R

Introduction, Variables, Data Types

Brief History

- R was developed initially as an alternative implementation of a language known as S
  - S first came out in 1975 and was originally developed at Bell Labs
- Work on R began in 1993, the first paper was publish in 1996, and the language reached version 1.0 in 2000
  - Lead by a team at the University of Auckland in New Zealand originally
- Designed originally for statisticians, not for programmers

Running R

- R can be run
  - From the command line, by using the command `R`
  - Using the shebang line `#!/usr/bin/Rscript`
  - In Jupyter using the IR kernel
  - From inside the RStudio IDE

Limitations of R

- Code is generally slower than other languages
  - This was an acceptable trade off given the ease of use
- Uses a lot of memory
  - No easy way to perform calculations in chunks, although some packages are starting to provide support for this
  - Is potentially a poor choice for big data

Assignment

- `R` supports two assignment operators: `<-` and `=`
- Although both are fine, most style guides and books suggest using `<-` is preferred
  - There are many people that argue the exact oppostite however
- `<-` Can be reversed to be written as `->` but this is not normally done

In [ ]:

```
a <- 1
b = 1
1 -> c
```

In [ ]:

```
a == b
```

In [ ]:

```
b == c
```

Variable Names

- Variables can contain letters, numbers, underscores, and the dot symbol
  - Because of some historical weirdness, dots in `R` are often found instead of underscores

    ```
    a.long.name <- "String"
    ```

- The following names should not be used

  ```
  c, q, s, t, C, D, F, I, T
  ```

In [ ]:

```
aLongName <- 0
a_long_name <- 0
a.long.name <- 0
```

In [ ]:

```
print(aLongName)
```

In [ ]:

```
print(a_long_name)
```

In [ ]:

```
print(a.long.name)
```

Data Types and Data Structures

- `R` has data types, and they are important, but they take a back seat to the data structures
  - A variable cannot be scalar in `R`
- The simplest data structure are `vector`s
  - Every assignment that seems like a single number, string, etc. is actually a single element vector

In [ ]:
```
num <- 1
print(num)
```

In [ ]:
```
string <- "String"
print(string)
```

In [ ]:
```
bool <- TRUE
print(bool)
```

Data Types

- The data types supported by `R` are:
  - integer
  - double
  - complex (Uses "i" rather than "j" as seen in python)
  - character (This can hold strings of any length)
  - logical

In [ ]:

```
#Integers must be denoted by appending "L" to the number
#Otherwise they will be interpreted as a double by default
int <- 1L

#typeof() function returns the type as a string
print(typeof(1L))
print(typeof(1))
```

In [ ]:

```
float.a <- 1
float.b <- 1.01

print(typeof(float.a))
print(typeof(float.b))
```

In [ ]:

```
#Infinity and Not-a-Number are both represnted as doubles
float.c <- NaN
float.d <- Inf
float.e <- -Inf

print(typeof(float.c))
print(typeof(float.d))
print(typeof(float.e))
```

In [ ]:

```
imaginary.a <- 1 + 1i
imaginary.b <- 1 + 0i

print(typeof(imaginary.a))
print(typeof(imaginary.b))
```

In [ ]:

```
string.example.1 <- "String"
string.example.2 <- 'String'

print(typeof(string.example.1))
print(typeof(string.example.2))

string.example.2 <- 1
print(typeof(string.example.2))
```

In [ ]:

```
#Logical values are typed in all uppercase letters
logic.t <- TRUE
logic.f <- FALSE

print(typeof(logic.t))
print(typeof(logic.f))
```

Testing Data Types

- `R` has numerous predicate functions relating to data types
- There is one for each data type
  - `is.DATA_TYPE_NAME(x)`
  - e.g. `is.integer(x)`
- There is also a generic number predicate
  - `is.numeric(x)`

In [ ]:

```
print(int)
print(is.integer(int))
print(is.double(int))
print(is.numeric(int))
print(is.numeric("1"))
```

Type Casting

- While data types will automatically be coerced in some situations, to explicitly cast use variations of the `as` function
  - `as.DATA_TYPE_NAME(x)`
  - **eg** `as.integer(1.003)`
- This pattern is used throughout `R`, not just with primitive data types

In [ ]:

```
print(as.character(1L))
print(as.integer(1.0004))
print(as.integer(Inf))
print(as.double(1L))
print(as.complex(1))
print(as.numeric(TRUE))
```

Data Structures

- Basic Data Structures in `R` can be described by the number of dimensions supported, and the data types allowed
- From "Advanced R" by Hadley Wickham

