Introduction to Jupyter and R

October 11, 2021

1 Introduction to R/Jupyter

This is a Jupyter Notebook. A Jupyter Notebook is an "open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text." We will use the R programming languge and LaTex typsetting in Jupyter to prepare assignments and projects (you can also use other languages in Jupyter). The purpose of this note book is to familiarize yourself with programming in R.

R is a powerful open source programming language that is used for statistical computation and graphics. In many ways, it is similar to Matlab and Python (so, if you know these languages, learning R will be fairly easy!).

The best way to learn R and Jupyter is to jump right in! Below we walk through how to use Jupyter and R. **Text in boldface indicates exercises that I am asking you to complete.**

This document was made as a general introduction to R programming and Jupyter notebooks. In the context of this course, it is an optional assignment that will not be graded or checked.

[ ]:
```
#This package will be used to autograde some of your answers
library(testthat)
```

1.1 Part I: Basics

Jupyter has different types of cells. This cell is a "markdown" cell, and is used to write equations (LaTeX) and text. The other type of cell that we will use is a "code" cell. This is where we will program using R. You can create a new cell from the plus sign above (delete using the scissor) and select the type of cell using the drop down menu above and to the right.

Note:

1. In autograded assignments, don't add or delete cells. This will impact the autograder.
2. Here's a LaTeX cheatsheet.

Double click on the "Type Markdown and LaTex: a^2" and type your favorite equation (come on, you know you have one...). $a^2 + b^2 = c^2$

In R/code cells, you can do (almost) anything that you can do in R, from basic computations to complex statistical modeling.

Create an R code cell below this cell. Compute the natural log of e. You may have guessed that the right functions for this are `log()` and `exp()`. But with R, you don't have to guess. You can type `?? log` to get some general information about functions related to logarithms (the same applies to other concepts); if you type "? X" you will get information about the specific function X (e.g., try `? exp`). Often, a Google search is a really effective way to learn about R functions.

[2]:
```
#YOUR CODE HERE
log(exp(1)) # No Answer - remove if you provide an answer
```

1

The function `c()` will allow you to create vectors ("c" for combine).

In the cell below, create three vectors:

1. a vector containing the numbers 1, 2, 5.3, 6, -2, 4;
2. a vector containing the strings "one", "two, and"three" (strings are formed in R by putting quotes around the words);
3. a vector of logicals: `TRUE`, `TRUE`, `FALSE`, `TRUE`.

[ ]:
```
#YOUR CODE HERE
fail() # No Answer - remove if you provide an answer
```

Next, type `seq(1,10)` in the R cell below. What does it do? What about `seq(1, 20, by = 2)`? `seq(10, 20, len = 100)`? Why are these convenient?

[3]:
```
#YOUR CODE HERE
seq(1,10)
seq(1,20, by=2)
seq(10,20, len=100)
```

1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10

1. 1 2. 3 3. 5 4. 7 5. 9 6. 11 7. 13 8. 15 9. 17 10. 19

1. 10 2. 10.1010101010101 3. 10.2020202020202 4. 10.3030303030303 5. 10.4040404040404 6. 10.5050505050505 7. 10.6060606060606 8. 10.7070707070707 9. 10.8080808080808 10. 10.9090909090909 11. 11.010101010101 12. 11.1111111111111 13. 11.2121212121212 14. 11.3131313131313 15. 11.4141414141414 16. 11.5151515151515 17. 11.6161616161616 18. 11.7171717171717 19. 11.8181818181818 20. 11.9191919191919 21. 12.020202020202 22. 12.1212121212121 23. 12.2222222222222 24. 12.3232323232323 25. 12.4242424242424 26. 12.5252525252525 27. 12.6262626262626 28. 12.7272727272727 29. 12.8282828282828 30. 12.9292929292929 31. 13.030303030303 32. 13.1313131313131 33. 13.2323232323232 34. 13.3333333333333 35. 13.4343434343434 36. 13.5353535353535 37. 13.6363636363636 38. 13.7373737373737 39. 13.8383838383838 40. 13.9393939393939 41. 14.040404040404 42. 14.1414141414141 43. 14.2424242424242 44. 14.3434343434343 45. 14.4444444444444 46. 14.5454545454545 47. 14.6464646464646 48. 14.7474747474747 49. 14.8484848484848 50. 14.9494949494949 51. 15.0505050505051 52. 15.1515151515152 53. 15.2525252525253 54. 15.3535353535354 55. 15.4545454545455 56. 15.5555555555556 57. 15.6565656565657

