR** →

```
| "B" |
```

Pointers/Values

Types (Nodes)

Result Names

Note: Names are meaningful to Algorithm, not necessarily to Language.
The development of S

The image on the previous page is a scan of the first graphic produced at the May 5 meeting, and according to Chambers:

“The upper portion depicts the concept of an interface between a proposed user-level language and an ‘algorithm,’ which meant a Fortran-callable subroutine. The lower portion of the figure shows diagrammatically a hierarchical, list-like structure for data, the direct forerunner of lists with named components, structures with attributes and eventually classes with slots... The two portions of the sketch in fact lead to the themes of function calls and objects, in retrospect."
The development of S

By the end of 1976, John Chambers and Rick Becker had produced a working version of their sketch.

The program was initially referred to as “the system” locally, and attempts to come up with new names yielded lots of “unacceptable” candidates, but all of which had the letter “S” in common; and so, given the precedent of the recently-developed C language...
Unix and S

Becker and Chambers made the most of the Unix project happening in parallel at the labs at the same time; by creating a Unix version of S (S Version 2), their system became portable (or at least it could go anywhere Unix could)

At that point, AT&T started licensing both Unix and S, with both university groups and “third-party” resellers in mind; this meant others could contribute to the development of the software
The development of S

About a decade after the first version of S, a complete revision of the language took place (S3, described in the “blue book”) which gave rise to an extensive modeling effort (described in the “white book”); for most of you, this version of S will most closely resemble R (to come).

One more decade and another version (S4, described in Chambers’ book “Computing with Data”) that brought with it a new class and method system.

In 1998 Chambers won the ACM (Association for Computing Machinery) Software System Award; S has “forever altered the way people analyze, visualize and manipulate data.”
And finally...

Ross Ihaka and Robert Gentleman (both at the University of Auckland at the time) wrote a reduced version of S for "teaching purposes"

In 1995 Ross and Robert were persuaded to release the code for R under the GPL (the General Public License; it allows you to use it freely, distributed it, and even sell it, as long as the receiver has the same rights and the source code is freely available)

R is now administered by a core group of about a dozen people; but many more contribute code, and many many more program in R
The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.
The R environment

The term "environment" is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

R, like S, is designed around a true computer language, and it allows users to add additional functionality by defining new functions. Much of the system is itself written in the R dialect of S, which makes it easy for users to follow the algorithmic choices made.

Many users think of R as a statistics system. We prefer to think of it of an environment within which statistical techniques are implemented. R can be extended (easily) via packages.
The R environment

While there is a fair bit of difference between how R and S are implemented, one of the most visible for users involves how code is shared.

The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity.

In my mind, the ease with which packages can be published, accessed (installed) and auditioned is one of the real innovations of R.

As an aside, what do (nearly) all of the names on the right have in common?
From the top...

You can start an R process in a Unix terminal window simply typing the command `R`; you can also launch the R application by clicking on the R icon in your “dock”

This will eventually give you a **prompt**: R has a pretty simple user interface; the most basic form of interaction involves **entering expressions, having the system evaluate them and then print a result**

Sound familiar?
Sound familiar?

This session is similar to what we’ve seen with the Unix shell or Python’s interactive interpreter; in a sense, these are all different kinds of “shells”

You can think of a shell as any program that provides an interface to the “computational” services offered by some system (the Unix operating system, Python, R, SQLite, MongoDB)

As an interface, you can make a distinction between a command line shell and a graphical shell (or a graphical user interface); as we have seen, OS X offers you a little of both
From the top...

At its most basic level, R can be thought of as a bulky calculator; it can perform basic arithmetic and has a number of built-in functions; this is similar to our experience with Python

```r
> 5+7           #addition
[1] 12

> 0.2*7.58      #multiplication
[1] 1.516

> cos(2)
[1] -0.4161468

> cos(exp(4))   #compound
[1] -0.3706617

> sqrt(25)
[1] 5
```

In each case, R prints the result of the numeric operation to your screen; each output is prefaced by “[1]”; we will have more to say about why that is shortly.

For the moment, the important thing to realize is that you are entering mathematical expressions, having them evaluated and then R is printing the result to your screen.
Learning R

Learning R is a mix of remembering functions and knowing how to get help when you don’t; R has a pretty extensive (and largely usable) help system that can answer questions about its functionality.

For details on a function you can type

```r
> ?cos
> help("cos")
> help.search("cos")
```

More function calls; `help` takes a “string” argument, the name of the function we’d like help on, while `help.search` performs fuzzy or regular expression matching for the string you provided.
Trigonometric Functions

Description:

These functions give the obvious trigonometric functions. They respectively compute the cosine, sine, tangent, arc-cosine, arc-sine, arc-tangent, and the two-argument arc-tangent.

Usage:

cos(x)
sin(x)
tan(x)
acos(x)
asin(x)
atan(x)
atan2(y, x)

Arguments:

x, y: numeric vector

Details:

The arc-tangent of two arguments 'atan2(y,x)' returns the angle between the x-axis and the vector from the origin to (x,y), i.e., for positive arguments 'atan2(y,x) == atan(y/x)'.

Angles are in radians, not degrees (i.e., a right angle is pi/2).

All except 'atan2' are generic functions: methods can be defined for them individually or via the 'Math' group generic.

References:

One, two, three, four....

Arithmetic operations in R can be combined using familiar *infix notation*; operators are written between operands (as in 2+2), as opposed to prefix (+ 2 2) or postfix (2 2 +)

Parentheses are used to clarify precedence and operators are assigned orders as well (for example, exponentiation is evaluated before multiplication, which comes before addition)

```
> ((12^2)/3)+3
[1] 51

> 12^2/3+3
[1] 51
```

* The notation is certainly familiar, the description may not be!
Catching the result

So far, we have answered the siren call of the R prompt with simple expressions that are evaluated and the result is simply printed to the screen; typically, we want to store the output of an expression

\[ x \leftarrow 5+7 \]
\[ y \leftarrow \sqrt{10} \]
\[ z \leftarrow \exp(y) + x^2 \]

In each case, we have calculated some \textit{value} and \textit{assigned} it to a \textit{variable}; the assignment mechanism in R is different than what we saw in Python, we'll take this up in detail later
Catching the result

Variable names can contain letters, digits, "." and "_"; although they cannot begin with a digit or "_", and if they start with a ".", the second character can't be a digit

It is possible, of course, to get around these restrictions if you have some burning need to (and for most of you, this need will never materialize); enclosing your name in **backtick quotes** will tell the R evaluator (the program catching and processing your expressions) that the string is a (variable) name

```r
> `2008vote` <- 100+3

> `more a sentence than, say, a variable` <- sqrt(200)
```
Creating variables

Variables in R have both a name and a value; when we refer to variables by name, R simply prints their value

```
> x <- 5+7
> x
[1] 12

> y <- sqrt(10)
> y
[1] 3.162278

> z <- exp(y) + x^2
> z
[1] 167.6243
```

When we refer to these variables in later expressions, R will substitute their value
Creating variables

It is likely that many of you have had a bush with either R or something like R and none of this is earth-shattering; you can make assignments and there are some rules about what you can name things.

