Review: R Basics

I. Getting Started

1. What are R and RStudio?

> "**R** is a free software programming language and software environment for statistical computing and graphics. The **R** language is widely used among statisticians and data miners for developing statistical software and data analysis."

- **R** is similar to the S language (commercial program S-Plus). However, **R** is open source.
- You can use **R** alone or in combination with the **RStudio** interface

> "**RStudio** is a set of integrated tools designed to help you be more productive with **R**. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management."

  - We prefer the **RStudio** interface because it has a more user-friendly layout
  - We will install both **R** and **RStudio**

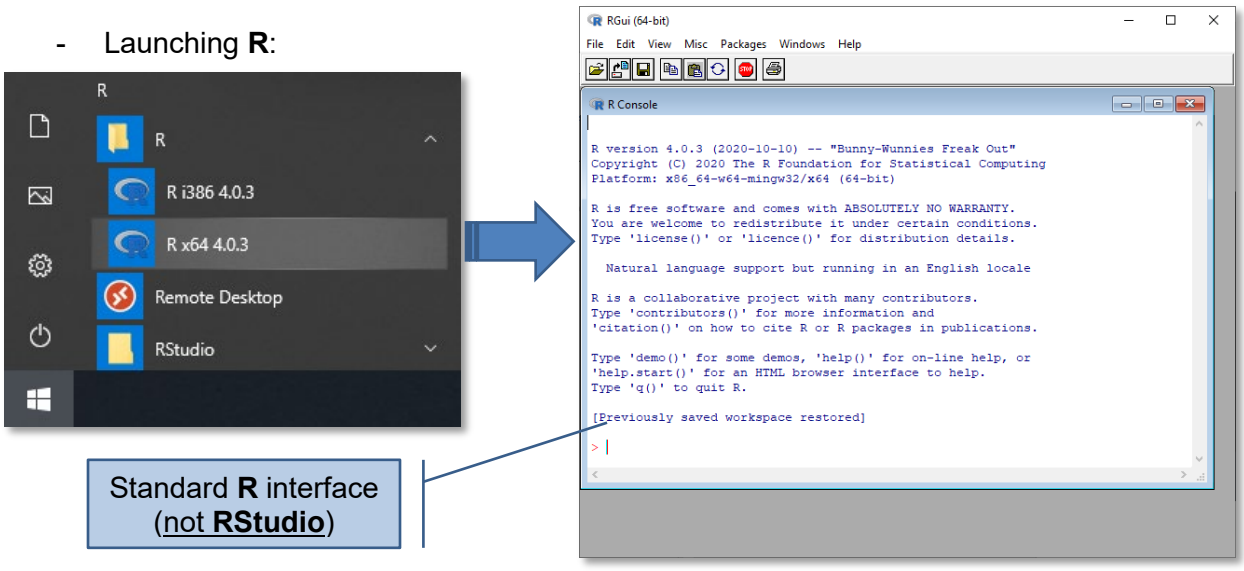
✓ **Note:** You do **not** need to have both **R** and **RStudio** running. You only need to open **RStudio**, and it will run **R**. However, you must have **R** installed to run **RStudio**.

2. Installing R

- You must install both **R** and **RStudio**, (1) first installing **R**, (2) then installing **RStudio**
- *Download* appropriate installation file for your operating system (Mac OS X or Windows) and *install*
  - **Windows:** Visit https://cran.r-project.org/bin/windows/base/ > "Download **R** 4.0.3 for Windows" to download .exe file. Choose the default installation options.
  - **Mac:** Visit https://cran.r-project.org/bin/macosx/ > Download R-4.0.3.pkg.

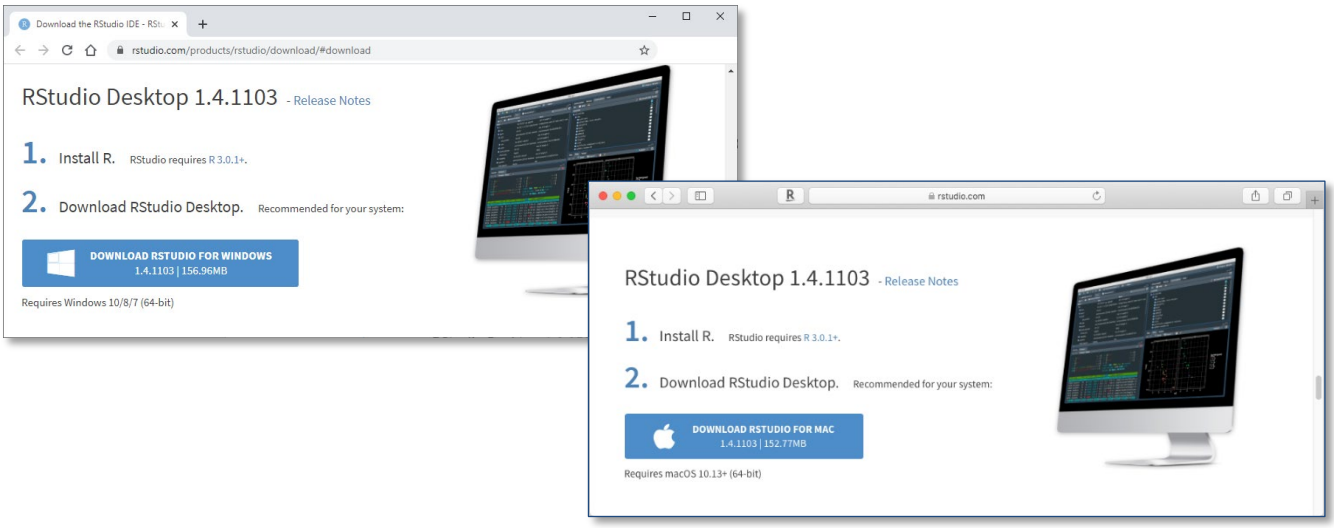
✓ **Note:** Release version may change. **R** 4.0.3 and **RStudio** 1.4.1103 are the current releases as of 25 January 2021.

