Introduction to R and RStudio

Panes in RStudio

When you launch RStudio, the program opens with four panes. Beginning in the lower left corner and moving clockwise, these panes are

- **Console**: Provides access to R itself; this is where you enter commands when working on problems.
- **Source Pane**: Provides access to a variety of different types of documents, including script files (.R) and rmarkdown files (.Rmd), and access to functions to run lines of R code in a script by sending them to the console, or to convert an rmarkdown files into a PDF or an HTML format.
- **Environment & History Pane**: Provides access to the data and functions you create while using R, and to a history of the commands used in your R session.
- **Files, Plots, Packages, Help & Viewer Pane**: Provides access to your computer's file structure, to help files for R commands, to a list of R packages available to you (packages provide access to additional commands beyond those available to you when you first launch R), to plots that you create, and to an internal web-like browser.

Communicating With R

The symbol `>` in the console is the command prompt, which indicates that R is awaiting your instructions; when you type a command and hit `Enter` or `Return`, R executes the command and returns any appropriate output; thus, this command adds together the numbers 1 and 3 and returns the number 4 as an answer

```
1 + 3
```

```
## [1] 4
```

If we assign this calculation to an object

```
answer = 1 + 3
```

then the result of the calculation is assigned to the object but is not returned to us; to see an object's value we can look for it in RStudio's **Environment Panel** or enter the object's name as a command in the **Console**.

```
Answer = 2 + 4
answer
```

```
## [1] 4
```

```
Answer
```

```
## [1] 6
```

Note that an object's name is case-sensitive so that `answer` and `Answer` are different objects.

Objects for Storing Data

A **vector** is an ordered collection of elements of the same type, which may be numerical values, integer values, logical values, or character strings. Note that "ordered" does not imply an ordering in terms of rank—smallest-to-largest or largest-to-smallest—or an alphabetical ordering; it simply means the vector's elements are given in the order in which we entered them into the object, which, for us, usually is the order in which the data was collected.

```
v00 = c(1.1, 2.2, 3.3)
v00
```

```
## [1] 1.1 2.2 3.3
```

```
v01 = c(1, 2, 3)
v01
```

```
## [1] 1 2 3
```

```
v02 = c(TRUE, TRUE, FALSE)
v02
```

```
## [1]  TRUE  TRUE FALSE
```

```
v03 = c("alpha", "bravo", "charley")
v03
```

```
## [1] "alpha"   "bravo"   "charley"
```

You can view an object's structure by examining it in the **Environment Panel** or by using R's structure command: `str( )`

```
str(v02)
```

```
##  logi [1:3] TRUE TRUE FALSE
```

which, for vector v02 returns that it is a logical vector with an index for its entries of 1, 2, and 3, and with values of `TRUE`, `TRUE`, and `FALSE`.

We can use a vector's index to correct errors, to add new values, or to create a new vector using already existing vectors.

```
v03[3] = "charlie"
v03
```

```
## [1] "alpha"   "bravo"   "charlie"
```

```
v00[4] = 4.4
v00
```

```
## [1] 1.1 2.2 3.3 4.4
```

```
v04 = c(v01[1], v02[2], v03[3])
v04
```

```
## [1] "1"       "TRUE"    "charlie"
```

Here are some other ways to create a vector when its entries follow a defined sequence or use a repetitive pattern.

```
v05 = seq(from = 0, to = 20, by = 4)
v05
```

```
## [1]  0  4  8 12 16 20
```

```
v06 = seq(0, 10, 2)
v06
```

```
## [1]  0  2  4  6  8 10
```

```
v07 = rep(1:4, times = 2)
v07
```

```
## [1] 1 2 3 4 1 2 3 4
```

```
v08 = rep(1:4, each = 2)
v08
```

```
## [1] 1 1 2 2 3 3 4 4
```

Note that 1:4 is equivalent to c(1, 2, 3, 4). Finally, we can complete mathematical operations using vectors, make logical inquires, and create sub-samples.

```
v09 = v08 - v07
v09
```

```
## [1]  0 -1 -1 -2  2  1  1  0
```

```
v10 = v09 == 0
v10
```

```
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
v11 = which(v09 < 1)
v11
```

```
## [1] 1 2 3 4 8
```

```
v12 = v09[!v09 < 1]
v12
```

```
## [1] 2 1 1
```

A **data.frame** is a collection of vectors, all equal in length but not necessarily of a single type of element, arranged, spreadsheet-like, with the vectors as columns.

```
df01 = data.frame(v07, v08, v09, v10)
df01
```

```
##   v07 v08 v09   v10
## 1   1   1   0  TRUE
## 2   2   1  -1 FALSE
## 3   3   2  -1 FALSE
## 4   4   2  -2 FALSE
## 5   1   3   2 FALSE
## 6   2   3   1 FALSE
## 7   3   4   1 FALSE
## 8   4   4   0  TRUE
```

