

- A sequence of integers:


```
> 11:17
```

 What if the second number is smaller than the first?
- A sequence of equally spaced real numbers


```
> seq( 3.2, 12, .4) ##OR seq( 3.2, 12, length=40)
```
- The `c` for *combine* function and an assignment


```
> heights <- c(71, 65, 68, 68, 70)
  ## builds the object, does not print it
```
- Use `scan` for interactive input. Return twice to stop.


```
> heights <- scan()
1: 71 65 68
4: 68 70
6:
Read 5 items
```

Alternatively, `=` can be used for assignment, but it has two other meanings, so `<-` is preferred. **Use informative names.**

Vector Arithmetic

Addition, multiplication, etc. of vectors is done element-by-element. (If you want matrix multiplication you have to ask for it specially with `%*%`.)

Caution: If one vector is shorter than the other, R recycles the shorter one, reusing the first elements.

```
> short.vectr <- c(1,2)
> heights / short.vectr
[1] 71.0 32.5 68.0 34.0 70.0
Warning message:
In heights/short.vectr :
  longer object length is not a multiple of shorter
  object length
```

If `heights` had 6 elements, we would get no warning. In some situations, the warning may be hidden. Though dangerous, this can be very useful, for example when adding a constant to a vector.

The usual operators, `+`, `-`, `/`, `*` work as expected. R uses the regular order of operations. Parenthesis are used to change order.

```
5 + 3*2^2
(5 + 3*2)^2
5 + (3*2)^2
2*1:3^2 # surprise!
```

Arithmetic Functions:

`log10` `log` `exp` `sqrt` `sum` `prod` `cumsum` `cumprod`

All of these work with vectors.

Extraction

To extract values from a vector, use square brackets.

```
> heights
[1] 71 65 68 68 70
> heights[4:5]
[1] 68 70
> heights[c(3,5,1)]
[1] 68 70 71
```

You can also change certain values using `[]`.

```
> heights[1:2] <- 67 # gets recycled to fill two
  positions
> heights
[1] 67 67 68 68 70
```

And you can use logical statements (TRUE or FALSE) to pull out some elements.

```
> (heights < 70)
[1] TRUE TRUE TRUE TRUE FALSE
> heights[heights < 70]
[1] 67 67 68 68
```

Input From File

Usually our data is stored in a plain text file separated with commas (.csv), tab (.txt), or spaces. You need to know what the data looks like in order to read it in to R.

Do not edit data files with a word processor. They add lots of formatting info which makes the file impossible to read. In Windows use WordPad or Excel. I recommend using comma separated values (csv) format and a spreadsheet.

You can use `scan(file="myfile.txt")` but we will emphasize `read.table` and its relatives.

```
> NBA <- read.csv("data/NBAtickets.csv", head=T)
> diamonds <- read.table("http://www.amstat.org/publications/jse/datasets/4c.dat", head=F)
> names(diamonds) <- c("carat", "cut", "color", "clarity", "depth", "table", "price", "x", "y", "z")
```

These functions check that each row has the same number of values. They build a "data frame" (looks like a matrix, but matrices only hold numbers)

Jim Robison-Cox

R Intro, Day 2

read.table Options

- Can read from a URL if you're on the web.
- Can skip lines with `, skip=3`
- Can specify the delimiter and what is a missing value.
`, sep="\t", na.string="."`
- `header = TRUE` means that the first line is a list of column names. Use all caps for TRUE and FALSE.

Common problems:

Using the default space delimiter with a split word like "New Jersey" not in quotes.

Here's a way to see which lines cause a problem.

```
> numEntries <- count.fields("file.txt")
> summary(numEntries)
> which(numEntries != 5)
```

Rstudio: use tab for file name completion. Windows and Mac: browse for a file on your computer using:

```
myDataFrame <- read.table(file.choose(), head=T)
```

Jim Robison-Cox

R Intro, Day 2

Getting Help

Basic

```
> help(read.table) ## OR
> ?read.table     ## you may want to first do
> help.start()    ## so help displays in a browser
  window
> args(read.table) ## simpler form
```

Search for more

```
> help.search("linear model") ## lots of hits
> RSiteSearch("pairwise comparison")
> example(pairs) ## uses of pairs function
> demo(graphics) ## many different plots
> vignette("frame") ## load pdf file(here from grid
  package)
```

Jim Robison-Cox

R Intro, Day 2

read.table creates a dataframe

A data frame is like a simple spreadsheet in that each subject's data is a row and each measurement (variable) is a column.