	Homogeneous	**Heterogeneous**
1-D	Vector	List
2-D	Matrix	DataFrame
N-D	Array	

Vectors

- A vector can be created by using the `c` function

```
a.vector <- c(1,2,3,4)
```

- All elements of a vector must be the same. If multiple types are passed to the `c` function, they will be coerced

In [ ]:
```
a.vector <- c(1,2,3,4)
print(a.vector)
```

In [ ]:
```
a.vector <- c(1.001,2,3,4)
print(a.vector)
```

In [ ]:
```
a.vector <- c(1.01,TRUE,3,4)
print(a.vector)
```

In [ ]:
```
a.vector <- c(TRUE,"a",3,4)
print(a.vector)
```

Factors

- Factors are vectors that are limited to certain values
  - Represent categorical data
  - Helpful in statistical analysis
- A factor can be created using the `factor` function, or converting an existing vector by using `as.factor`

In [ ]:

```
factor.1 <- factor(c("UMBC","UMCP","UMUC","UMB","UB"))
print(factor.1)
cat("\n")
factor.2 <- factor(c("Senior","Junior","Senior",
                     "Junior","Sophmore"))
print(factor.2)
```

In [ ]:

```r
# Can use the levels keyword to specify all possible values
factor.3 <- factor(c("Senior","Junior","Senior",
                     "Junior","Sophmore"),
                    levels=c("Senior","Junior",
                             "Sophmore",'Freshman'))
print(factor.3)
cat("\n")
factor.4 <- as.factor(c("Senior","Junior",
                        "Senior","Junior","Sophmore"))
print(factor.4)
```

Lists

- A list is a one dimensional (technically) data structure
  - It can hold a mixture of any data types
  - It can recursively hold other lists and vectors
- Created using the `list` function

```
a.list <- list("a",2,3.14,FALSE)
```

In [ ]:

```
a.list <- list("a", 2, 3.14, FALSE)

#The str function will show the structure of a variable
#str DOES NOT stand for string, it stands for structure
str(a.list)
print(a.list)
```

In [ ]:
```
recursive.list <- list("a", 2, 3.14, list("re","cursive"))
str(recursive.list)
```

In [ ]:

```
# If you try to use c recursively, there is no error
# Everything is just flattened
a.vector <- c(1,2,3,c(4,5))
str(a.vector)

#Applying c to an arguments including at least one list
#coerces the entire structure to a list
coerced.list <- c(1,2,3,list(4,5),list(6,7))
str(coerced.list)
```

Attributes

- Under the surface, R is a very object-oriented language
  - We will talk more about creating user-defined objects in a later lecture
- All data structures we will discuss today have attributes that can be assigned values
- The general syntax is

```
attr(OBJECT, "ATTRIBUTE_NAME") <- ATTRIBUTE_VALUE
```

In [ ]:

```
obj <- c(3,4,5,6)
print(attr(obj,"time_created"))
attr(obj,"time_created") <- date()
print(attr(obj,"time_created"))
cat("\n")
print(attributes(obj))
```

Special Attributes

- While an attribute name can be anything, a few special attributes exist that modify the behavior of the object
  - Names
  - Dimensions
  - Class
- These attributes are so important that they have dedicated functions to access them, and cannot be access with the `attr` function

Naming Indexes

- An existing list or vector can be given named indices by setting the names attribute
- Just as before, we assign into what looks like function call

```
names(OBJECT) <- c(SERIES OF CHARACTERS)
```

- A list or vector can also be created using named indices

```
VARIABLE <- c(a = 1, b = 2)
```

In [ ]:

```
scores <-  c(80,75,80,100,95,85)
names(scores) <- c("Regex HW","Regex Quiz",
                   "Shell HW","Shell Quiz",
                   "R HW", "R Quiz")
print(scores)
```

Matrices

- A matrix is a 2-d data structure that is homogenous in type
  - Usually numbers, but could be boolean or characters too
- Can by created by
  - Using the `matrix` function
  - Adding dimensions to an already existing vector
  - Using the `cbind` or `rbind` functions

In [ ]:

```
# Using the Matrix Function
m <- matrix( c(1,2,3,4,5,6,7,8,9,10,11,12),
             nrow=3, ncol=4 )
print(m)
cat("\n")
m2 <- matrix(1:12,ncol=4)
print(m2)
```

In [ ]:

```
#Creating a matrix of zeros
zeros <- matrix(0,nrow=3,ncol=4)
print(zeros)
cat("\n")
print(dim(zeros))
```