We can access the elements in a data.frame using the data.frame's index, which takes the form

[row number, column number]

```
v13 = df01[1, ]
v13
```

```
##   v07 v08 v09  v10
## 1   1   1   0 TRUE
```

```
v14 = df01[ , 3:4]
v14
```

```
##   v09   v10
## 1   0  TRUE
## 2  -1 FALSE
## 3  -1 FALSE
## 4  -2 FALSE
## 5   2 FALSE
## 6   1 FALSE
## 7   1 FALSE
## 8   0  TRUE
```

```
v15 = df01[4, 3]
v15
```

```
## [1] -2
```

```
v16 = df01$v08
v16
```

```
## [1] 1 1 2 2 3 3 4 4
```

A **matrix** is similar to a data.frame, but every element in a matrix is of the same type, usually numerical.

```
m01 = matrix(1:10, nrow = 5)
m01
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

```
m02 = matrix(1:10, ncol = 5)
m02
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

A **list** is an object that holds other objects, even if those objects are of different types.

```
li01 = list(v00, df01, m01)
li01
```

```
## [[1]]
## [1] 1.1 2.2 3.3 4.4
##
## [[2]]
##   v07 v08 v09   v10
## 1   1   1   0  TRUE
## 2   2   1  -1 FALSE
## 3   3   2  -1 FALSE
## 4   4   2  -2 FALSE
## 5   1   3   2 FALSE
## 6   2   3   1 FALSE
## 7   3   4   1 FALSE
## 8   4   4   0  TRUE
##
## [[3]]
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

Managing Your Environment

One nice feature of RStudio is that the **Environment Panel** provides a list of the objects (and functions: more on these later) you have created. If your environment becomes too cluttered, you can delete items by switching to the **Grid** view, clicking on the check-box next to the object(s) you wish to delete, and then clicking on the **broom** icon. You can remove all items from the **List** view by simply clicking on the **broom** icon. You also can view and remove objects from the console using the `ls( )` and the `rm( )` commands.

```
ls()
```

```
##  [1] "answer" "Answer" "df01"   "l01"    "m01"    "m02"    "v00"
##  [8] "v01"    "v02"    "v03"    "v04"    "v05"    "v06"    "v07"
## [15] "v08"    "v09"    "v10"    "v11"    "v12"    "v13"    "v14"
## [22] "v15"    "v16"
```

```
rm(df01)
ls()
```

```
##  [1] "answer" "Answer" "l01"    "m01"    "m02"    "v00"    "v01"
##  [8] "v02"    "v03"    "v04"    "v05"    "v06"    "v07"    "v08"
## [15] "v09"    "v10"    "v11"    "v12"    "v13"    "v14"    "v15"
## [22] "v16"
```