58. 15.7575757575758 59. 15.8585858585859 60. 15.959595959596 61. 16.0606060606061 62. 16.1616161616162 63. 16.2626262626263 64. 16.3636363636364 65. 16.4646464646465 66. 16.5656565656566 67. 16.6666666666667 68. 16.7676767676768 69. 16.8686868686869 70. 16.969696969697 71. 17.0707070707071 72. 17.1717171717172 73. 17.2727272727273 74. 17.3737373737374 75. 17.4747474747475 76. 17.5757575757576 77. 17.6767676767677 78. 17.7777777777778 79. 17.8787878787879 80. 17.979797979798 81. 18.0808080808081 82. 18.1818181818182 83. 18.2828282828283 84. 18.3838383838384 85. 18.4848484848485 86. 18.5858585858586 87. 18.6868686868687 88. 18.7878787878788 89. 18.8888888888889 90. 18.989898989899 91. 19.0909090909091 92. 19.1919191919192 93. 19.2929292929293 94. 19.3939393939394 95. 19.4949494949495 96. 19.5959595959596 97. 19.6969696969697 98. 19.7979797979798 99. 19.8989898989899 100. 20

What does the `rep()` function do? This function can take in several arguments. To see what they are, use the help file! Explore the arguments `times` and `each`. What do they do?

[12]:
```
#YOUR CODE HERE
help('rep')
```

[10]:
```
rep(1:4, each = 2, times = 3)
```

1. 1 2. 1 3. 2 4. 2 5. 3 6. 3 7. 4 8. 4 9. 1 10. 1 11. 2 12. 2 13. 3 14. 3 15. 4 16. 4 17. 1 18. 1 19. 2 20. 2 21. 3 22. 3 23. 4 24. 4

You can store vectors in R using $=$ or $<-$.

Store the numbers 1 through 5 in a vector `v1`. Create another vector, `v2`, and join it together in a new vector, `new`. *Note: This will be an autograded answer. The autograder will check whether you store the correct values in `v1`.*

[6]:
```
#assign 1, 2, 3, 4, 5 to the variable v1
#YOUR CODE HERE
v1  = c(1,2,3,4,5)
#accessing data stored in variable v1
v1
#create two vectors, then join them together in a new vector
v2 = c(9, 10, 1)
new = c(v1, v2)
new
```

1. 1 2. 2 3. 3 4. 4 5. 5

1. 1 2. 2 3. 3 4. 4 5. 5 6. 9 7. 10 8. 1

[7]:
```
# Test Cell
```

Factors, also known as categorical/qualitative variables, are important in statistics. "Shoe color" (Black, Brown, etc.), "drink size" (S, M, L) and "espresso roast" (Light, Medium, Dark) are examples of factors. Below, in `f1`, create a factor with six groups, labeled 1 through 6 using the `factor()` and `rep()` functions; note that R treats these values as factors; so, for example, you can't multiply `f1` by a number. Try multiplying `f1` by 3...

[13]:
```
#YOUR CODE HERE

f1 = factor(rep(1:6, each=1, times=1))
f2 = factor(c("a",7,"blue", "blue"))
f2

f1*3
```

1. a 2. 7 3. blue 4. blue

Levels: 1. '7' 2. 'a' 3. 'blue'

```
Warning message in Ops.factor(f1, 3):
"'*' not meaningful for factors"
```

1. <NA> 2. <NA> 3. <NA> 4. <NA> 5. <NA> 6. <NA>

[14]:
```
f1
```

1. 1 2. 2 3. 3 4. 4 5. 5 6. 6

Levels: 1. '1' 2. '2' 3. '3' 4. '4' 5. '5' 6. '6'

[ ]:
```
# Test Cell
```

In the code cell below, I've written several commonly used functions. Explore. Change/add some arguments. See the power of R! Note that the first function creates (pseudo) random numbers from a normal distribution. Don't worry if you don't know what that is yet; we'll learn about it. But R can generate random numbers...that's cool!