Note, however, that R uses “copying” semantics (you might also read that R is a “pass by value” language); this means when we assign the value of one variable to another, it is not “linked” to the original variable.

> x <- 5
> y <- x
> x <- 10

> x
[1] 10

> y
[1] 5
Creating variables

In addition to “<-” there are several kinds of assignment operators

> y = sqrt(10)
> x <- 5*7
> 5*7 -> x
> x <<- 5*7 # we’ll get to this later

According to Chambers, the first three are synonymous; the only thing that distinguishes them is the “grammar” associated with their application (for clarity, many prefer the consistent use of “<-”)

Historically, the arrows came first (bi-directional assignments were useful in an age before command line editing) and the “=” followed in later versions of the language
Objects

The variables we have created (the names we have assigned) all refer to objects; as we will see, everything in R is an object (sound familiar?)

While working in R, the system accumulates all of your objects in your workspace; you can view the contents of your workspace with the function called objects() or, out of deference to its UNIX roots, ls()

> x <- 5+7

> y <- sqrt(10)

> z <- exp(y) + x^2 ; w <- 100 # semi-colons

> objects()

[1] "w" "x" "y" "z"
Removing objects

We can remove objects with the functions `rm()` or `remove()`

```r
> objects()
[1] "w" "x" "y" "z"

> remove("x")
> objects()
[1] "w" "y" "z"

> rm("y","z")
> objects()
[1] "w"
```
A hasty retreat...

Before we know enough to be dangerous, let's exit out of R; or rather, quit R

This is done with the command \texttt{q()}; notice that like \texttt{help()} and \texttt{cos()}, \texttt{q()} is another function

You will be asked whether you want to save your workspace; for now answer “\texttt{y}”; you will then be returned to the UNIX prompt
The R session

And now, let’s get back into R; what do you notice?

Each time we start R, we create a new R session; during a session we will run a series of commands, and create a number of objects

We might read in data, fit a statistical model, view some graphical diagnostics of the fit, maybe run simulations, define functions, the works

Re-loading your previous workspace
Your workspace

As mentioned previously, R stores objects in your *workspace*

When you end your R session, you are asked if you want to save the objects you created; if you do, they are saved in a file called `.RData` that is stored in the directory where you started R (typed the command `R`)

When you start and R session from a directory where you have previously saved your workspace, it will restore it for you -- You can (and are encouraged to) have different directories devoted to different projects; each will have its own `.RData`
A bit more about your workspace

Technically, your workspace is only one of several locations where R can find data and functions, or, as we heard last time objects

Where did R find pi? (For that matter, where has it been finding cos() and help()?)
A bit more about your R workspace

Your workspace heads a list of places where R looks when it needs to associate a name with a value; we can see this list with the command

> search()

[1] ".GlobalEnv"   "package:stats"   "package:graphics"

Technically, your workspace is an “environment” in R (an environment is just a collection of bindings, a mapping between names and values); when we end our session this workspace disappears (again, we will explain the “[1]” and “[4]” shortly)

When we ask R for an object, it runs through this list in order and inquires whether each one has the relevant variable

We refer to this list as your search path
A bit more about your R workspace

We can find the location of an object as follows

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

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

When we make an assignment, we’ve told R to ignore the version in the other package
A bit more about your R workspace

If you are curious, you can see what R is holding in these different workspaces

```r
> objects("package:base")
```

[1] "AIC"                "ARMAacf"              "ARMAtoMA"
[4] "Box.test"           "C"                    "D"
[7] "Gamma"              "HoltWinters"          "IQR"
[10] "KalmanForecast"     "KalmanLike"           "KalmanRun"
[13] "KalmanSmooth"       "NLSstAsymptotic"     "NLSstClosestX"
[16] "NLSstLfAsymptote"   "NLSstRtAsymptote"    "PP.test"
[19] "SSD"                "SSasymp"             "SSasympOff"
[22] "SSasympOrig"        "SSbiexp"             "SSf1"
[25] "SSfpl"              "SSgompertz"          "SSlogis"
[28] "SSmicmen"           "SSweibull"            "StructTS"
[31] "TukeyHSD"           "TukeyHSD.aov"        "acf"
[34] "acf2AR"             "add.scope"           "add1"
[37] "addmargins"         "aggregate"           "aggregate.data.frame"
Finding things

We saw commands like `which` and `where` that search through a similar kind of path in Unix, looking for executable files (applications)

Naming and resolving names is an important activity and one that we have talked about in general -- Many of these ideas are common to R, Unix, Python...
Sidenote: Transcripts of your sessions

In each directory where you’ve run R, you will also find a file called `.RHistory`; R uses this file to maintain a list of your previous commands.

Commands are accumulated across sessions, so you can recover information on possibly dimly remembered analyses; You can retrieve and edit previously typed commands (from the current and any previously saved sessions) with the up- and down-arrow keys.

In general, R provides a series of functions that help you collect, tag, search and manage your history; in part, this is because the designers of how the language see you moving from casual analysis (tool user) to programmer (tool maker).

```r
> help("history")
```
Load or Save or Display the Commands History

Description:

Load or save or display the commands history.

Usage:

loadhistory(file = ".Rhistory")
savehistory(file = ".Rhistory")

history(max.show = 25, reverse = FALSE, pattern, ...)

timestamp(stamp = date(),
        prefix = "##------ ", suffix = " ------##",
        quiet = FALSE)

Arguments:

file: The name of the file in which to save the history, or from
      which to load it. The path is relative to the current working
directory.

max.show: The maximum number of lines to show. 'Inf' will give all of
          the currently available history.

reverse: logical. If true, the lines are shown in reverse order. Note:
         this is not useful when there are continuation lines.

pattern: A character string to be matched against the lines of the
         history

...: Arguments to be passed to 'grep' when doing the matching.

stamp: A value or vector of values to be written into the history.

prefix: A prefix to apply to each line.

suffix: A suffix to apply to each line.
Saving workspace

While on the topic of tracking history, you can save the contents of your workspace at any time with the function calls

\[
\begin{align*}
&> \text{save}(x,y,\text{file}=\text{"mywork.Rda"}) \\
&> \text{save.image()} \quad \# \text{shorthand, save everything to .RData}
\end{align*}
\]

When you quit \(q()\) out of R, \text{save.image} is called implicitly and you are prompted to save your work

Caution: While R is running, any objects you create live only in your computer’s memory; if the program dies for some reason, your work will be lost*

*Moral of the story: Saving your work periodically is not a bad idea!
The evolution of data in S

As we hinted earlier, S went through a series of changes as it evolved from its first form in the 70s; during this process, data structures and the organization of computations on these structures changed.

As a result there are several ways to think about the object structures in R; some of this history is still with us!

**Object types:** These are internal types defined in the underlying C implementation of the language.

**Vector structures:** These were, in some sense, the earliest classes of objects in S; they are indexable objects with extra attributes.