- Launching **R**:

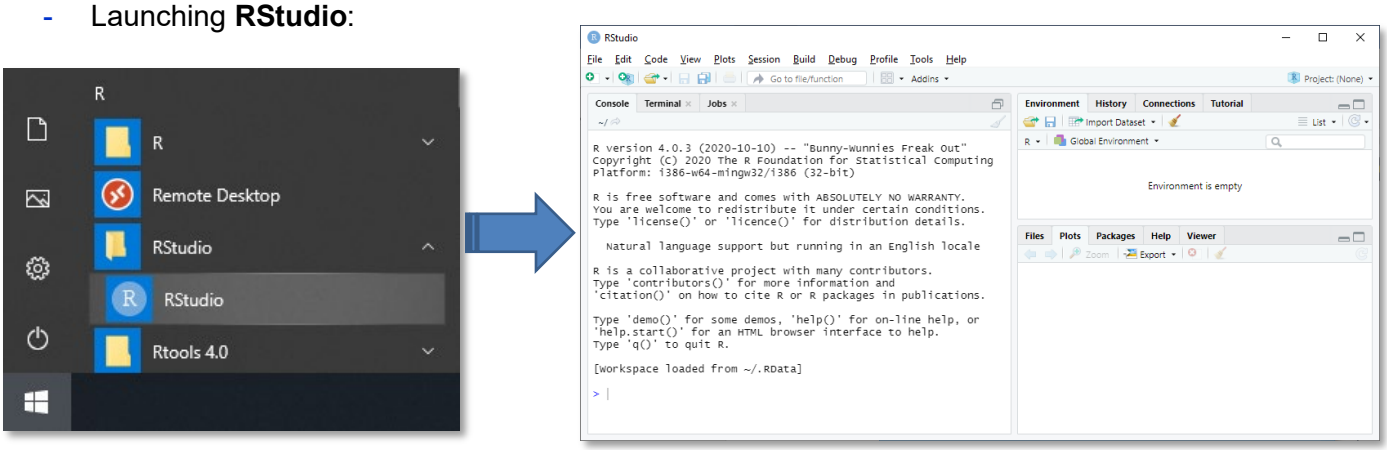


3. Installing RStudio

- *Download* the appropriate installation file for your operating system (Mac OS X or Windows) and *install*
    - **Windows/Mac:** Visit https://rstudio.com/products/rstudio/download/#download > Download **RStudio** Desktop, Open Source Edition (free)