Columns may be numeric or character data. If character, they are converted into a "factor". Look at a summary to see the difference:

```
> summary(diamonds[,1:2])
  carat      cut
Min.   :0.200   Fair      : 1610
1st Qu.:0.400   Good       : 4906
Median :0.700   Very Good :12082
Mean   :0.798   Premium   :13791
3rd Qu.:1.040   Ideal     :21551
Max.   :5.010
```

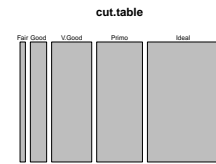
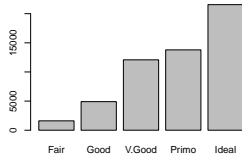
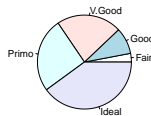
Summaries for categorical variables are frequency tables. For quantitative variables they are five-number summary and the mean. How would you plot the distribution of values for a (categorical) factor? for a quantitative variable?

Jim Robison-Cox

R Intro, Day 2

Plots for Categorical Data

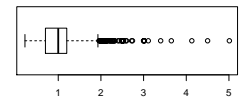
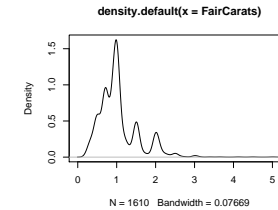
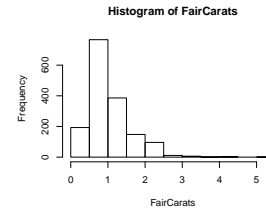
```
> cut.table <- table(diamonds$cut) ## tabulate the data
pie(cut.table)      barchart(cut.table)      mosaicplot(cut.table)
```



Pie charts are discouraged because it's hard to compare angles. Heights (bar plot) or widths (mosaicplot) are easier to compare visually.

Plots for Quantitative Data

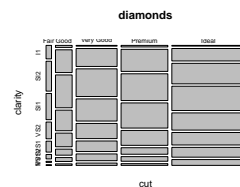
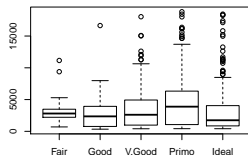
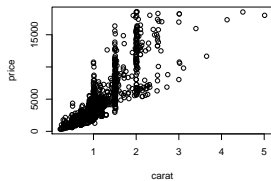
```
FairCarats <- subset(diamonds, cut == "Fair")$carat
hist(FairCarats)
plot(density(FairCarats))
boxplot(FairCarats, horizontal=TRUE)
```



```
> stem(subset(kidsfeet, sex=="G")$length)
The decimal point is at the |           ## stem and
leaf plot
18 | 6
20 | 59067
22 | 000255675
24 | 0017
```

Plots for Two Variables

```
plot(price ~ carat, data=diamonds, subset = cut=="
Fair") ##OR
with(subset(diamonds, cut=="Fair"), plot(carat, price
))
boxplot(price ~ cut, diamonds[sample(53940,540),])
mosaicplot(cut~clarity, diamonds)
```



Dataframes

- Two ways to create a dataframe

```
stat505 <- data.frame( names = c("x X", "y Y", "z Z"
),
                      bannerID=c("0086", "0023", "
0099"),
                      HW1 = 10)
diamonds <- read.table("http://www.amstat.org/
publications/jse/datasets/4c.dat")
names(diamonds) <- c("carat", "color", "clarity", "
cert", "price")
```
- A list of columns, not a matrix. Each column is a vector of numbers or a factor.
- Extract one column using

```
stat408$HW1 ## the dollar sign for a list
stat408[["HW1"]] ## [["name"]] or [[3]] for a
list
stat408[,3] ## get 3rd column (like a matrix)
stat408[,-3] ## all but 3rd column (like a
matrix)
```

Use `names(stat408)` to see column names of a data.frame.
 Use `colnames(stat408)` for a matrix or dataframe.
 Extract using dollar sign or square brackets.
 Or `attach` a dataframe to add its columns as variables to our workspace.

```
ls()           ## list available objects
search()      ## show search path
attach(diamonds)
search()      ## how has search path changed?
ls(pos=2)     ## where are these objects?
```

Problems with `attach`

- Changes to the dataframe do not propagate. Must `detach()` and then `attach()` again.
- Name collisions: Two attached dataframes having a common column name. Which "x" R will find first?
- Poor programming practice. See "R style Guide from Google" on the class home page.

