What is R and RStudio

- R is an interactive programming language that we will interact with through the program RStudio. RStudio is an interactive development environment that makes it easier in some ways to interact with R. There is a desktop version of the program that you might have installed on your computer, but there is also a server version of RStudio that is hooked up to the infrastrastructure at SESYNC, which gives you more space and memory, and importantly allows you to access files that live on our servers such as group project spaces.
- What we'll cover here is also applicable RStudio

Overview

In this morning session we will focus on "base" R, i.e. the functionality that comes built into the language without installing any other packages. We'll cover:

- basic ways R thinks about data
- how to get your data into R if it lives on a disk
- ways to manipulate and subset your data
- basic plotting
- how to write a function

Load up rstudio.sesync.org

check sticky notes

You should see 3 or 4 panels in the window

1. console
2. global environment and history
3. file browser, help, plots, etc.
4. A fourth panel if you open up a new file

The `>` prompt

At its very basic, R can act like a big calculator. Let's try that out by typing some math into the console window and pressing enter

```
1 + 2
```

There is a lot of basic math built into R. You can use all of the usual suspects like `+ - / ^ *` as you might expect. Notice how the answer appears in the console window and gives you another prompt arrow `>`

Record your work in a script

So R is keeping track of everything that you type into the console. You can access this information in the history window, or by using the up arrow on your keyboard at the `>` prompt to access previous commands you've entered.

Usually you will want to do more with R than a few simple calculations like this. So instead of writing what you want directly into the console, let's open up a new file to write into.

File, new file, R Script
check sticky notes

Type into the R script file some math the same way you did at the console before. You can run the code by clicking `Run` or using cmd/ctrl + Enter. By default this runs the line of code where your cursor is, but you can run multiple lines at once by highlighting them.

Comments

Any line that starts with a `#` is not read by the computer. This is a great way to explain the steps of what you are doing which makes it easier to remember what you are doing with certain pieces of code. Also makes it easier for others to understand what is going on. Use them like breadcrumbs!

Create a vector

We can create an ordered list of things like numbers, called a **vector** by combining them together with `c()` and commas as such

```
c(1,2,3,4,5)
```

If the numbers are all in order like we have here, you can use the shorthand `:' to not need to type them all out. This can be used in conjunction with commas.

```
c(1:5,7:10)
```

Assignment operator `<-`

Store your vector by giving it a name and using the assignment operator `<-` which looks like a left-facing arrow. A short-cut for creating it is `alt` + `-` although I'm not sure that saves any time.

We know this by looking at the environment

The point of this is that now you can refer to that vector, or any item or group of items in that vector using its name and **bracket notation**

Brackets allow you to refer to the number of an item in a vector, keeping in mind that the items in a vector are *ordered* like the alphabet.

R has a stored in alphabet vector named letters. combine the 3rd, 9th, and 19th items of letters into a new vector and give it an appropriate name. What is the appropriate name?

green sticky when done

Getting help

If we wanted to know more about the letters vector, we can use the built-in help by asking `help(letters)` or `?letters` for short. Notice that the help is in a tab in the lower right corner panel of rstudio and it also has a search bar.

Character vectors

Another thing to notice about the letters is that each one of them is in quotes. How is this different than `myvector` with the numbers 1 through 5, what do you think this means?

Data frames

Say we wanted to store some number data associated with each letter that was stored in another vector. Make a vector of the numbers 1 through 26 called numbers

```
numbers <- c(1:26)
```

Combine the built-in `letters` and your new `numbers` into a **data frame** with the aptly named function `data.frame()` which has a period between the words. This is like the `c()` we used before to make vectors except now we are combining vectors together.

```
data.frame(letters, numbers)
```

Save this data frame as `mydf`. How many rows and columns are in `mydf`?

Another helpful function to know about is `str()` which is short for structure, and allows you to easily find out information about an object in your environment. Where else do you see this information in the RStudio window?

- in the environment window
- by opening up the object
- printing it in the console

Ten to fifteen minutes in

Loading data into R

Now that we've got a handle on the very basics, we're going to load some data into R. Data could live in the cloud, on the web, in a database, as papers in your desk drawer, or any number of places. R has ways to handle most of those, but right now we're dealing with data that lives on a disk in flat files. This is fairly common.