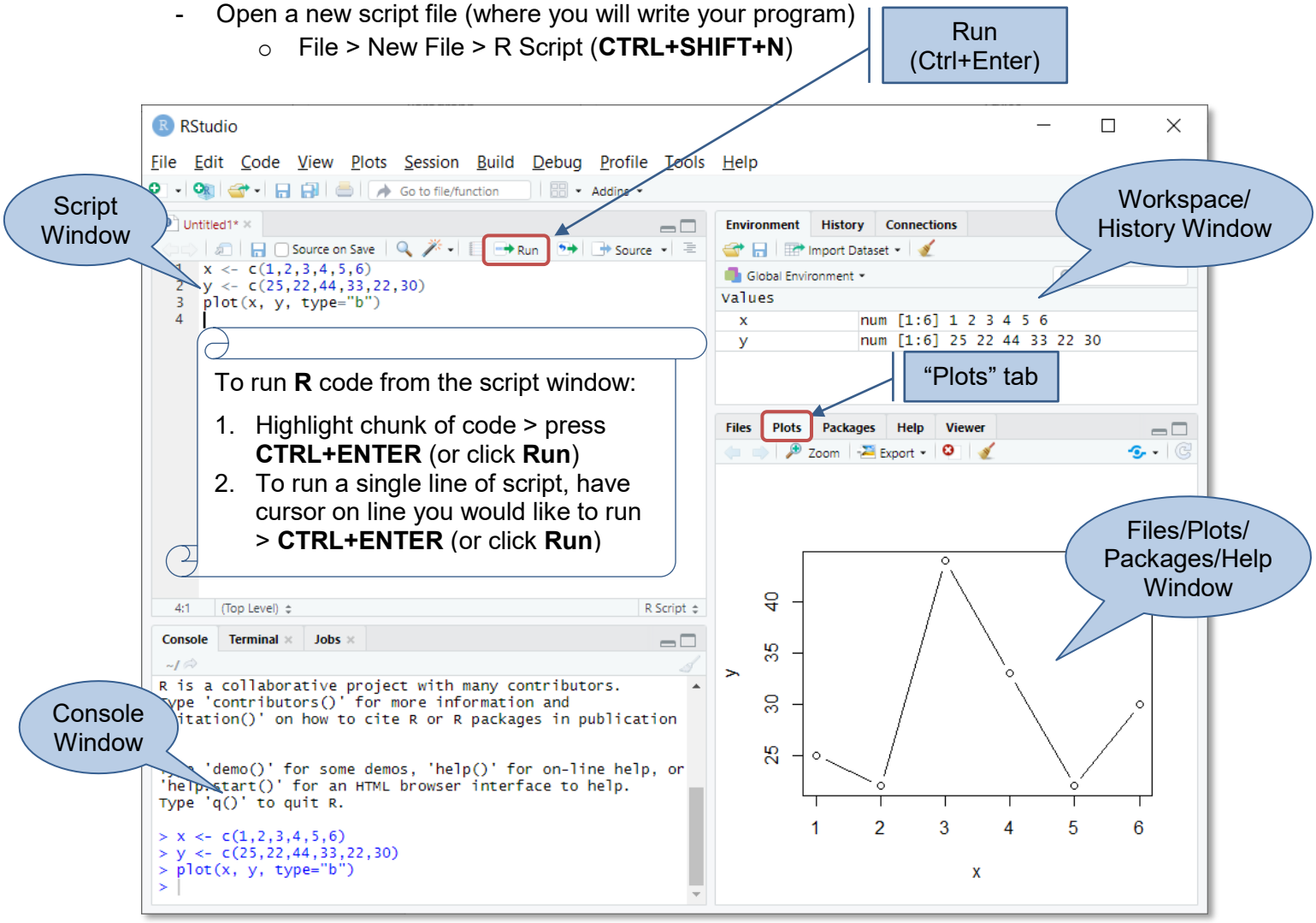


- Launching **RStudio**:



4. RStudio Windows

- Open a new script file (where you will write your program)
  - File > New File > R Script (**CTRL+SHIFT+N**)



- **Script window:** Your **R** code can be collected and saved in the script file. You can create a new script file with **CTRL+SHIFT+N.**
- **Console window:** Will contain your numeric output after you execute a command. You can also type **R** commands after the ">" in the Console and **R** will execute the command.
- **Workspace window:** Lists objects that R has in its memory.
- **Files/Plots/Packages:** You can view plots, install/load packages, use the help features.

✓ **Note:** To open an existing program in **RStudio**: File > Open File. Or right-click on the R-script file: Open with… > **RStudio**

5. Working Directory

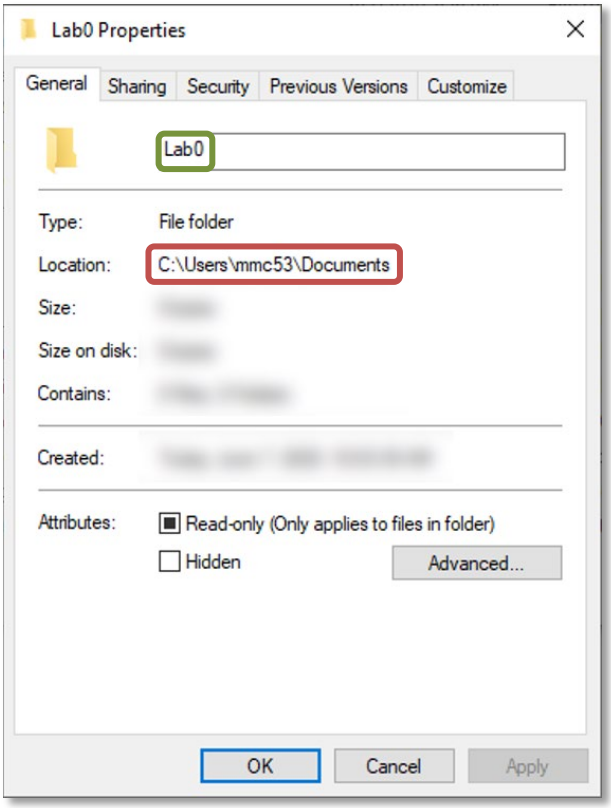
- Your **working directory** (WD) is the folder on your computer in which you are currently working. When you ask **R** to open a file, it will look in your WD by default. When you tell **R** to save a file, it will save it in the WD.
- When setting your WD, use either single or double quotes around the full path. Use either use **forward slashes (/)** or **double back slashes (\\)** in the path.

```
> setwd("C:/Users/mmc53/Documents/Lab0")
```

or

```
> setwd("C:\\Users\\mmc53\\Documents\\Lab0")
```

✓ **Note:** To find a path of a folder (**Windows**), right-click the FOLDER > Properties and copy the folder Location.



✓ **Note:** To find a path of a folder (**Mac**), click on the FOLDER > ⌘+I (Get info) and copy the path after "Where:"

II. Examples of R Commands

1. R as a Calculator

- **R** can be used as a calculator. You may type commands directly into the console (window with "**>**") or run the commands from the script window. If you want to save the code, type your commands in the script window and save your script file.
- **Parentheses** ( ) are used to separate operations.

Arithmetic Operators		Logical Operators and Comparisons	
Operator	**Description**	**Operator**	**Description**
+	addition	**<**	less than
-	subtraction	**<=**	less than or equal to
*****	multiplication	**>**	greater than
/	division	**>=**	greater than or equal to
^ or ******	exponentiation	**==**	exactly equal to
		!=	not equal to
		!x	Not x
		x | y	x OR y
		x & y	x AND y

```
> 2^2+6
> 2^(2+6)
> 4>2
```

- If you use parentheses and are missing a closing parenthesis, a "+" will appear on the command line in the console. Use **ESC** to exit the calculation and return to ">" on the command line.

```
> (7+9)*(12-3-
+
```

2. Creating Objects and Assigning Values

- Objects are assigned values using the arrow <- or the equal sign =
- Object names must begin with a **letter**
- The following command assigns the number 5 to the object a and the number 3 to z

```
> a <- 5
> z = 3
```

- Check the content of an object by entering the name of the object on the command line. You can also perform calculations using objects once they've been created.

```
> a <- 5
> a
[1] 5
```

```
> a+2
[1] 7
```

```
> b <- a*7
> b
[1] 35
```

✓ **Note:** **R** is case sensitive – that is, **R** treats `b` and `B` as different objects

```
> B
Error: object 'B' not found
```

3. Scalars, Vectors, Matrices, and Data Frames

- **R** has data structures (vectors, matrices, arrays, data frames) that you can operate on using functions that perform statistical analyses and create graphs.