**S3 classes:** These are objects with class attributes that trigger a basic kind of method “dispatch” and we’ll see an example today.

**S4 classes:** Also known as formal classes, and a subject we will get to much later in the quarter.
The evolution of data in S

As a consequence, there are various ways that we can ask R to describe the data it's holding -- We can, for example, fetch the underlying type (in the C implementation) with the function `typeof()` (and you might see references to `mode()` or `storage.mode()` which can also be used)

Vectors are the most basic data structure in R and the function `is.vector()` will tell us if an object is a vector; this structure can hold only one kind of data, and `typeof()` will tell us which type it is

Prior to S3, object-oriented programming was not as widespread as it is now; since S3, we can ask any object what its class is with the function `class()`

Finally, the function `str()` exhibits a representation of the “internal” *structure* of an object and `object.size()` tells you how much memory an object is taking up (approximately)
Data structures in R

Over the next few lectures, we'll examine the basic data types in R -- As was the case with Python, these will be our starting point for making more complex objects

**vectors**: ordered collections of primitive elements

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

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

**data frames**: two-dimensional data tables

**factors**: categorical variables
Vectors

*Vectors* are a primitive type of object in R; a vector is just a collection of values grouped in a single container.

The basic types of values are

- **numeric**: real numbers
- **complex**: complex numbers
- **integer**: integers
- **character**: character strings
- **logical**: elements are TRUE or FALSE
- **raw**: elements are raw bytes

Vectors can also contain the *NA* symbol to indicate a missing value.

Aside from this special symbol, **vectors hold data of the same type** -- We refer to this type as the *mode* of the vector (How does this compare to Python? What is the analogous data type?)
Working with vectors

When displaying a vector, R lists the elements from left to right, using multiple rows depending on the “width” of your display; each new row includes the index of the value starting that row

This explains why we saw the “[ 1 ]” in front of each variable’s value we printed in our previous session; R is interpreting single values as vectors of length 1
Creating vectors

The simplest way to create a vector is to simply concatenate a series of primitive elements using the function `c(...)`

```r
> x <- c(1,2,3)  # numeric elements; when you type values, they are stored as numeric

> x <- c("brown","whitman")  # characters

> x <- c(TRUE,FALSE,TRUE,TRUE)  # logical
```
Creating vectors

Integer vectors are often used to create indices into other vectors (or more complex things like matrices); R has a number of simple functions that return integer vectors

```r
> x <- 1:100     # integer elements
> class(x)
[1] "integer"

> x <- rep(1,5)
> x
[1] 1 1 1 1 1

> x <- 4:6
> y <- rep(x,1:3)
> y
[1] 4 5 5 6 6 6
```
Creating vectors

In addition to `rep`, the function `seq` is also frequently used for creating vectors (this time of class `numeric`)

```r
> x <- seq(0,3,len=100)  # creates a sequence of 100
# evenly spaced values
# between 0 and 3

> x <- seq(0,3,by=1)     # another option
> class(x)
[1] "numeric"

> x
[1] 0 1 2 3
```
Creating vectors

You can also add names to the elements of a vector either when you create it

```r
> x <- c("a" = 1, "b" = 2, "c" = 3)  # adding names

> names(x)
[1] "a" "b" "c"
```

Or by using an assignment with the function `names()`

```r
> x <- 1:26
> names(x) = letters  # remember this syntax!
```
Creating vectors

Note that the default printing method for a vector changes when you assign it names

```r
> x <- 1:26
> x
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 16 17 18 19 20 21 22 23 24 25 26

> names(x) <- letters
> x
 a b c d e f g h i j k l m n o p
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
 q r s t u v w x y z
17 18 19 20 21 22 23 24 25 26
```
Vectorized operations

Since the basic object in R is a vector, many operations in the language work on entire vectors

For example, each variable below is a vector with length 100

```r
> x <- runif(100)
> z <- sqrt(x)
> y <- x + 2*z + rep(c(1,2),50)
```
The recycle rule

What happens if we add two vectors of different lengths?

R replicates or recycles the shorter vector to have the same length as the longer one.

```r
> c(1,2,3) + 3
[1] 4 5 6

> c(100,200,300,400)+c(1,10)
[1] 101 210 301 410

> c(100,200,300,400)+c(1,2,3)
[1] 101 202 303 401

Warning message:
longer object length is not a multiple of shorter object length in:
c(100, 200, 300, 400) + c(1, 2, 3)
Coercion

Since a vector can only hold data of one kind, R will coerce elements to a common type

```r
> x <- c(1:3,"a")
> x
[1] "1" "2" "3" "a"

> class(x)
[1] "character"
```

This might also happen when you try to perform an arithmetic operation

```r
> x <- c(TRUE,FALSE,TRUE,TRUE,TRUE)
> x + 3
[1] 4 3 4 4
```
Working with vectors

R lets you perform simple operations like sorting and finding just the unique values of a vector

```r
> sort(somenums)
[1] 0.1256883 0.1337630 0.3177054 0.3380988 0.4516134 0.5265388 0.5496565 0.6209405 0.8841781 0.9857302

> unique(c(TRUE,TRUE,FALSE,FALSE))
[1] TRUE FALSE

> x <- c(1,3,1,1,4,5)
> duplicated(x)
[1] FALSE FALSE  TRUE  TRUE FALSE FALSE
```
Attributes

You can assign attributes to any R object (except NULL); they are used to store names of vector entries, for example, or the dimension of a matrix (next few slides)

It is through this attribute mechanism that we move from vectors to S3 classes; an S3 object has an attribute “class”

```r
> x <- c('a'=1,'b'=2,'c'=3)
> attributes(x)
$names
 [1] "a" "b" "c"

> attr(x,"info") <- "store some details here"
> attributes(x)
$names
 [1] "a" "b" "c"

$info
 [1] "store some details here"
```
Data structures in R

Recall the evolution of the basic data structures in R; we’ll now consider matrices

vectors: ordered collections of primitive elements

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

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

data frames: two-dimensional data tables

factors: categorical variables
Matrices

We can have matrices of the same basic types as vectors, and like vectors they can only hold one type of data

```r
> x # characters
[1,] "a"  "f"  "k"  "p"  "u"
[2,] "b"  "g"  "l"  "q"  "v"
[3,] "c"  "h"  "m"  "r"  "w"
[4,] "d"  "i"  "n"  "s"  "x"
[5,] "e"  "j"  "o"  "t"  "y"

> y # integers
   [,1] [,2] [,3]
[1,]  1  3  5
[2,]  2  4  6
```
Note that a different printing convention is followed for matrices; we get an indication of the dimension of the matrix (its rows and columns).