The data we'll be using throughout this workshop is about a small mammal community in southern Arizona that's been monitored over the past 35 years. It's part of a larger projet studying the effects of rodents and ants on the plant community. There are different types of experimental plots where they manipulate which rodents have access to the plots. It's real data that's simplified for our purposes but it's been used in over 100 publications.

`read.csv()`

The data are in a folder in the sesync public-data folder which you all have access to read from. We'll use the `read.csv()` function which requires knowing the file path name. It is on the etherpad or you can type along with me. The general syntax for this functino is to supply the file name and then optional arguments such as whether or not the file name has column names in the first row of the file.

```
read.csv('/nfs/public-data/CSI2015/plots.csv')
```

Let's look at the help menu for the `read.csv()` function. How do we get help? In the help menu usage we see the **arguments** to pass to the function. When we read in the plots.csv file, we didn't specify which argument we were supplying, so R assumed by default that it was the first argument that it asks for. We could have also written `file =` and it would do the same thing.

Read in the files `plots.csv` and `surveys.csv` and store them as data frames called plots and surveys with the assignment operator.

```
plots <- read.csv('/nfs/public-data/CSI2015/plots.csv')
surveys <- read.csv('/nfs/public-data/CSI2015/surveys.csv')
```

How many rows and columns are in the surveys data frame? What about in the plots data frame?

green sticky when you have them both read in and can tell how many rows and columns are in each

---

Info about data frames

Some functions to get a quick idea of your data are `dim, nrow, ncol, names, str, summary, head`

What is a data frame? Under the hood, a data frame is a list of equal-length vectors, which makes it sort of like an Excel spreadsheet. It is one of the most commonly used objects in R. An important thing to keep in mind is that each column has all of the same type of information within it, like characters or numbers, but each column in the data frame can have a different type of data. Let's look at what types of information are in each of the columns in the `surveys` data frame.

dollar sign operator

You may have seen this information already in the console for the `str` function. Notice the dollar signs `$` at the beginning of each line returned. This is an easy way to access a column in a data frame where you know the name of the column.

Lists

Briefly, let's not forget that data frames are **lists** under the hood. Lists are a flexible type of object in R and can store basically anything you want. Whereas we use `c()` to construct vectors and `data.frame()` to construct data frames, we make lists with `list()`.

if there is time otherwise skip to types of vectors `c()` can construct vectors from lists by combining several lists into one. If you combine a vector and a list, R will coerce the vectors into a list and then combine them.

Notice the difference between:

```
str(list(list(1,2), c(3,4)))
str(c(list(1,2),c(3,4)))
```

Types of vectors

When we think about the type of data that could be in each of the columns, it will typically be one of 4 major types:

- **integer**: strictly positive whole numbers
- **numeric**: numbers including negatives and decimals
- **character**: words or letters
- **logical**: true or false

Types from least to most flexible are: logical, integer, double, character

Logical data

Logical data is stored as either a 1 or 0, True or False. They become very handy in conjunction with logical operators such as `<` and `<=`.

- less than
- less than or equal to
- greater than
- greater than or equal to
- **exactly equal to** is a double equal sign. The single equal sign works like the assignment operator does in R, so it is not in the same category of these logical operators
- **not equal to** is another one that is a little funky, and it is an exclamation point followed by an equal sign.
- **x | y** and **x & y** can be used to combine statements.

Factors

A factor is a special type of integer vector in R, and perhaps the most confusing and frustrating type of vector to work with, which is why we need to talk about it.

Factors are special types of **integer vectors** that can only contain predefined values. They are used to store categorical data and by default, `read.csv()` makes all of your character data into factors. The things that make factors special is that they have an attribute `levels()` which stores the predefined set of values that the categories can be. This is sometimes handy but a very important distinction from other types of character data.

Notice how when factor data are summarized they behave differently than integers, by converting the month column in surveys data frame to a factor with `as.factor()`

```
surveys$month <- as.factor(surveys$month)
summary(surveys)
```

What is different from before?

Add a column

Notice how we just converted a column in the data frame with the assignment operator. Just as easily, we can create new columns in a data frame as well.