- **Scalars**: Single numbers
  - `a`, `z`, and `b` (defined above) are scalars

```
> hours <- 40
> weeks <- 52
```

- **Vectors**: Row of numbers (1-dimensional). Can be numerical, character, or logical.
  - To define a vector, use the `c()` function (concatenate) with values separated by a comma

```
> employee <- c("John", "Pete", "Ben", "Mary") # character vector
> hourly <- c(12, 15, 22, 16)                  # numeric vector
> grad <- c(FALSE, TRUE, TRUE, TRUE)           # logical vector
```

  - Individual elements of a vector can be accessed using **indexing** `[i]`

```
> employee[2]
[1] "Pete"
> employee[2] = "Peter"    # changing 2nd element of employee vector
> employee
[1] "John"  "Peter" "Ben"   "Mary"
```

✓ **Note:** Begin a **comment** using `#`

- The bracket operator in **R** `[ ]` is used to choose elements of a vector or data frame that meet certain conditions. Here, we want to identify the elements of the vector `hourly` that are greater than 15:

```
> hourly[hourly>15]
[1] 22 16
```

✓ **Note:** When I see brackets, I read them as "**where**." For example, I'm asking **R** to give me the values of hourly *where* hourly is greater than 15.

✓ **Note:** If you multiply a scalar and a vector, the scalar will be multiplied by **each element** of the vector.

```
> hours*hourly
[1] 480 600 880 640
```

- **Matrices**: Table (2-dimensional; used for data that is of the same type (e.g., all numeric or all character)). Useful for linear algebra-type operations.

- **Data Frame**: Table (each column is a different variable, can be of different types (e.g., character, numeric)). When you think of a data set, you typically think of a **data frame**. You will import data as a data frame and we will perform analyses on variables in the data frame. Vectors within a data fame must be of the same length.
  - To define a data frame with vectors (3,4,5) and (1,2,3), use the `data.frame()` function with the vector names separated by a comma

```
> wages <- data.frame(employee, hourly)
> wages
  employee hourly
1     John     12
2    Peter     15
3      Ben     22
4     Mary     16
```

  - Individual rows of a data frame can be accessed using **indexing** `dfname[row#,]`
  - Individual columns of a data frame can be accessed using **indexing** `dfname[,column#]`
    - More on this in Section III (4).

```
> wages[1,]  # first row of data frame wages
  employee hourly
1     John     12
> wages[,1]  # first column of data frame wages
[1] John  Peter Ben   Mary
```

✓ **Note:** All objects that exist in your current session are listed in the upper right (workspace) window in **RStudio**. Alternatively, run `ls()` to view a list of objects that exist in your session. To clear/delete all objects, run `rm(list=ls()).`

```
> ls()
[1] "a"         "annual"    "b"         "employee"    "faculty"
[6] "grad"      "hourly"    "hours"     "student"     "studentfacultyratio"
[11] "wages"    "weeks"     "z"
> rm(list=ls())
```

4. Missing Values

- In **R**, missing values are represented by the symbol NA (not available). Impossible values (e.g., dividing by zero) are represented by the symbol NaN (not a number).

```
> employeeM <- c("John", "Pete", "Ben", NA)
> hourlyM <- c(12, 15, NA, 16)
> gradM <- c(FALSE, NA, TRUE, TRUE)
> wagesM <- data.frame(employeeM, hourlyM, gradM)
> wagesM
  employeeM hourlyM gradM
1      John      12 FALSE
2      Pete      15    NA
3       Ben      NA  TRUE
4      <NA>      16  TRUE
> annualM <- hourlyM*hours*weeks
> annualM
[1] 24960 31200    NA 33280
```

5. Functions

- We can manually compute the average annual salary of our workers:

```
> annual
[1] 24960 31200 45760 33280
> (24960+31200+45760+33280)/4
[1] 33800
```

- OR we can use built-in **R** functions to perform common mathematical operations:

```
> avgsal <- mean(annual)
> avgsal
[1] 33800
```

- **Common mathematical functions:**

Function	Description
abs(***x***)	absolute value |x|
sqrt(***x***)	square root $\sqrt{x}$
ceiling(***x***)	ceiling(3.475) gives 4
floor(***x***)	floor(3.475) gives 3
trunc(***x***)	trunc(5.99) gives 5
round(***x***, digits=***n***)	round(3.475, digits=2) gives 3.48
log(***x***)	natural logarithm ln(x)
exp(***x***)	e^x

- **Other useful functions:**

Function	Description

seq(***from***, ***to***, ***by***)
seq(***from***, ***to, length***)
	Generate a sequence
	`> indices <- seq(1, 10, by=2)`
	`> indices`
	`[1] 1 3 5 7 9`
	Shortcut: `1:10` gives sequence from 1 to 10 by 1

rep(***x***, ***ntimes***)	Repeat *x n* times
	`> y <- rep(1:3, 2)`
	`> y`
	`[1] 1 2 3 1 2 3`

is.na(***x***)	Returns TRUE if *x* (or any element of *x*) is missing
	`> is.na(annualM)`
	`[1] FALSE FALSE TRUE FALSE`

sort(***x***)	Sorts vector from smallest to largest `> sort(x, decreasing=TRUE) # largest to smallest`

unique(***x***)	Vector of unique values of ***x***
	`> unique(y)`
	`[1] 1 2 3`

length(***x***)	Length of a vector
	`> length(y)`
	`[1] 6`

class(***x***)	Determine the class of an object or its "internal" type.
	`> class(y)`
	`[1] "integer"`

- **Common statistical functions:**

  - In each function below, specify the option `na.rm=TRUE` to strip missing values before performing calculation if missing values are present in a variable.