```r
> w                  # numeric
[1,] 0.379 0.724 0.705 0.494 0.197 0.368 0.647
[2,] 0.477 0.935 0.121 0.475 0.699 0.046 0.469
[3,] 0.316 0.306 0.875 0.387 0.029 0.156 0.960
[4,] 0.172 0.860 0.836 0.492 0.624 0.294 0.950
[5,] 0.890 0.475 0.624 0.508 0.062 0.495 0.914

> u                  # logical
[1,] TRUE TRUE FALSE FALSE FALSE FALSE FALSE
[2,] TRUE TRUE TRUE FALSE FALSE FALSE FALSE
```
Matrices

You can create matrices from vectors

```r
> x <- 1:3
> y <- c(7,9,11)
> w <- cbind(x,y)       # bind together as columns
> w
   [,1] [,2]
[1,]  1  7
[2,]  2  9
[3,]  3 11

> w <- rbind(x,y)       # bind together as rows
> w
   [,1] [,2]   [,3]
[1,]  1  2  3
[2,]  7  9 11
```
Matrices

And you can use a special constructor

```r
> x <- matrix(runif(10),5,2)  # 5 rows and 2 columns
> x

   [,1]       [,2]
[1,] 0.8892423 0.73841712
[2,] 0.3158160 0.09425744
[3,] 0.7490556 0.93721217
[4,] 0.8831179 0.47290106
[5,] 0.2369386 0.96440986
```
Matrices

And you can use a special constructor; you can leave out either the rows or the columns if the data are the right length.

```r
> x <- matrix(runif(10),5)      # 5 rows and 2 columns
> x <- matrix(runif(10),5,2)    # 5 rows and 2 columns
```

R will run through the data, arranging it into the proper shape, filling in columns first; if you want row order, you specify that with

```r
> x <- matrix(1:10,5,2,byrow=T)
> x
 [,1] [,2]
[1,]  1  2
[2,]  3  4
[3,]  5  6
[4,]  7  8
[5,]  9 10
```
When R thinks about a matrix, it's really thinking about a vector with some additional information about the dimensions of the matrix -- These are stored as an integer vector of length 2

As we mentioned before, this kind of information is an attribute of the object -- Attributes can be associated with any object in R

We can get at the dimension via

```
> x <- matrix(runif(10),5,2)     # 5 rows and 2 columns
> dim(x)
[1] 5 2

> class(x)
[1] "matrix"

> attributes(x)
$dim
[1] 5 2
```
Matrices

Finally, you can change the shape of a matrix

```r
> x <- 1:10
> dim(x) <- c(2,5)       # remember this syntax also!
> x
[1,] 1  3  5  7  9
[2,] 2  4  6  8 10
```
Matrices

You can also assign names to the rows and columns of a matrix

```r
> x <- matrix(letters[1:8],4,2)
> rownames(x) <-
+ c("whitman","boxer","brown","fiorina")

> x
   [,1] [,2]
whitman  "a"  "e"
boxer    "b"  "f"
brown    "c"  "g"
fiorina  "d"  "h"

> colnames(x) <- c("var1","var2")
> x
   var1 var2
whitman  "a"  "e"
boxer    "b"  "f"
brown    "c"  "g"
fiorina  "d"  "h"
```
Matrices

These names are also stored as the attributes of the matrix; attributes are stored as a list -- something we will get to shortly

> attributes(x)
$dim
[1] 4 2

$dimnames
$dimnames[[1]]
[1] "whitman" "boxer" "brown" "fiorina"

$dimnames[[2]]
[1] "var1" "var2"
Vectorized operations

As with vectors, R can operate on entire matrices at once

> x

<table>
<thead>
<tr>
<th></th>
<th>var1</th>
<th>var2</th>
</tr>
</thead>
<tbody>
<tr>
<td>whitman</td>
<td>0.2638521</td>
<td>0.01246128</td>
</tr>
<tr>
<td>boxer</td>
<td>0.3532604</td>
<td>0.62873228</td>
</tr>
<tr>
<td>brown</td>
<td>0.3309769</td>
<td>0.17113513</td>
</tr>
<tr>
<td>fiorina</td>
<td>0.7694430</td>
<td>0.53395135</td>
</tr>
</tbody>
</table>

> (sqrt(x) + exp(x))^0.1

<table>
<thead>
<tr>
<th></th>
<th>var1</th>
<th>var2</th>
</tr>
</thead>
<tbody>
<tr>
<td>whitman</td>
<td>1.061456</td>
<td>1.011773</td>
</tr>
<tr>
<td>boxer</td>
<td>1.072737</td>
<td>1.103116</td>
</tr>
<tr>
<td>brown</td>
<td>1.070026</td>
<td>1.048144</td>
</tr>
<tr>
<td>fiorina</td>
<td>1.117446</td>
<td>1.093137</td>
</tr>
</tbody>
</table>
Operations on matrices

R also has a large number of built-in functions to compute with matrices (multiplication, addition) and to form various decompositions

```r
> x %*% y   # matrix multiplication
> w <- t(x) # transpose of x
> xinv <- solve(x) # inverse of x
> d <- diag(x) # extract the diagonal of x
> x <- diag(1,10,5) # create a 10x5 matrix, 1's on the diagonal
> eigen(x) # the eigenvalue decomposition, output as a list
```
Matrices

Finally, an operator that makes life a lot easier...

```r
> x <- matrix(1:50,10)
> dim(x)
[1] 10 5

> apply(x,1,sum)
[1] 105 110 115 120 125 130 135 140 145 150

> apply(x,2,mean)
[1]  5.5 15.5 25.5 35.5 45.5
```
Subsetting

R has 5 rules for selecting certain elements from a vector or a matrix -- They all use the operator “[”

- Indexing by position
- Indexing by exclusion
- Indexing by name
- Indexing with a logical mask
- Empty subsetting
Subsetting

Indexing vectors by position

```r
> x <- letters
> x[1]
[1] "a"

> x[c(10,11)]
[1] "j" "k"

> x[c(10,11,43)]
[1] "j" "k" NA

> x[0]
character(0)

> x[c(0,10,11)]
[1] "j" "k"
```
Subsetting

Exclusion of elements in a vector by position

> letters[-c(10,11)]
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "l" "m"
[12] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x"
[23] "y" "z"

> letters[c(-10,11,43)]
Error: only 0's may mix with negative subscripts
Subsetting

Indexing a vector by name

```r
> x <- 1:3
> names(x) <- c("a","b","c")

> x["a"]
a
1

> x["a","b"]
Error in x["a", "b"] : incorrect number of dimensions

> x["-a"]
Error in -"a" : Invalid argument to unary operator
```
Subsetting

Indexing a vector with a logical mask

```r
> x <- 1:5
> x[c(TRUE,TRUE,FALSE,FALSE,FALSE)]
[1] 1 2

> x >= 3  # remember the recycle rule!
[1] FALSE FALSE TRUE TRUE TRUE

> x[x >= 3]
[1] 3 4 5

> x[x >= 2 & x < 4]  # combining w/ logical operators
[1] 2 3
```
Logical operators

These will appear both in subsetting as well as to control program flow (with loops, if-statements, and so on) -- they include operators like <, >, <=, >=, == and !=

On vectors, they apply elementwise (vectorized) and return a vector of logicals (TRUE or FALSE depending on whether the element satisfied the condition or not)

You can combine them with & for AND, and | for OR; the function any(x) returns TRUE if any element in x is true and all(x) returns TRUE if all the elements in x are true
Subsetting

And finally, empty subsetting