Functions like `plot()` allow us to specify `data=diamonds`. Otherwise, use "with" to temporarily attach the dataframe, then detach.

```
with(diamonds, plot(carat, price))
## or just a subset:
with(subset(diamonds, cert == "GIA"), plot(carat, price))
```

To see what attributes this dataframe has:

```
is.data.frame(diamonds)
is.matrix(diamonds)
is.list(diamonds)
class(diamonds)
class(diamonds$carat); plot(diamonds$carat); summary(diamonds$carat)
class(diamonds$cut); plot(diamonds$cut); summary(diamonds$cut)
```

Class determines how R handles an object.

Every object has a "class". See Chapter 7 in "R Nutshell".

`plot` and `summary` are generic functions. They look for a special version of themselves to use on any particular class.

Typing the name of a function may provide its definition.

```
> q
```

Is an internal function.

```
> ls ## that's el-es gives a definition
> summary
> print
> summary.factor
```

`summary` and `print` are generic functions. `summary.factor` is visible. It is a version of `summary` specifically built to summarize a factor variable.

We will not be creating generic functions, but we do need to know that they exist. Otherwise some R output would be very mysterious.

Logical Comparison

Operators

<	less than	<=	less than or equal	!=	not equal
>	greater than	>=	greater than or =	==	equal
!	not	,	or	&, &&	and
all(x)	all TRUE?	xor(x,y)	one TRUE, not all	any(x)	any TRUE?

| and & are used in subset and ifelse to evaluate vectors.

|| and && are used in flow-control if statements on 1st elements.

with(diamonds, **which**(color=="D" & cert=="GIA")) tells us which elements of the dataframe satisfy both conditions.

Each class has a test function like **is.list()** above.

R has

```
if ( age > 30){  
  print(" Untrusted")  
} else {  
  print(" Trusted")  
}
```

if only evaluates the first element of a vector. Use **ifelse** to evaluate each element.

```
X.is <- ifelse(x == 3, "x is 3", "x <> 3")
```

Type Conversion

Convert one type to another.

```
(counts <- matrix(1:12, nrow=4, ncol=3))  
class(counts)  
class(ncounts <- as.numeric(counts))  
class(ncounts) <- "matrix" ## can't just reset it  
attr(ncounts, "dim") <- c(3,4) ## set dim to make it a  
  matrix  
ncounts  
class(countDF <- as.data.frame(counts))  
names(countDF) <- c("col1", "col2", "col3")  
unlist(countDF)  
unclass(diamonds$cert) ## removes the factor class  
class(unclass(diamonds$cert))
```

Note: a matrix is stored as a stack of columns with a dimension attribute. Changing its dimension does not alter the order, does not transpose it.

Function Construction

- Build a function for repetitive analyses
- Speeds analysis, less room for error.
- Start with a single run-thru to debug.
- Identify inputs and outputs.

Build a function to tabulate fish by length class (25 mm groups) and mark.

```
ruby <- read.csv("Ruby-AllFish.csv")  
rubyRBT2006 <- subset(ruby, species=="RBT" & site=="  
  Ghorn" & year==2006 & length >100 )  
summary(rubyRBT2006)  
with(rubyRBT2006, table(cut(length, seq(100,475,25)),  
  mark, trip)) ## problem: trip 1 is never marked  
rubyRBT2006$type <- with(rubyRBT2006, ifelse(trip ==  
  1, "pass1", ifelse(mark == 1, "both", "pass2")))  
with(rubyRBT2006, table(cut(length, seq(100,475,25)),  
  type))
```

What are the inputs and outputs?