Function	Description
mean(***x***)	Mean of object x
var(***x***)	Variance
sd(***x***)	Standard deviation
median(***x***)	Median
quantile(***x***, ***probs***)	Quantiles where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0,1].
range(***x***)	Range
sum(***x***)	Sum
min(***x***)	Minimum
max(***x***)	Maximum
cumsum(***x***)	Cumulative sum
summary(***x***)	Summary statistics (min, 1st quartile, median, mean, 3rd quartile, max)
sum(!is.na(***x***))	Number of non-missing values

```
> annualM
[1] 24960 31200    NA 33280
> mean(annualM)
[1] NA
> mean(annualM, na.rm=TRUE)
[1] 29813.33
```

- **Statistical probability functions:**

  - The following table describes functions related to probability distributions.

Function	Description
dnorm(***x***)	Normal density function at *x* (by default from N(mean=0 sd=1))

pnorm(***q***)	Cumulative normal probability for *q* (i.e., P(Z ≤ *q*) = *p*)
	`> pnorm(1.96)`
	`[1] 0.9750021`

qnorm(***p***)	Normal quantile (i.e., value at the *p* percentile of N(0,1) distribution)
	`> qnorm(.95)`
	`[1] 1.644854`

rnorm(***n***, mean=**0**,sd=**1**)	Generate *n* observations (random normal deviates) from a Normal distribution with mean `mean` and standard deviation `sd`.
	`# 50 random normal variates from N(mean=50, sd=10)`
	`> x <- rnorm(50, mean=50, sd=10)`

dbinom(***x***, ***size***, ***prob***)
pbinom(***q***, ***size***, ***prob***)
qbinom(***p***, ***size***, ***prob***)
rbinom(***n***, ***size***, ***prob***)
	Binomial(*n*,*p*) where `size` is the number of trials (*n*) and `prob` is the probability of a success (*p*).
	`# prob of 0 to 5 heads of fair coin out of 10 flips`
	`> dbinom(0:5, 10, .5)`
	`# prob of <= 5 heads out of 10 flips of fair coin`
	`> pbinom(5, 10, .5)`

dunif(***x***, min=**0**, max=**1**)
punif(***q***, min=**0**, max=**1**)
qunif(***p***, min=**0**, max=**1**)
runif(***n***, min=**0**, max=**1**)
	Uniform(min, max), follows the same pattern (default is Uniform(0,1))
	`# 10 values from Uniform(0,1)`
	`> x <- runif(10)`

III. Working with Existing Data

1. Importing Data

- Read data into **R** using the **read.table()** function, creating a data frame. Specify the following:
  1. The **location** of the file — typically this is a file on your file system
     - If the file is in your working directory, you do not have to specify the file path; just specify the file name and extension in quotes
     - If the file is **not** in your working directory, you must specify the file path and file name using either use forward slashes (/) or double back slashes (\\) in the path.
  2. The `sep` argument indicates the character used to separate items in the data file. Commonly used separators:
     - Comma separated (CSV), `sep=","`
     - Tab separated, `sep="\t"`
  3. The `header` argument indicates whether the first line in the data file contains variable names.

```
# No path: data file in my working directory
> mydata <- read.table("hgb.csv", header=TRUE, sep=",")
# Data file not in my working directory
> mydata <- read.table("C:\\Users\\hgb.csv", header=TRUE, sep=",")
```

- Read a CSV file into **R** using the function **read.csv()**

```
# No path: CSV file in my working directory
> mydata <- read.csv("hgb.csv", header=TRUE)
```

- Where's my working directory, again?

```
> getwd()
[1] "C:/Users/mmc53/Documents/Lab0"
```

2. Structure of the Data Frame

- View the structure of your data set and a list of the variables and variable types using **str**(object name):

```
> str(mydata)
'data.frame':    979 obs. of  13 variables:
 $ id       : int  1 2 3 4 5 6 7 8 9 10 ...
 $ group    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ age      : int  26 23 24 26 28 25 31 22 21 25 ...
 $ edyrs    : int  17 13 19 13 18 13 15 14 20 19 ...
 $ income   : num  2.18 2.21 2.21 2.21 2.23 ...
 $ wt0      : num  154 158 132 168 186 ...
 $ wt1      : num  196 200 165 206 226 ...
 $ parity   : int  1 0 1 2 3 1 2 2 1 3 ...
 $ prenatal: int  0 0 1 1 0 0 1 1 1 1 ...
 $ psmoke   : int  1 0 1 0 0 0 0 0 1 0 ...
 $ hgb9     : num  9.74 10 10.35 11.28 10.68 ...
 $ hgb36    : num  6.89 7.7 7.51 8.43 7.49 ...
 $ water    : num  299 267 308 293 302 ...
```

- View the **dimensions** (# of rows, # of columns) of your data frame with **dim**(dfname)
- View the **number of rows** of your data frame with **nrow**(dfname), view number of columns with **ncol**(dfname)
- View the **first** few lines of data using **head**(dfname, n=2)
- View the **last** few lines of data using **tail**(dfname, n=2)
  - Note: If n omitted, head() and tail() default to printing 6 lines of data

```
> dim(mydata)  # dimension of data frame (no. rows, no. columns)
[1] 979  13
> nrow(mydata) # number of rows (observations) of a data frame
[1] 979
> ncol(mydata) # number of columns (variables) of a data frame
[1] 13
> head(mydata, n=5) # first 5 lines
  id group age edyrs income   wt0   wt1 parity prenatal psmoke  hgb9 hgb36 water
1  1     1  26    17  2.176 154.2 196.5      1        0      1  9.74  6.89 298.8
2  2     1  23    13  2.207 158.0 199.5      0        0      0 10.00  7.70 267.1
3  3     1  24    19  2.207 131.7 164.7      1        1      1 10.35  7.51 308.4
4  4     1  26    13  2.215 167.6 206.5      2        1      0 11.28  8.43 292.9
5  5     1  28    18  2.226 186.5 225.7      3        0      0 10.68  7.49 301.6
> tail(mydata, n=5) # last 5 lines
     id group age edyrs income   wt0   wt1 parity prenatal psmoke  hgb9 hgb36 water
975 975     3  28    12 10.785 159.9 194.2      0        1      0 12.10  9.25 435.2
976 976     3  27    12 10.892 146.3 189.1      2        1      0 10.81  7.07 266.7
977 977     3  24    12 11.220 116.6 151.4      1        1      0 10.46  7.58 261.1
978 978     3  23    12 11.809 140.2 179.1      2        0      0 10.31  6.41 379.4
979 979     3  25    12     NA 133.5 178.8      1        1      0 10.54  9.48  88.6
```