```r
> x <- 1:5
> x[ ]
[1] 1 2 3 4 5
```
Subsetting

Uh, why the empty subsetting rule?

Notationally, this makes sense when you consider the subsetting operations for matrices which operate on rows and columns -- It also comes up when we talk about assignments for vectors...
Assignments for vectors

We can modify specific elements in a vector using our subsetting rules

```r
> x <- 1:5
> x[c(2,4)] = 0
> x
[1] 1 0 3 0 5

> x[x < 2] <- 10
> x
[1] 10 10  3 10  5

> x[] <- 5
> x
[1] 5 5 5 5 5
```
Subsetting

The subsetting syntax for matrices comes right from vectors and we can apply the same basic rules.

Indexing by position

```r
> x <- matrix(letters[1:10],5,2)
> x[1,2]
[1] "f"

> x[c(3,4,5),c(1,2)]
[,1] [,2]
[1,] "c" "h"
[2,] "d" "i"
[3,] "e" "j"
```
Subsetting

Indexing by exclusion

```r
> x <- matrix(letters[1:10],5,2)
> x[1,-2]
[1] “a”

> x[-c(3,4,5),c(1,2)]
 [,1]  [,2]
[1,] "a" "f"
[2,] "b" "g"
```

Indexing with a logical mask

```r
> x <- matrix(letters[1:10],5,2)
> x[c(T,T,T,F,F,F),c(F,T)]
[1] “a” “g” “h”
```
Subsetting

Indexing by name

```r
> x <- matrix(letters[1:8],4,2)

> rownames(x) <-
+ c("whitman","boxer","brown","fiorina")

> colnames(x) <- c("var1","var2")
> x
      var1 var2
whitman  "a"  "e"
boxer    "b"  "f"
brown    "c"  "g"
fiorina  "d"  "h"

> x[c("boxer","fiorina"),c("var1","var2")]
      var1 var2
boxer   "b"  "f"
fiorina "d"  "h"
```
Subsetting

Empty subsetting

```r
> x
     var1 var2
  john  "a"  "e"
bryon  "b"  "f"
hao    "c"  "g"
chenni "d"  "h"

> x[,1]
     john  bryon  hao  chenni
      "a"    "b"    "c"    "d"

> x[,1,drop=F]
     var1
  john  "a"
bryon  "b"
hao    "c"
chenni "d"
```

# single column becomes a vector
# make it matrix with one column
Subsetting

Recall that the empty subset represented the entire vector; so when we read something like \( x[,1] \), we want all of the elements in the empty position (in this case rows)

Note that in the second case, R returned a vector rather than a matrix (all notion of \( var1 \) disappeared) -- It is the default behavior in R to drop the structure for a single column and return a vector (to keep that from happening use the \( \text{drop=F} \) parameter)
Subsetting

Since R views a matrix as a fancy vector, you can make some indexing shortcuts:

```r
> x <- matrix(letters[1:10],5,2)
> x
 [,1] [,2]
[1,] "a" "f"
[2,] "b" "g"
[3,] "c" "h"
[4,] "d" "i"
[5,] "e" "j"

> x[rbind(c(1,2),c(1,1),c(3,2))]  # matrix for an index
[1] "f" "a" "h"

> x[1:7]                           # vector for an index
[1] "a" "b" "c" "d" "e" "f" "g"
```
Subsetting

Or what about this?

```r
> x <- matrix(letters[1:10], 5, 2)
> x
   [,1] [,2]
[1,]  "a"  "f"
[2,]  "b"  "g"
[3,]  "c"  "h"
[4,]  "d"  "i"
[5,]  "e"  "j"

> y <- matrix(runif(10), 5, 2)
> y
   [,1]        [,2]
[1,] 0.63061690 0.99150369
[2,] 0.06743318 0.57479874
[3,] 0.94084094 0.95612238
[4,] 0.32397860 0.01915618
[5,] 0.06459324 0.33232415

> x[y > 0.5]
[1] "a" "c" "f" "g" "h"
```
A tour of R through Shazam

But this is just the briefest of introductions -- For the rest of the lecture, I want to apply this to solve a real problem, well, “real” in the sense that it’s one of your homework assignments.

I promised that we’d make the underlying components of the Shazam algorithm a bit more comprehensible -- Delivering on that promise will give us a pretty good introduction to many of the features of R.

Rather than keep on with vectors, let’s plunge ahead and look at a (slightly) more complicated “object” in R, a “class” that represents audio or sounds -- We’ll see that it’s been implemented as a kind of extension of vector.

The first task, it would seem would be to bring “audio data” into R -- We are going to start with a very simple type of sound file but will indicate at the end of the lecture how things can be made more complex.
WAV files

The WAV format (or WAVE, which is short for the Waveform Audio File Format) is actually pretty simple -- A audio waveform is represented as **amplitude values recorded at a sequence of times** (there’s some extra detail we could go into about how these amplitudes are quantized, but that’s enough for now)

Matthias Heymann (now at Duke) has developed “sound”, a package that allows you to load a WAV file into R and perform **basic manipulations on the underlying waveform** -- At its core, a sound file is a sequence of amplitudes and the R “object” Matthias developed behaves like a **vector**

So, let’s have a look at CRAN, the Comprehensive R Archive Network...
sos
A Sound Interface for R
classes and methods for spatial data

sound

sp
Univariate and Multivariate Spatial Modeling

spBayes
SPEcies Association Analysis

spaa
Sparse PArtial Correlation Estimation
classes and methods for spatio-temporal data

spacetime
SPArse Matrix

space
Perform sparse hierarchical clustering and sparse k-means clustering

spatcounts
Sparklines and graphical tables for tex and html

spatgraphs
The sparr package: SPAtial Relative Risk

spatial
development regression

spatialCovariance
Spatial count regression

spatialkernel
Graphs for spatial point patterns

spatialsegregation
Functions for Kriging and Point Pattern Analysis

spatstat
Computation of spatial covariance matrices for data on rectangles

spatial
Nonparametric estimation of spatial segregation in a multivariate point process

spatialcov
Segregation measures for multitype spatial point patterns

spatialCovariance
Spatial Point Pattern analysis, model-fitting, simulation, tests

spatial
Statistical Process Control

spc
Spatial Coverage Sampling and Random Sampling from Compact Geographical Strata

spcosa
Spatial dependence: weighting schemes, statistics and models

spdep
Stochastic Proximity Embedding

spectralGP
Approximate Gaussian processes using the Fourier basis

speedglm
Fitting Linear and Generalized Linear Models to large data sets

spef
Semiparametric estimating functions
sound: A Sound Interface for R

Basic functions for dealing with wav files and sound samples.

Version: 1.3
Depends: R (≥ 1.4.1)
Published: 2010-09-28
Author: Matthias Heymann
Maintainer: Matthias Heymann <mail at MatthiasHeymann.de>
License: GPL (≥ 2)
URL: http://www.MatthiasHeymann.de
CRAN checks: sound results

Downloads:

Package source: sound_1.3.tar.gz
MacOS X binary: sound_1.3.tgz
Windows binary: sound_1.3.zip
Reference manual: sound.pdf
Old sources: sound archive

Reverse dependencies:

Reverse suggests: seewave
Package ‘sound’

September 28, 2010

Version 1.3
Date 2010-09-27
Title A Sound Interface for R
Author Matthias Heymann <mail@MatthiasHeymann.de>
Maintainer Matthias Heymann <mail@MatthiasHeymann.de>
Depends R (>= 1.4.1)
Description Basic functions for dealing with wav files and sound samples.
License GPL (>= 2)
URL http://www.MatthiasHeymann.de
Repository CRAN
Date/Publication 2010-09-28 07:22:36

R topics documented:

appendSample ................................................. 2
bits ........................................................... 3
center ....................................................... 4
channels ..................................................... 5
cutSample ................................................... 6
cutSampleEnds ............................................. 7
duration ..................................................... 8
fitSampleParameters ...................................... 10
left ............................................................. 11
loadSample .................................................. 12
mirror ......................................................... 13
normalize .................................................... 14
noSilence .................................................... 15
nullSample ................................................... 16
Ops.Sample .................................................. 17
Loading packages

One of the great success stories of R (as opposed to S) is it’s package management system -- It is very easy to share code with others and to make use of other people’s code

In the lab, the sound package should already be installed and it is also available on homework.stat202a.org -- If you would like to follow along with this lesson on your own machine, however, you will first have to install the package...