In [ ]:

```
#Adding Dimensions to an existing Vector
vec <- 1:12
print(vec)
print(dim(vec))
cat("\n")
dim(vec) <- c(3,4)
print(vec)
```

In [ ]:

```
#Using cbind
m3 <- cbind(c(1,2,3),c(4,5,6),c(7,8,9),c(10,11,12))
print(m3)
cat("\n")
m4 <- rbind(c(1,4,7,10),c(2,5,8,11),c(3,6,9,12))
print(m4)
```

Data Frames

- Data Frames are 2-d data structures in which a given column of the data frame must have the same type, but columns may have different types
- Each row is like a record in a simple database
- Is generally the most common data structure encountered in R

Creating a Data Frame

- While Data Frames are often created by reading directly from a file, it is also possible to create them programmatically.
- The general syntax is

```
df <- data.frame(COL1 = c(VALUES FOR COL 1),
                 COL2 = c(VALUES FOR COl2), ...,
                 COL_N = c(VALUES FOR COL_N))
```

In [ ]:

```
df <- data.frame(name=c("UMBC","UMCP","Towson"),
                 zipcode=c(21250,20742,21252),
                 undergrad=c(11142,28472,19596),
                 graduate=c(2498,10611,3109))
print(df)
```

Common Functions on a Data Frame

- The function `nrow` returns the number of rows in the data frame
- The functions `ncol` and `length` both return the number of columns
- The names of the the rows can be accessed and changed using the `row.names` function

In [ ]:

```
print(nrow(df))
print(ncol(df))
row.names(df) <- c('A','B','C')
print(df)
```

Reading Data

- `R` has many built in functions to read data files into data frames
  - `read.table` reads a space separated file by default, and is the base to many other functions
  - `read.csv` reads a comma separated values file, is actually just a call to read.table
- `R` supports many other formats through various libraries
  - One of the most common libraries is `foreign` which reads in data from many similar languages to `R`

In [ ]:

```
usm <- read.table("data/usm.tsv",sep="\t",header=TRUE)
print(usm)
```

In [ ]:

```
usm2 <- read.csv("data/usm.csv",row.names=1)
print(usm2)
```

Writing Data

- `R` similarly supports many different formats in which to write data to a file
  - `write.table`
  - `write.csv`
- By default, column and row names are printed to the file, to remove them set `col.names` or `row.names` to FALSE

In [ ]:
```
write.csv(usm2,'data/usm2.csv')
```

In [ ]:
```
write.csv(usm2,'data/usm2.csv',append=TRUE,col.names=FALSE)
```

In [ ]:
```
write.table(usm2,'data/usm2.csv',sep=","
          ,append=TRUE,col.names=FALSE)
```

Math

- Standard operations of +,-,*,/, and ^
- Modulus operator is %%
- Integer division is %/%
- Square root and absolute value are part of R's base package

In [ ]:

```
#Addition
print(1 + 1)
print(1 + 1.0)
print(1 + 1i + 2)
print(2 + 1 + 3i)
print(2 + 3i + 4 + 5i)
```

In [ ]:

```
#Subtraction
print(3-2)
print(0-3)
```

In [ ]:

```
#Multiplication
print(3 * 4)
print(3 * .12)
```

In [ ]:

```
#Division
print(3/4)
print(0/4)
print(0/0)
print(3/0)
print(-3/0)
```

In [ ]:

```
# Integer Division
print(3 %/% 4)
print(12 %/% 5)
print(3 %/% 0)
print(0 %/% 0)
```

In [ ]:

```
#Modulus

print(3 %% 3)
print(10 %% 3)
print(0 %% 0)
print(3 %% 0)
```

In [ ]:

```
print(3 ^ 3)
print(9 ^ 0.5)
print(10 ^ -2)
```

High-Dimensional Math

- Mathmatical operation on higher dimensional data structures is navtively part of `R`
- For scalar operations, like mutiplying every value by 2, the dimensionality doesn't matter
  - For operations involving two data frames, two matrices, etc. the size should match to prevent unintended outcomes
- In addition, both matrices and data.frames can be transposed using the `t` function

In [ ]:

```
#Vector / Scalar Math
vec <- 1:5
print(vec * 2)
print(vec / 10)
print(vec + 1)
```

In [ ]:

```
#Vector addition
vec2 <- 10:15
print(vec + vec2)
vec2 <- 11:15
print(vec + vec2)
```