3. Accessing Variables in a Data Frame

- Print the **variable names** of the variables in your data set with the **names()** function:

```
> names(mydata)
 [1] "id"       "group"    "age"      "edyrs"    "income"
 [6] "wt0"      "wt1"      "parity"   "prenatal" "psmoke"
[11] "hgb9"     "hgb36"    "water"
```

- Let's find the average age of the individuals in our data set:

```
> mean(age)
Error in mean(age) : object 'age' not found
```

  - What happened? `age` is not an object on its own. It is a column in the data frame, `mydata`.

- Access individual variables in a data frame using the `$` (e.g., `mydata$wtgain`)

```
> summary(mydata$hgb9)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   8.65   10.71   11.18   11.18   11.67   13.30
> summary(mydata$hgb36)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  5.130   7.400   8.200   8.226   9.090  11.690
```

4. Sorting a Data Frame

- Sort a data frame using the **order()** function. By default, sorting is ascending. Prepend the sorting variable by a minus sign to request sorting in descending order.

```
> mydatasort1 <- mydata[order(mydata$age),]    # ascending
> mydatasort2 <- mydata[order(-mydata$age),]   # descending
# Sort on both age and edyrs
> mydatasort3 <- mydata[order(mydata$age, mydata$edyrs),]
> head(mydatasort3)
     id group age edyrs income   wt0   wt1 parity prenatal psmoke  hgb9 hgb36 water
93   93     1  19    11  2.911 105.3 135.1      1        0      1 11.73  7.88 497.1
255 255     1  20    12  6.213 135.7 174.2      3        1      0 12.43  9.70 325.7
104 104     1  21     8  3.083 161.6 192.6      1        0      0 11.67  9.11 467.1
752 752     3  21     8  3.146 127.1 161.6      1        1      0 10.80  7.52 410.1
13   13     1  21    11  2.282 184.2 214.3      1        0      0 10.80  7.53 347.4
88   88     1  21    11  2.866 188.1 222.1      2        0      0 10.87  7.24 398.5
```

5. Subsetting a Data Frame

- Subset a data frame **by rows** using **indexing**. `mydata[c(1,2,3),]` selects **rows** 1, 2, and 3 from `mydata`.

```
> mydataFirst10rows <- mydata[1:10,]
> mydataFirst10rows
   id group age edyrs income   wt0   wt1 parity prenatal psmoke  hgb9 hgb36 water
1   1     1  26    17  2.176 154.2 196.5      1        0      1  9.74  6.89 298.8
2   2     1  23    13  2.207 158.0 199.5      0        0      0 10.00  7.70 267.1
3   3     1  24    19  2.207 131.7 164.7      1        1      1 10.35  7.51 308.4
4   4     1  26    13  2.215 167.6 206.5      2        1      0 11.28  8.43 292.9
5   5     1  28    18  2.226 186.5 225.7      3        0      0 10.68  7.49 301.6
6   6     1  25    13  2.230 138.2 175.3      1        0      0 13.16 10.08 288.0
7   7     1  31    15  2.231 165.6 219.0      2        1      0 10.50  7.29 367.9
8   8     1  22    14  2.236 121.2 153.5      2        1      0 11.51  8.29 355.6
9   9     1  21    20  2.242 132.8 162.8      1        1      1 10.28  6.64 489.1
10 10     1  25    19  2.246 170.2 210.3      3        1      0 10.43  7.19 441.7
```

- Subset a data frame **by columns** using **indexing**. `mydata[,c(1,2)]` selects columns 1 and 2 from `mydata`. May also call the individual variable names, as shown below:

```
> mydataFirst2columns <- mydata[,c(1,2)]
> names(mydata)
 [1] "id"       "group"    "age"      "edyrs"    "income"
 [6] "wt0"      "wt1"      "parity"   "prenatal" "psmoke"
[11] "hgb9"     "hgb36"    "water"
> mydatasubset <- mydata[,c("id", "group")]
> head(mydatasubset, n=3)
  id group
1  1     1
2  2     1
3  3     1
```

- The `subset()` function is the easiest way to select variables and to select observations based on **variable conditions**. In the following example, we select all rows that have a value of age greater than or equal to 20 and less than or equal to 25 (age between 20-25, inclusive). `newdata` contains only the variables `group` and `age`. Omitting the `select` option includes all variables in the new subset.

```
# Option 1: subset() function
> newdata <- subset(mydata, age >= 20 & age <= 25, select=c(group,
age))
> dim(newdata)
[1] 460   2
> summary(newdata$age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  20.00   24.00   24.00   24.12   25.00   25.00
```