```r
> install.packages("sound")  # not necessary in the lab or homework
> library(help="sound")      # what's this package about?
> library("sound")
```
Information on package 'sound'

Description:

Package:            sound
Version:            1.3
Date:               2010-09-27
Title:              A Sound Interface for R
Author:             Matthias Heymann <mail@MatthiasHeymann.de>
Maintainer:         Matthias Heymann <mail@MatthiasHeymann.de>
Depends:            R (>= 1.4.1)
Description:        Basic functions for dealing with wav files and sound samples.
License:            GPL (>= 2)
URL:                http://www.MatthiasHeymann.de
Packaged:           2010-09-27 18:40:30 UTC; Matthias
Repository:         CRAN
Date/Publication:   2010-09-28 07:22:36
Built:              R 2.11.1; ; 2010-09-29 06:59:30 UTC; unix

Index:

Ops.Sample    Basic Operations for Sample Objects
Sample        Sample Objects
Sine          Create Sample Objects for the Basic waveforms
WavPlayer     Set or Get the System Command for Playing WAV Files
appendSample  Append Sample Objects
bits          Bits per Sample
center        Center a Sample Object.
channels      Number of Channels of a Sample Object
cutSample     Cut Sample Objects
cutSampleEnds Prepare Sample Object for appendSample
Now, let’s get busy!

The sound package was built around a use case in which audio files are stored on your local computer, that is, we refer to WAV files with a file name -- One small improvement to this package would be to allow for general “connections” so that one could pick up data from the web

For now, if you are working on a lab machine (or your own laptop), you can just download files from our course web site using the following R command

```
> download.file(
+    "http://www.stat.ucla.edu/~cocteau/stat202a/jay-z.wav",
+    "jay-z.wav")
```

trying URL 'http://www.stat.ucla.edu/~cocteau/stat202a/jay-z.wav'
Content type 'audio/x-wav' length 4430002 bytes (4.2 Mb)
opened URL
=================================================================
downloaded 4.2 Mb

(You can also use files directly from /data/music on homework)
# read "empire state" by jay-z into r using a function from the sound package

> jz <- loadSample("jay-z.wav")
> class(jz)
[1] "Sample"

> jz
type : mono
rate : 8000 samples / second
quality : 16 bits / sample
length : 2214979 samples
R memory : 8859916 bytes
HD memory : 4430002 bytes
duration : 276.872 seconds

# a Sample object consists of the sound (audio waveform) as a matrix (in case there
# was a left and right channel) and constants representing the sampling rate and the
# number of bits used per sample

> names(jz)
[1] "sound" "rate" "bits"

> class(jz$sound)
[1] "matrix"

> dim(jz$sound)
[1] 1 2214979

> jz$rate
[1] 8000
> jz$bits
[1] 16
# in terms of methods, we can subset a Sample object, focusing on just the first
# 4 seconds, say (remember the WAV file had a sample rate of 8000 samples/second)...

\[ jz[1:32000] \]

\begin{itemize}
  \item type : mono
  \item rate : 8000 samples / second
  \item quality : 16 bits / sample
  \item length : 32000 samples
  \item R memory : 128000 bytes
  \item HD memory : 64044 bytes
  \item duration : 4 seconds
\end{itemize}

# ... or we can make a “custom” plot, specially designed for this class of object
# about 55 seconds into the song, alicia keys starts to sing (8000 samples/second)

\[ \text{plot}(jz[(53\times8000+1):(57\times8000)]) \]
R and objects

And with this, we see our first hint at proper **objects in R** -- In this case, we say that `Sample` is an example of an S3 class (although we won’t go into a lot of detail now except to say that the S refers to the S language and 3 refers to its version; and there are also S4 classes associates with S version 4).

To apply the term “object” to `jz`, we expect to find descriptions of **both state and behavior**, data and methods -- While objects as containers for data are not philosophically dissimilar from those we saw in Python, the way we invoke methods in R is considerably different.
R and objects: Data-directed programming

We have already noted that, as with Python, everything in R is an object -- The designers of S/R believe strongly that through objects our programs become easier to design and, ultimately, more trustworthy.

According to Robert Gentleman “... it is often easier to design, write and maintain software when there is some clear separation of the data representation from the operations that are to be performed on it.”

Or, as John Chambers puts it “As you become more involved with software for data analysis, however, creating related classes and methods... [makes] the analysis more natural and convenient... and the results more trustworthy”
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.

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.
Generic functions

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>
S3 classes and methods

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* and `is.object()` tests to see if an R object has a class attribute

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()`

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

# find all the methods for this function that R knows about...

> methods("summary")
[1] summary.aov            summary.aovlist        summary.aspell*
[4] summary.connection     summary.data.frame     summary.Date
[7] summary.default        summary.ecdf*          summary.factor
[10] summary.glm            summary.infl           summary.lm
[13] summary.loess*         summary.loglm*         summary.manova
[16] summary.matrix         summary.mlm            summary.negbin*
[19] summary.nls*          summary.packageStatus* summary.polr*
[22] summary.POSIXct        summary.POSIXlt        summary.ppr*
[25] summary.prcomp*        summary.princomp*      summary.rlm*
[28] summary.srcfile        summary.srcref         summary.stepfun
[31] summary.stl*          summary.table          summary.tukeysmooth*

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
Non-visible functions are asterisked
> 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
Aside: Some history

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 define themselves via 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())

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}, \texttt{glm}, \texttt{gam}, \texttt{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.
Returning to our example

In the next few slides, we’ll make use of two classes of objects (one for the audio signal, \texttt{Sample}, and one for the spectrogram, \texttt{specgram}) -- We’ll make use of commands like \texttt{help()}, \texttt{class()}, \texttt{names()}, \texttt{typeof()} and \texttt{methods()} to do a little exploration of what each object is all about...
Returning to our example

We have already seen how to read a WAV file into R and have found that the resulting object is of class `Sample` -- We can have a look at the various methods Matthias provided us to work with this class