```
rm(list = ls())
ls()
```

```
## character(0)
```

Reading in a Data File and Saving a Data File

Although creating a small vector, data.frame, matrix, or list is easy, creating one with hundreds of elements or creating dozens of individual data objects is tedious at best; thus, the ability to load data saved from an earlier session or the ability to read in a spreadsheet file is helpful. Reading in data from a saved file is easy if you know the file's path. Suppose we have the following structure of folders and files, where folders are shown without extensions and data files with extensions:

- CourseWork
  - Chem260
    - kinetic.xlsx
  - Chem351
    - syllabus.docx
    - DataFiles
      - DyeConc.RData
      - SPS01.cvs
    - Handouts

A file's path depends on its location relative to the current working directory. For example, if the working directory is the folder *Chem351*, then the pathname for the file *syllabus.docx* is "syllabus.docx," the pathname to the file *DyeConc.RData*, which is in the *DataFiles* folder is "DataFiles/DyeConc.RData" and the pathname to the file *kinetic.xlsx*, which is in the *Chem260* folder is "../Chem260/kinetic.xlsx" where ../ indicates that the path first moves up one folder to the *CourseWork* folder; to move up two folders the path begins with ../../ followed by the remainder of the file's path.

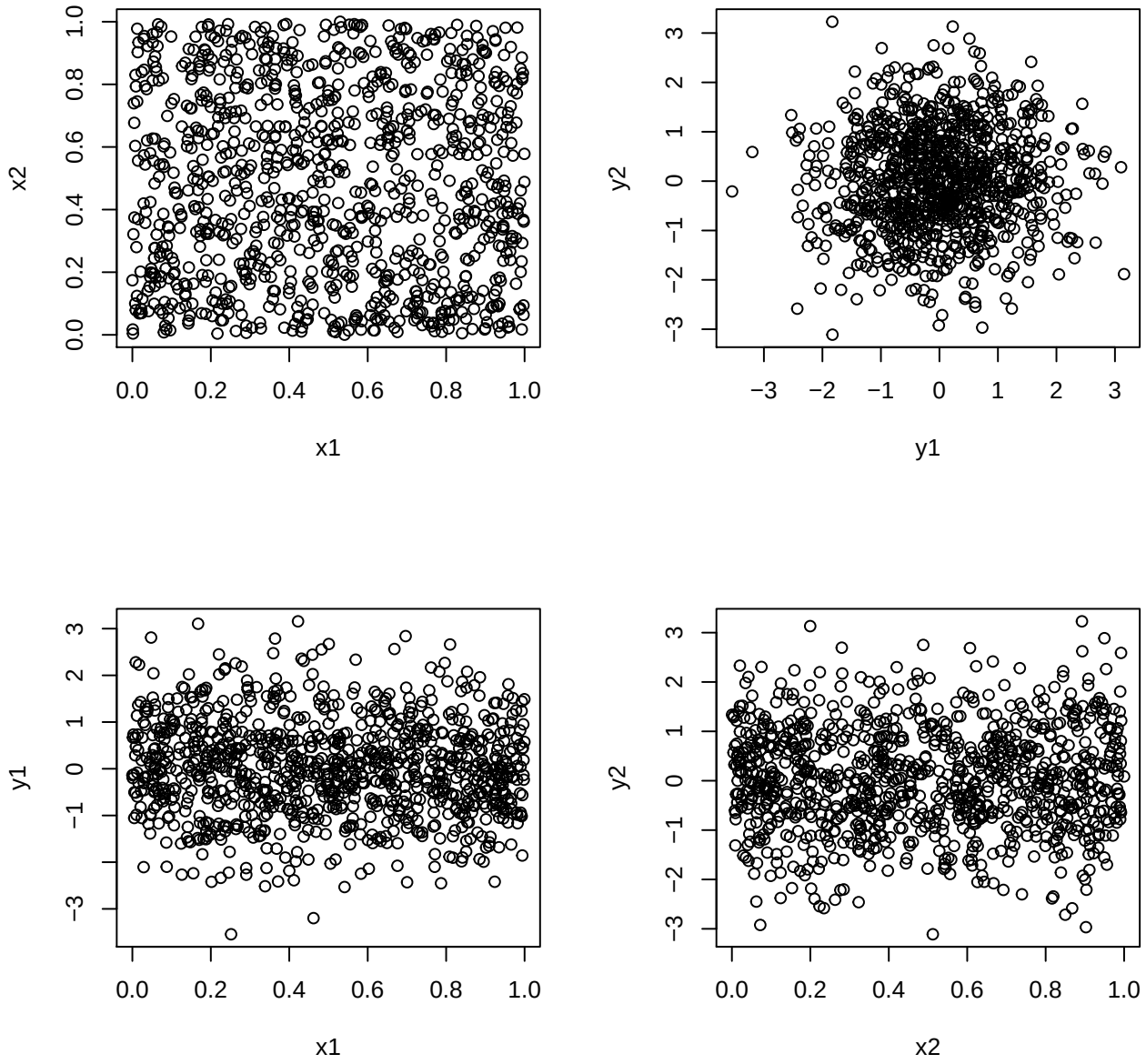
You can set your default working directory using `Prefrences` from the `RStudio` menu (Note: Your folder for this class is a good option). If you are unsure of your working directory, use the `getwd( )` command

```
getwd()
```

```
## [1] "/Users/davidharvey/Dropbox/Teaching/Chem 351"
```

To read in a `.RData` file, we use the `load( )` command, to read in a spreadsheet saved as a `.csv` file, we use the `read.csv( )` command (Note: The `read.csv( )` command assumes that the first row in the file provides the names for the columns), and to source (that is, run) an `.R` script we file we use the `source( )` command.

```
load(file = "BeefLiver.RData")
elements = read.csv(file = "ElementData.csv")
source("sampleScript.R")
```



Although not shown here, the commands `read.csv(file.choose( ))`, `load(file.choose( ))`, and `source(file.choose( ))` allow you to use a browser to find the file you wish to open. You also can load an `.RData` file or load, but not source, an `.R` script file from RStudio's **File Panel** by clicking on its name, and you can use the *Import Dataset* button in the **Environment Panel** to read in a `.csv` file.

To save one or more objects to an `.RData` file, use the `save( )` command; you can save a single object to a `.csv` file using the `write.csv( )` command. For both commands, the filename is the pathname to where you wish to store the file.

```
save(PBconc, elements, file = "saved.RData")
write.csv(elements, file = "saved.cvs")
```

Getting Help

There are extensive help files for R commands (or functions, as they sometimes are called), which you can search for using the **Help Panel** or by using the `help( )` command.

```
help(head)
```

A help file shows you the command's proper syntax, including the types of values you can pass to the command and their default values, if any—more details on this later—and provides you with some examples of how the command is used.