- You can use the brackets in **R** `[ ]` to choose a subset of **rows** in the data set based on specified conditions. Here, we are choosing the rows (observations) where age is ≥ 20 and ≤ 25:

```
# Option 2: bracket operator
> newdata2 <- mydata[mydata$age >= 20 & mydata$age <= 25,]
> summary(newdata2$age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  20.00   24.00   24.00   24.12   25.00   25.00
```

6. Creating New Variables in a Data Frame

- Create new variables and add them to the data frame using `mydata$newvariable`:

```
> mydata$wtgain = mydata$wt1 - mydata$wt0
> names(mydata)
[1] "id"      "group"    "age"     "edyrs"    "income"   "wt0"      "wt1"
[8] "parity"  "prenatal" "psmoke"  "hgb9"     "hgb36"    "water"    "wtgain"
> summary(mydata$wtgain)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 24.50   37.70   40.90   40.75   43.90   63.90
> mydata[1:10, c("wt0", "wt1", "wtgain")]  # print rows 1-10
     wt0   wt1 wtgain
1  154.2 196.5   42.3
2  158.0 199.5   41.5
3  131.7 164.7   33.0
4  167.6 206.5   38.9
5  186.5 225.7   39.2
6  138.2 175.3   37.1
7  165.6 219.0   53.4
8  121.2 153.5   32.3
9  132.8 162.8   30.0
10 170.2 210.3   40.1
```

- When recoding a variable, can identify which entries of the vector meet our criteria for recoding using the `[ ]` notation (e.g., identify ages values that are > 30)

```
> mydata$age[mydata$age > 30]
[1] 31 31 31 31 31 31
```

- Then recode those values to be equal to the desired value (e.g., 30). In this example, we want to cap everyone's age at 30. If anyone is > 30, they will have their age value set to equal 30.

```
> mydata$age[mydata$age > 30] <- 30
```

- This is commonly used when recoding **missing values**. Missing values sometimes come coded as 99. R will read the value 99 as an actual value. We need to recode 99 to NA.

```
> dataname$v1[dataname$v1 == 99] <- NA
```

Notice the double == (logical operator, Section II.1)

7. Exporting a Data Frame to CSV

- Export a data frame to a CSV file using the **write.csv()** function. First, specify the name of the data frame to be exported (mydata, here), then specify the path and file name.csv enclosed in quotes. row.names = FALSE will prevent the automatic row numbers from being exported as the first column of the CSV file.

```
# Modify path to point to your destination
> write.csv(mydata, " C:\\Users\\Lab0\\saveddata.csv", row.names =
FALSE)

# Will default to your working directory
> write.csv(mydata, "saveddata2.csv", row.names = FALSE)
```

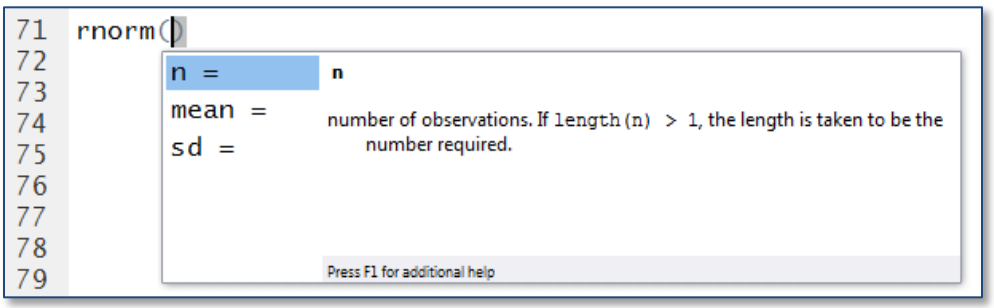
IV. Additional Topics

1. Getting Help

- **R** has built-in help files

```
> ?rnorm
> help(rnorm)
> example(rnorm)
```

- **RStudio** will show possible function arguments when you press **TAB** after the function name and parentheses:



- Good resources:
  - http://cran.r-project.org/doc/manuals/R-intro.pdf
  - http://cran.r-project.org/doc/contrib/Short-refcard.pdf