Make a new column in the plots table called `Control` that is a logical vector based on whether or not that plot is a control plot. These plots have `plots$plot_type == "Control"`. It may be easier than you think!

```
plots$control <- plots$plot_type == "Control"
```

How can you verify that you made a logical vector?

subset with brackets `[ ]`

The square brackets work as follows: anything before the comma refers to the rows that will be selected, anything after the comma refers to the number of columns that should be returned.

- **nothing**: returns everything
- **positive integers**: return elements at the specified positions
- **logical vectors**: select elements where the corresponding logical value is true. this is what you used to create the new column control
- **negative integers**: omit elements at the selected positions

subset with double brackets `[[ ]]`

Note the way that subset double brackets with the name of the column is the same as subset with the `$` operator. Use double brackets

select rows based on a condition

`subset()`

This is a specialized function for subsetting data frames that saves time because you don't need to repeat the name of the data frame. Look at the help menu to see what the arguments are.

use the `subset()` function to create a new data frame called `surveys1990` that consists of surveys done in the year 1990.

Base plotting

`plot()`

This is the most basic function for plotting in R. It is polymorphic, which means that is uses the input data to determine the plot type. If your data only has one number, it will plot all of those values.

scatterplot

To plot two values against each other in a scatterplot, use `plot(x,y)` or `plot(y~x)`

Plot the surveys `wgt` column against month and year

histogram

Histograms are made with the `hist()` function. This is also a good time to introduce the `log()` function.

boxplots

Boxplots, or box and whisker plots, use a similar syntax to the alternative to scatterplots with the `~` that can be thought of as "by". Use `boxplot()` to plot `surveys$wgt` by `year` and then `month`

if there is time: **color using factors**
`r boxplot(surveys$wgt ~ surveys$year*surveys$month, col=levels(surveys$month))`

graphical parameters

We can adjust many graphical parameters

- xlab, ylab, main
- pch, col, cex
- lty, lwd
- ylim, xlim

multi-panel plots

Another nice thing to do is compare plots side by side or vertically.

`par(mfrow=c(x,y)` where x is the number of rows and y is the number of columns
`par(mar=c(a,b,c,d)` to adjust the margins of the panels starting from a being the bottom

```
surveys1990 <- subset(surveys, year==1990)
surveys1996 <- subest(surveys, year==1996)

par(mfrow=c(1,2)
hist(log(surveys1990$wgt))
hist(log(surveys1996$wgt))
```

saving figures

(If there is time)

Functions

When you start using and writing functions, the real power of R programming will really allow you to use your time efficiently. If you find yourself repeating very similar lines of code, those are usually the areas to think about writing a function. Then your project becomes more modular and easier to change or update. It can also reduce your chances of making mistakes

components of a function

A function needs to have a name, it probably has at least one argument, and a body of code that does something. At the end is usually returns something, even if just a message to say if it worked.

The syntax looks something like this:
`r myfunction <- function(x) { # does something here # return a value }` The name of this function would be what? What is the argument passed to this function?

The agruments control how the function operates when you call it. Let's see how this works

write a stderr function

The base R language does not have a function to calculate the standard error of the mean. Crazy! That's okay though because all of the components we need do exist as functions: `sqrt` `var` and `length` for sample size. Let's write a function to calculate the standard error of the `wgt` column in the surveys table.

First let's create a subset of the surveys data frame that doesn't include any missing values. There is a handy functino for this called `na.omit`

```
surveys <- na.omit(surveys)
```

Recall that the standard error is calculated as the square root of the quantity variance over the sample size. Using the three functions given, find the standard error of the wgt column in your subsetted data.

```
sqrt(var(surveys$wgt)/length(surveys$wgt))
```

```
[1] 0.1601575
```

green sticky when done

We can generalize this calculation that we made by storing it as a function called `stderr`. Typing `stderr` is a lot shorter than the whole thing we just did.

```
stderr <- function(x) {
    sqrt(var(x)/length(x))
    }
```

Calculate the standard error of the mean of the `wgt` column for the subset of surveys done in 1990 using your new function. Then do the same for all surveys done in the month of June.
Green stickies when you're done

```
stderr(subset(surveys, year==1990)$wgt)
stderr(subset(surveys, month==6)$wgt)
```

```
[1] 0.5726112
[1] 0.5745725
```

good practice

- keep functions short and in manageable chunks
- use comments to describe the inputs to the function, what it does, and what is the output
- check for errors along the way. test out your function with different examples and try to "break" it