In [ ]:

```
#Element-wise multiplication
print(vec * vec2)
cat("\n")
#Dot Product
print(vec %*% vec2)
#print(cvec,vec2))
```

In [ ]:

```
#Matrix / Vector Operations
mat <- matrix(1:20,nrow=5)
print(mat)
print(mat / vec)
```

In [ ]:

```
#Matrix / Vector Operations
mat2 <- matrix(1:20,nrow=4)
print(mat2)
print(mat2 / vec)
```

In [ ]:
```
#DataFrame Operations
print(usm)
cat("\n")
print(usm * 2)
```

In [ ]:

```
#Transposition
print(t(mat))
cat("\n")
```

In [ ]:

```
#What is the datastructure returned by this function?
print(t(usm))
print(as.data.frame(t(usm)))
```

Boolean Comparison

- `R` supports the standard boolean operators of `<, >, <=, >=, == !=`
  - The and an or operators are `&` and `|` respectively
- When used between vectors or matrices, returns a object of the same size filled with boolean values

In [ ]:

```
##Standard Scalar Comparison
print(3 == 4)
print(3 < 4)
print(3 < 4 & 5 < 10)
print(3 == 4 | 4 != 4)
```

In [ ]:

```
## Comparing Data Structures
print(vec)
print(vec2)
cat("\n")
print(vec == vec2)
print(vec < vec2)
```

In [ ]:

```
#Vector and Matrix Comparison
print(vec)
print(mat)
cat("\n")
print(vec == mat)
```

Subsetting Vectors

- **Indexing starts at 1!**
- Subsetting is done using square brackets ([ ])
- Subsetting is most commonly done with a vector of
  - Positive Integers
  - Negative Integers
  - Boolean Values

Positive Integer Subsetting

- Positive integers denote which values to return

In [ ]:

```
print(vec)
print(vec[1])
print(vec[2:3])
print(vec[c(1,5)])
#Can repeat indices
print(vec[c(2,2)])
```

Positive Integer Subsetting

- Negative integers denote which values to *not* return

In [ ]:
```
print(vec)
print(vec[-1])
print(vec[-2:-3])
print(vec[c(-1,-5)])
```

Boolean Value Subsetting

- Values are returned when the subsetting vector contains TRUE
- To prevent unexpected errors, the vector used to subset should be the same length as the vector being indexed into
  - If the index vector is shorter than the vector being indexed, the values will repeat as many times as necessary

In [ ]:

```
# Explicit Boolean Subsetting
print(vec)
print(vec[c(TRUE,FALSE,TRUE,FALSE,TRUE)])
cat("\n")
#Using an expression
print(vec[vec %% 2 == 0])
```

Subsetting Lists

- Subsetting a list with the [] operator will return another list
    - To return a specific value (as a vector) use **[[]]**
- The dollar operator is an alias for **[[]]**, but only **[[]]** can use a variable to do the subsetting

In [ ]:

```
#Returns a list
li <- list(a=1,b=2,c=3,d=4,e=5)
print(li[2])
print(li[[2]])
print(li[['b']])
print(li$b)
idx <- 'b'
cat("\n")
print(li[[idx]])
print(li$idx)
```

Subsetting Matrices

- Matrices can also be subset using the **[]** operator
  - With matrices, two indices can be provided, in the order of row,column
  - If just one is provided, it treats the matrix like a vector

In [ ]:

```
print(mat)
cat("\n")
print(mat[5])
print(mat[5,])
print(mat[,4])
print(mat[5,4])
print(mat[c(5,4),])
```

Subsetting Data Frames

- Subsetting Data Frames is very similar to matrices, but passing one index considered a column
  - The $ operator as used with lists can also be used to refer to a specific column
- Rows (or observations) are selected by adding a comma after the row indices

In [ ]:

```
print(usm[1])
cat("\n")
print(usm['Name'])
#This is a vector rather than a one column DF
print(usm$Name)
```

In [ ]:

```
print(usm[usm['Undergraduate.Enrollment'] > 10000,])
cat("\n")
print(usm[usm['Undergraduate.Enrollment'] > 10000,'name'])
usm['total'] <- usm[3] + usm[4]
print(usm)
```

R’s built-in help system

- `R` has excellent built in help capabilities
  - To access the documentation for a specific function, type `? FUNCTION_NAME`
  - To search all helpfiles for a keyword, use the `??` function
- Typing a function without any arguments or parentheses will at a minimum show you the signature of the function
  - If code is not compiled, the code of the function will be displayed too

In [ ]:

```
?read.table
```

In [ ]:

```
read.table
```