[ ]:
```
x <- rnorm(50, mean=5, sd=1) #generates 50 random numbers from a gaussian with mean 5 and sd 1. Don't worry...you'll know what this means soon!
hist(x, density = 20)
length(x) #return the length of x
sum(x) #sum the numbers in x
mean(x) #calculate the mean of the numbers in x
var(x) #calculate the variance of the numbers in x
sd(x) #calculate the standard deviation of x
median(x) #calculate the median of x
range(x) #calculate the range of x
log(x) #calculate the natural log of x
summary(x) #return 5-number summary of x

hist(x, density = 20, freq = FALSE) #histogram of those random numbers; freq = FALSE makes x a 'density' (integrates to 1); density shades the boxes
```

```
curve(dnorm(x,mean = mean(x),sd = sd(x)),from = min(x), to = max(x), add =␣
↪TRUE) #overlay of normal density
```

2 Part II: Vector Indexing

R (like Matlab and Python) has a nice indexing system. Given a matrix A, we can access the $[i, j]$ element of A by writing `A[i,j]`. We can access the i^{th} row by typing `A[i,]`. And, we can access the j^{th} column by typing `A[,j]`. And, we can do even fancier things too…

Modify the third line of code below to access the second, fourth, and sixth elements of the vector `a`. Take note of whether R starts indexing at 0 or 1! Also, note that the last line uses the minus sign to access elements *other than* those specified in the index.

[15]:
```
a = seq(2, 16, by = 2)
a
a[c(2, 4,6)]

###
a[-c(1,5)]
```

1. 2 2. 4 3. 6 4. 8 5. 10 6. 12 7. 14 8. 16

1. 4 2. 8 3. 12

1. 4 2. 6 3. 8 4. 12 5. 14 6. 16

Write code that prints the third through sixth elements of `a` and store it in a variable `b` (there's a short and long way).

[16]:
```
#YOUR CODE HERE
a[3:6]
```

1. 6 2. 8 3. 10 4. 12

[17]:
```
# Test Cell
```

Logical indexing is another powerful tool for working with data in R.

Let's start with typing things like `a > 10`, `a <= 4`, `a == 10`, and `a != 10`. What do these lines do?

[ ]:
```
#YOUR CODE HERE
fail() # No Answer - remove if you provide an answer
```

Now let's look at more complicated logical statements. Note that & represents "and", and | represents "or" in R.

Write code that prints TRUE for values of a greater than 6 and less than or equal to 10, and FALSE otherwise. Then, write code that prints TRUE for values of a less than or equal to 4 or greater or equal to 12, and FALSE otherwise.

[19]:
```
#YOUR CODE HERE
a > 6 & a <= 10
a <= 4 | a >= 12
```

1. FALSE 2. FALSE 3. FALSE 4. TRUE 5. TRUE 6. FALSE 7. FALSE 8. FALSE

1. TRUE 2. TRUE 3. FALSE 4. FALSE 5. FALSE 6. TRUE 7. TRUE 8. TRUE

The real power of logical indexing comes when we take these vectors of logicals and put them in the index of a vector. The first line below tells R to print all elements of `a` such that `a < 6`.

Write code to print elements of a equal to 10. Then, print elements of a less than six or equal to 10.

[22]:
```
#YOUR CODE HERE
a[a == 10]
a[a < 6 | a == 10]
```

10

1. 2 2. 4 3. 10

2.1 Part III: Functions, Loops and Beyond!

As we've seen, R has many built in functions; but, you can write your own, too! Below is the syntax.

Write a function that concatenates two vectors.

[ ]:
```
y = function(input){
    #stuff
    return(output)
}

#YOUR CODE HERE
fail() # No Answer - remove if you provide an answer
```

The function f1 below produces results like 'T' 'H' 'T' 'T'. We can think of `f1` a function that flips a coin four times.

Write a function `f2` that can be interpreted as rolling a fair die eight times. How could you modify the code to roll a biased die?

Notice that every time you run these function, the results are different! Uncomment the line `set.seed(99)` and notice what happens.

[ ]:
```r
#set.seed(99)
f1 = function(){
    s = sample(c("H", "T"), size = 4, replace = TRUE)
    return(s)
}

f1()

f2 = function(){
    s = sample(c(1,2,3,4,5,6), size = 8, replace = TRUE)
    return(s)
}

f2()
```

In R, just like in C++, Matlab, Python, etc., you can write loops to repeat a sequence of instructions until a certain condition is met.

Below is the syntax for a loop. Inside the loop, write a line of code that populates the matrix `M` with the numbers 1 through 20.

[26]:
```r
n = 20;
M = matrix(NA, ncol = 2, nrow = 10); #check the help file so that you know what␣
↪this does!

for (i in 1:n){
    #YOUR CODE HERE
    fail() # No Answer - remove if you provide an answer
}
M
```

```
    Error in fail(): could not find function "fail"
Traceback:
```

Now, write a loop to calculate the mean of each column of `M`.