```r
> jz <- loadSample("jay-z.wav")
> class(jz)
[1] "Sample"

> jz

type      : mono
rate      : 8000 samples / second
quality   : 16 bits / sample
length    : 2214979 samples
R memory  : 8859916 bytes
HD memory : 4430002 bytes
duration  : 276.872 seconds

> methods(class="Sample")
[1] [.Sample Ops.Sample play.Sample
[7] sampleLength<-.Sample sum.Sample
```
A second sound file

We’ll use as a comparison Sinatra’s “New York, New York” (why not?) --
Again, you can either download the file or use it directly from /data/music
on homework

```r
> download.file(
+   "http://www.stat.ucla.edu/~cocteau/stat202a/sinatra.wav",
+   "sinatra2.wav")

trying URL 'http://www.stat.ucla.edu/~cocteau/stat202a/sinatra.wav'
Content type 'audio/x-wav' length 3299244 bytes (3.1 Mb)
opened URL
=================================================================
downloaded 3.1 Mb

> sinatra <- loadSample("sinatra.wav")
> class(sinatra)
[1] "Sample"

> sinatra

type : mono
rate : 8000 samples / second
quality : 16 bits / sample
length : 1649600 samples
R memory : 6598400 bytes
HD memory : 3299244 bytes
duration : 206.2 seconds
```
For homework you outlined the basic steps of the Shazam algorithm -- After we've acquired some data, the first step involves computing a time-frequency representation of an audio signal known as a spectrogram.

For this, we appeal to a second package, `signal`, which is basically a port to R of a number of signal processing functions from Matlab.

```r
> install.packages("signal")  # not necessary in the lab/homework
> library(help="signal")      # what's this package about?
> library("signal")
```
# compute the spectrogram for 15 seconds of each of jay-z and sinatra -- the
# exact choice was made so that something changed in the song during the interval

# the constants define the sampling rate for the signal and the size of the
# window we want to use to compute the fourier transform (here, 64ms)

> jzspec <- specgram(jz$sound[(50*8000+1):(65*8000)],64*8000/1000,8000,64*8000/1000)

> class(jzspec)
[1] "specgram"

# a specgram object has data representing the “matrix” of intensities, and vectors
# of the times and frequencies at which it was computed -- the matrix is made up of
# complex numbers

> names(jzspec)
[1] "S" "f" "t"

> class(jzspec$S)
[1] "matrix"

> dim(jzspec$S)
[1] 256 467

> typeof(jzspec$S)
[1] "complex"
# and now examine the methods associated with specgram objects

> class(jzspec)
[1] "specgram"

> methods(class ="specgram")
[1] plot.specgram  print.specgram

# have a look at the plot function, for example -- it’s just a call to image()
# note that it maps black to low intensity and white to high...

> plot.specgram
function (x, ...)
{
    image(20 * log10(t(abs(x$S))), col = gray(0:512/512), axes = FALSE, ...)
}

> plot(jzspec)

# finally, compute the spectrogram for francis albert...

> sinspec <- specgram(sinatra$sound[(10*8000+1):(25*8000)],64*8000/1000,8000,64*8000/1000)
default specgram plot: 15 seconds of jay-z.wav
we can make a plot that’s closer to a figure in the shazam paper by letting white
denote low intensity and black be high intensity -- we also show the values un-logged
and add axis labels

the function gray() takes values between 0 and 1 -- 0 is black and 1 is white and
returns a hexadecimal representation of the color (more on this later)
> grays <- gray((100:0)/100)

the function image() takes vectors that correspond to the column and row values
(here time and frequency) and a matrix

first make an image from 15 seconds of jay-z -- 5 seconds in, alicia keys starts singing

> image(jzspec$t,jzspec$f,t(abs(jzspec$S)),col=grays,xlab="time",ylab="frequency")
> title("jay-z, empire state of mind")

... and now 15 seconds of sinatra -- it’s band until 6 seconds in, then he
"starts spreading the news..."

> image(sinspec$t,sinspec$f,t(abs(sinspec$S)),col=grays,xlab="time",ylab="frequency")
> title("sinatra, (new york)^2")
jay-z, empire state of mind
sinatra, (new york)^2
Since the spectrogram is really a matrix, we can look at slices at fixed times (columns) to compare the peaks in intensity -- We subset a matrix with a pair of indices, the first representing rows, the second columns

```r
# start by pulling out some of the components of the specgram object --
# technically it’s better to leave them with the object but for
# readability of the next few slides, i’ll pull them out

> freqs <- jzspec$f
> times <- jzspec$t

> jzmat <- abs(jzspec$S)
> sinmat <- abs(sinspec$S)

> plot(freqs,jzmat[,400],type="l",xlab="frequency",ylab="intensity")
> lines(freqs,jzmat[,100],col=5)
> title("slices before and after alicia keys (cyan=before, black=after")
```
slices before and after alicia keys (cyan = before, black = after)
# the code, for completeness -- how many times does alicia keys sing?
# (hint: rhymes with door)

> bigspec <- specgram(jz$sound,64*8000/1000,8000,64*8000/1000)
> plot(spec$t,abs(spec$S[177,]),type="l",xlab="times",ylab="intensity at 2750 Hz")
> title("a slice from the full spectrogram for jay-z")
Identifying constellations

The paper is somewhat vague about the way in which high points are identified -- We’ll take a relatively simple approach for this first introduction and leave it to you to enhance it.

We’ll divide the 256 frequencies (rows in the specgram object) into 16 bins and create strips that cut across time -- We then compute the maximum in each column in the strip and tag as constellation points the top 5% of these maxima.

This simple approach is particularly well-suited to fast computations in R -- It will give us a chance to see vector and matrix operations...
# let's consider a chunk of rows that go from 209 to 224 (or 3250 Hz to 3484.375 Hz)

```r
> i <- 14
> ((i-1)*16+1):(i*16)
```

```r
> strip <- jzmat[((i-1)*16+1):(i*16),]
```

# now find the max values in each column -- that is, for each column in the strip
# look for the maximum of the 16 values (the second line creates a matrix where
# each column is assigned its max value

```r
> maxs <- apply(strip,2,max)
> mmaxs <- matrix(maxs,nrow=16,ncol=ncol(strip),byrow=T)
```

# finally, take the top 5% of the max values...

```r
> cutoff <- quantile(maxs,p=0.95)
```

# ... and create a binary matrix

```r
> const <- (strip==mmaxs) & (mmaxs>cutoff)
```

# and let's have a look...