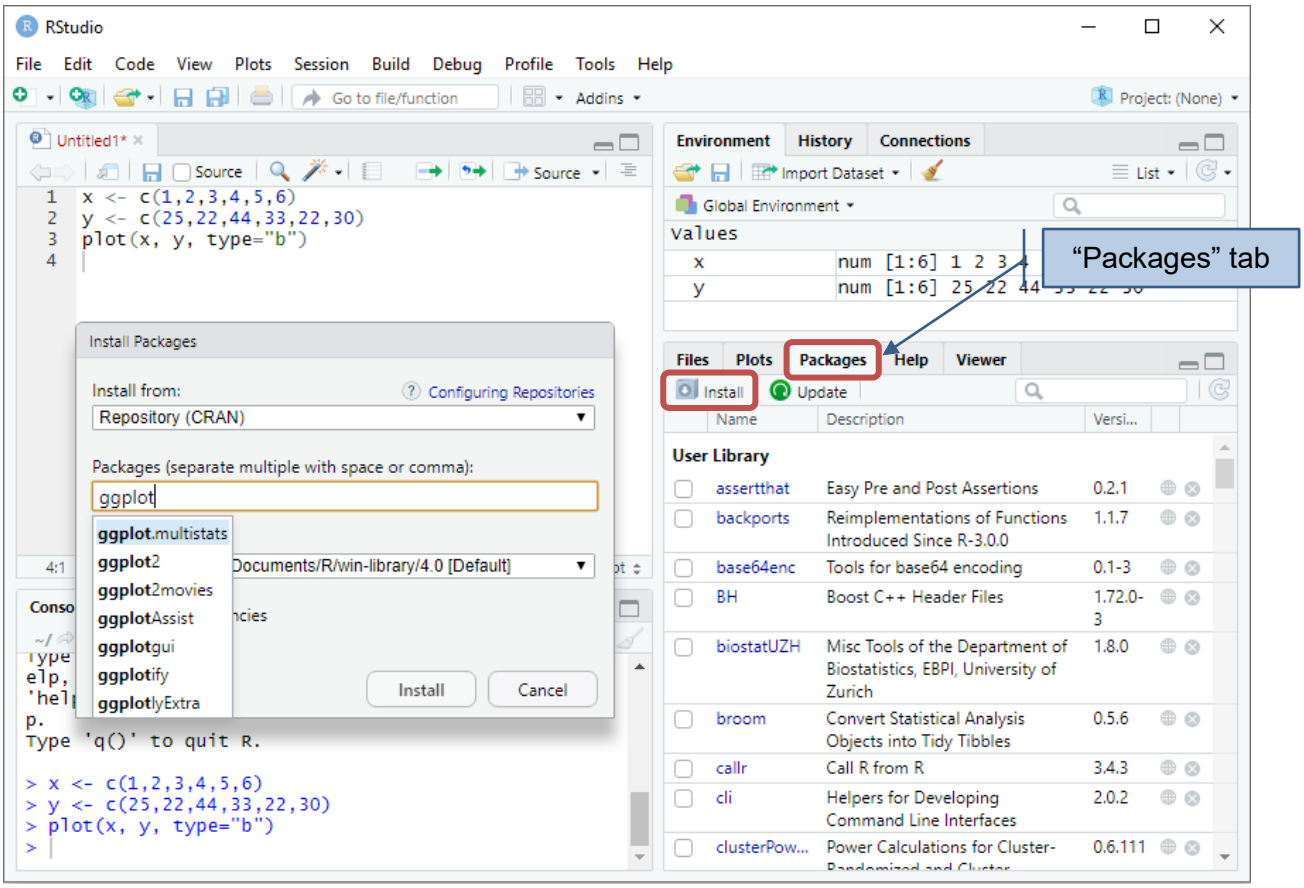
2. Libraries

- Packages are collections of **R** functions and data. A standard set of packages are included with **R**. The packages that you have installed are listed in your user `library`.
- **R** also has many user-created packages that contain specialized techniques (which is partly why **R** is so popular).
- To use the functions/data in a new package, you must:

1. Install the package to your computer (once)
2. Load the library (every time you re-start R and want to call functions from the package)

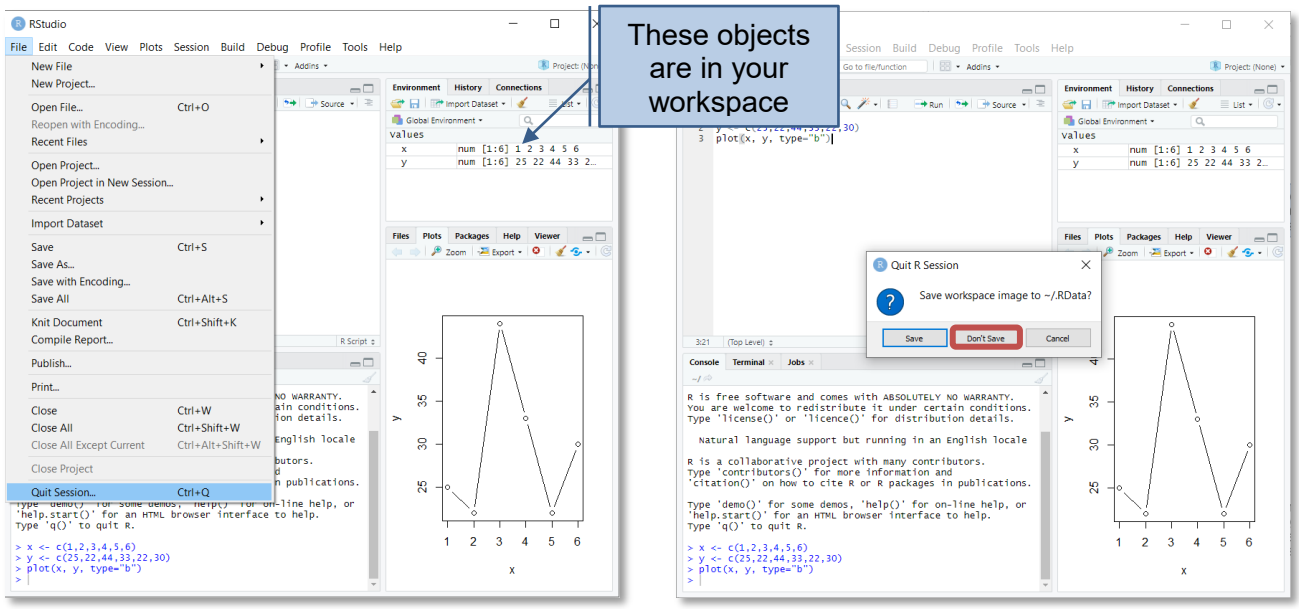
- To install the package, click **Install** in the **Packages** window and type `ggplot2` or run the command `install.packages("ggplot2")`
- To load the package: type `library(ggplot2)`

```
> install.packages("ggplot2")   # run only once
> library(ggplot2)              # run in every new RStudio session
```



3. Exiting RStudio

- Be sure to **save** your .R script file before exiting **RStudio** (File > Save)
- Exit **RStudio** by quitting the application or File > Quit Session



- You will be asked if you would like to **save your workspace**
- The **workspace** is your current R working environment and includes any objects defined in your session (vectors, matrices, data frames, lists, functions). You can save an image of the current workspace that is automatically reloaded the next time **RStudio** is started. I generally do **not** recommend doing this, but instead re-running the code to re-generate the objects in your next session.