[25]:
```r
m = matrix(NA, nrow = 2, ncol = 1)
for (i in 1:2){
    #YOUR CODE HERE
    mean(m[i])
}
m
colMeans(M)
```

A matrix: 2×1 of type lgl

NA
NA

```
Error in is.data.frame(x): object 'M' not found
Traceback:
```

```
1. colMeans(M)

2. is.data.frame(x)
```

2.1.1 Here's an incredibly important lesson about R: many tasks that require loops in other languages (e.g., C++) *do not* require loops in R.

As an example, use the function `colMeans()` to calculate the means of the columns of `M` without a loop. Store these means in a variable `M_means`. Then, look at the help file for `colMeans()` to see related functions.

[ ]:
```
#YOUR CODE HERE
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
# Test Cell
```

Note that the family of functions that includes `colMeans()` does not include a function to calculate the standard deviation or variance of a column (or row). But we can still do this without loops, using the `apply()` function.

Below, the `apply()` function is set up to calculate the variance of each row of M. Modify the code to calculate the standard deviation of each column of M. Store your answer in `col_sd`.

[ ]:
```
apply(M, 1, var)
#YOUR CODE HERE
fail() # No Answer - remove if you provide an answer
```

[ ]:
```
# Test Cell
```

Recall above that we wrote a function `f1` that flips a coin four times. As odd as it sounds, it will become desirable for us to repeat many times (say, m times) the process of flipping a coin four times. The result might be a matrix with m rows and four columns.

Write a loop to repeat the process of "flipping a coin four times" 10,000 times. The result should be a 10,000 by 4 matrix.

[ ]:
```
#YOUR CODE HERE
fail() # No Answer - remove if you provide an answer
```

But, as master R programmers, we avoid loops whenever possible. One reason why we do this is because loops are slower than the fancy R alternatives. As an alternative to the loop above, let's use the `replicate()` function to repeat the coin flipping process.

Using the help function as your guide, use `replicate()` to repeat the process of "flipping a coin four times" 10,000 times.

[ ]:
```
#YOUR CODE HERE
fail() # No Answer - remove if you provide an answer
```

2.2 Part IV: Working with Data Frames

Datasets in R are stored as *data frames*. Data frames are just tables of data (vectors of data of equal length). It will be important for future assignments (and in life!) for you to be able to work with data frames.

Take the vectors below and store them in a data frame, called `mydataframe`, using the function `data.frame()`.

[ ]:
```
d = c(1, 2, 3, 4)
e = c("red", "white", "blue", NA)
f = c(TRUE, TRUE, TRUE, FALSE)

#YOUR CODE HERE
fail() # No Answer - remove if you provide an answer
```

[ ]:
```

```

Note that the names of the columns aren't informative (they are just the names of the vectors used to create that dataframe). We can change that...

[ ]:
```
#rename columns of data frame
names(mydataframe) = c("ID", "Color", "Passed")
mydataframe

#name columns while creating data frame
dataframe2 = data.frame(ID=d, Color=e, Passed=f)
dataframe2
```

There are a few ways that you can access elements of a dataframe.

1. You can access columns of a dataframe by typing `mydataframe$NameOfColumn` or by typing `mydataframe[ ,i]` (the latter will print the i^{th} column).
2. To access the i^{th} row, type `mydataframe[i, ]`.
3. To access the $[i, j]$ element, type `mydataframe[i,j]`. Note that data frames are indexed just like matrices: [row, column].

Print the `Color` column of the dataframe from above in two different ways. Print the second row. Print the element containing "white".

```
[ ]: #YOUR CODE HERE
     fail() # No Answer - remove if you provide an answer
```

You can save a dataframe using the function `write.table()`.

Below, I give code to save `mydataframe` in a location on my computer. Modify the code to save it somewhere on your computer. I would take this time to think about being organized with your files for this course! Notice the `sep` argument is set to `"\t"`. This saves the file as a tab separated file. Look at the help file to see other options here.

```
[ ]: ###Writing data to file
     #write.table(mydataframe, paste("~/Google Drive/University of Colorado/example.
      ↪txt", sep = ""), sep = "\t")
```

Now, let's read the file you saved back into R (this is going to be important for homework assignments!).

Modify the code below to read your file back into R. Investigate what the `header` argument is doing!