```r
> image(times, freqs[((i-1)*16+1):(i*16)],t(strip),col=gray((100:0)/100))
> image(times, freqs[((i-1)*16+1):(i*16)],t(const),col=gray(c(1,0)))
```
# collecting these lines into a function that will repeat the operation for
# all 16 strips, pasting together the result into a matrix that’s the same
# dimension as the original spectrogram

> find_constellation <- function(mat){

    const <- matrix(TRUE,nrow=nrow(mat),ncol=ncol(mat))

    for(i in 1:16){

        strip <- mat[((i-1)*16+1):(i*16),]

        maxs <- apply(strip,2,max)
        mmaxs <- matrix(maxs,nrow=16,ncol=ncol(strip),byrow=T)

        cutoff <- quantile(maxs,p=0.95)

        const[((i-1)*16+1):(i*16),] <- (strip==mmaxs)&(mmaxs>cutoff)
    }

    return(const)
}

# and applying it to jz and sinatra sounds

> jzmap <- find_constellation(jzmat)
> sinmap <- find_constellation(sinmat)

> image(times,freqs,t(jzmap),col=gray(c(1,0)))
> image(times,freqs,t(sinmap),col=gray(c(1,0)))
constellation for jay–z
constellation for sinatra
Targets

It’s likely that one of the next components of your system involves identifying a set of targets associated with each point in the constellation -- Because time and frequency have such different scales, we opted to represent each point by it’s row and column indices in the spectrogram.

Then, given a point in the constellation, we identify it’s targets as the 10 nearest points that are ahead of it in time -- Nearest is judged using Euclidean distance with the row and column indices of candidates.

A better approach would be to use a weighted Euclidean distance that rescales each coordinate (time and frequency) so that one can more precisely tailor the extent to which targets should be in front of versus above and below a point in the constellation.
\begin{align*}
(1, 6) \\
(2, 2) & (2, 4) & (2, 9) \\
(3, 7) \\
\end{align*}

\begin{align*}
d^2 &= 5 \\
d^2 &= 25 \\
d^2 &= 10 \\
\end{align*}
# start by changing from a matrix with TRUE/FALSE to a collection
# of pairs, each representing the row and column index of the point

> mfreqs = matrix(1:length(freqs),ncol=length(times),nrow=length(freqs))
> mtimes = matrix(1:length(times),ncol=length(times),nrow=length(freqs),byrow=T)

> jzconst <- cbind(mtimes[jzmap],mfreqs[jzmap])

# and have a look...

> plot(jzconst,col=5,pch=19,xlab="columns",ylab="rows")

# pick a point in the constellation (here we just take the 186th, somewhat
# randomly) and identify 10 targets ahead of it in time

> i <- 186

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

> for(j in 1:10) lines(rbind(jzconst[i,],candidates[j,]))

# doing this a few times creates the plot on the next page... the order() function
# returns the set of indices required to rearrange a vector into ascending order --
# here we provide it another variable to sort on, the distance between the
# 186th constellation point and all the candidate points (we’ll talk about its
# format later)

> idist <- function(pt,candidates){
    apply(candidates,1,function(x,y){ sqrt(sum((y-x)^2)) },y=pt)
}
Hashing

Presumably a later step involves computing a hash of each of the triples representing a segment joining a constellation point and a target (the first and second frequencies and the time difference between them)

The purpose of the hash function is to provide us with a mapping of these triples of information into a single number so that we can search a database more quickly -- Having to hunt along for matching end points could take a long time if you have to, in effect, run a program for each match (even a fraction of a second per song could add up to hours if your collection of music is large enough)

Databases can be highly optimized to conduct searches on single columns of numbers (among other things, of course)
Hashing

Suppose for the moment that we need to simply encode the row numbers associated with the frequencies in the spectrogram matrix -- Each of these are integers that run from 1 to 256

Recalling that $256 = 2^8$ we can use the following simple mapping to go from the row number for frequency 1, $r_1$, and that for frequency 2, $r_2$:

\[
\begin{align*}
&> r_1 \leftarrow 129 \\
&> r_2 \leftarrow 205 \\
&> h \leftarrow (r_2-1)*2^8 + (r_1-1) \\
&> h \\
&[1] \quad 52352
\end{align*}
\]
Hashing

We can then “decode” the hash using the following mod operators -- First we look at the remainder after division by $2^8$ and then rescale the rest...

```r
> h
[1] 52352

> r1d <- (h %% 2^8) + 1
> r2d <- (h-r1d+1)/2^8

> r1d
[1] 129

> r2d
[1] 204
```
Hashing

To encode the time difference, we’ll use another 8 bits -- Each **unit difference between the columns of the spectrogram represents 32 ms** by design so that a difference of 255 means \(0.032 \times 255 = 8.16\) seconds

That's plenty of time as we're assuming we will have between 3 and 30 constellation points per second -- To encode this value along with the others is really no more than another shift and rescaling

See? Hashing is not so mysterious -- And now, with this function in hand, we can create our “database” of constellations...
## Here are two usable hash functions -- we should be careful about time differences larger than 256 (we’ll talk about how to protect ourselves from this kind of thing later)

```r
> ihash <- function(r1, r2, dtr) {
    dtr*2^16 + (r2-1)*2^8 + (r1-1)
}

> dihash <- function(h) {
    r1 <- (h %% 2^8) + 1
    h <- (h-r1+1)/2^8
    r2 <- (h %% 2^8) + 1
    dtr <- (h-r2+1)/2^8
    return(c(r1, r2, dtr))
}

> ihash(15, 224, 131)
[1] 8642318

> dihash(ihash(15, 224, 131))
[1] 15 224 131
```
Pulling it all together

We now create a single function that will go from a Sample object through to a collection of hashes -- We are assuming we’ll process each song in overlapping 15 second segments, building hashes for all the constellation points that occur before the 10 second mark

The following function also assumes a mono sound file with a sample rate of 8000 samples per second and 16 bits per sample -- It’s possible to reformat just about any sound file to these specifications (or you can make this all more robust!)
```r
shazam <- function(snd){

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

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

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

hashes <- NULL

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

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

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

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

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

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

return(hashes)
}
```
```r
# test to see if we have any hashes that match between jay-z and sinatra...

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

# now try an unknown sample recorded by my laptop

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

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

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

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

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

> plot(ushaz[,1:2], pch=19, col=5, cex=0.5, xlab="time", ylab="frequency")
> points(ushaz[,1:2][ushaz[,3] %in% jzshaz[,3]],)
```
Ok, so this lecture was packed -- Some basic components in the language and a brief introduction to at least one of the object models in R

It’s a lot to digest and over the next few lectures we’ll pull things together properly -- For the moment, I just wanted to give you a sense of how we can use R to help us refine our Shazam algorithm

As an aside, we can use the shazam function we created to piece together constellations for the entire song -- If we then run pieces of our unknown sample through it, we see big matches at sensible places...
seconds 25–40 of unknown
seconds 35–50 of unknown