Note that the `head()` function prints the first few rows of your data frame. This can be helpful when you want to see how your data are organized, but don't want to print the whole thing.

```
[ ]: #data = read.table(paste("~/Google Drive/University of Colorado/Boulder",
         #                      "example.txt", sep = ""), header = TRUE, sep = "\t")
     #head(data)
```

3 Part V: Real-World Data Example

3.1 Topic #4: t-tests in R

Let's explore a dataset about book prices from Amazon. The data consists of data on $n = 325$ books and includes measurements of:

- `aprice`: The price listed on Amazon (dollars)
- `lprice`: The book's list price (dollars)
- `weight`: The book's weight (ounces)
- `pages`: The number of pages in the book
- `height`: The book's height (inches)
- `width`: The book's width (inches)
- `thick`: The thickness of the book (inches)

- `cover`: Whether the book is a hard cover of paperback.
- And other variables...

First, we'll read this data in from Github...

```
[ ]: library(RCurl) #a package that includes the function getURL(), which allows for␣
     ↪reading data from github.
     library(ggplot2) #a package for nice plots!

     #getURL is a nice way of reading in data from the web
     url = getURL(paste0("https://raw.githubusercontent.com/bzaharatos/",
                         "-Statistical-Modeling-for-Data-Science-Applications/",
                         "master/Modern%20Regression%20Analysis%20/Datasets/amazon.
     ↪txt"))
     #stores the data in the dataframe amazon
     amazon = read.csv(text = url, sep = "\t")

     #prints the names in the dataframe
     names(amazon)
```

Next, let's create a new data frame, called `df`, and store a subset of the variables. In addition, we'll change the names of the variables in the dataframe to something cleaner and easier to work with. Take note of how to do this :)

```
[ ]: df = data.frame(aprice = amazon$Amazon.Price, lprice = as.numeric(amazon$List.
     ↪Price),
                     pages = amazon$NumPages, width = amazon$Width, weight =␣
     ↪amazon$Weight..oz,
                     height = amazon$Height, thick = amazon$Thick, cover =␣
     ↪amazon$Hard..Paper)

     summary(df)
```

From the summary, we can see that there are missing values in the dataset, coded as `NA`. There are many ways to deal with missing data. Suppose that sample unit i has a missing measurement for variable z_j. We could:

1. Delete sample unit i from the dataset, i.e., delete the entire row. That might be reasonable if there are very view missing values and if we think the values are missing at random.
2. Delete the variable z_j from the dataset, i.e., delete the entire column. This might be reasonable if there are many many other missing values for z_j and if we think z_j might not be neccesary for our overall prediction/explanation goals.
3. Impute missing values by substituting each missing value with an estimate.

For more information on missing values, see this resource.

Since most of our columns/variables are not missing values, and since these variables will be useful to us in our analysis, option 2 seems unreasonable. Let's first try option 3: impute the missing values of `lprice`, `pages`, `width`, `weight`, `height`, and `thick` with the mean of each. The `which()` and `is.na()` functions might help:

```
[ ]: which(is.na(df$lprice))
```

```
[ ]: #YOUR CODE HERE
     fail() # No Answer - remove if you provide an answer
     summary(df)
```

```
[ ]:
```

Use the `summary()` function to print numerical summaries of this dataset.

```
[ ]: #YOUR CODE HERE
     fail() # No Answer - remove if you provide an answer
```

Use the `sort()` function to order the`lprice` variable from lowest to highest. Remember to use the `df` data frame!

```
[ ]: #YOUR CODE HERE
     fail() # No Answer - remove if you provide an answer
```

Code the `cover` varible as a factor.

```
[ ]: #YOUR CODE HERE
     fail() # No Answer - remove if you provide an answer
```

```
[ ]:
```

Note that you could provide more descriptive labels for the levels of this factor (note that `H` = "Hardcover" and `P` = "Paperback"). The easiest way do do this is with the `levels()` function: `levels(x) = value`.

```
[ ]: levels(df$cover) = c("Hardcover","Paperback")
     summary(df)
```

Print a histogram of the `pages` variable. Comment on it's shape.

```
[ ]: #YOUR CODE HERE
     fail() # No Answer - remove if you provide an answer
```

YOUR ANSWER HERE

Use the `plot()` function to produce a scatterplot of `aprice` (y) against `lprice` (x). What do you notice about this plot?

[ ]:
```
#YOUR CODE HERE
fail() # No Answer - remove if you provide an answer
```

YOUR ANSWER HERE

Use the `boxplot()` function to produce a boxplot of `pages` conditioned on `cover`. Interpret this plot.

[ ]:
```
#YOUR CODE HERE
fail() # No Answer - remove if you provide an answer
```

YOUR ANSWER HERE

Note another way to read data from the web…

[ ]:
```
### Reading data from the web...
read.table("http://www.stats.ox.ac.uk/pub/datasets/csb/ch11b.dat")
```