Applied Data Science: Lecture 0

David Shilane

Before the Course

- Applied Data Science has a number of prerequisites, including a preliminary knowledge of statistics, introductory computer programming, and some familiarity with the R language.
- For those with a more limited background with R, this document is intended to be a primer of the simple techniques.
- Understanding these methods will be foundational to the more advanced methods we'll work on in the course.

Why R?

- About as good as it gets for working with data.
- Highly interactive – you can easily explore your data while you are programming.
- The best combination of:
  - *A fully configurable programming language, and*
  - *Easy to use tools that minimize your labor.*

Advantages of R

- Free and Open Source
- Reading Data
- Exploring Data
- Cleaning Data
- Analyses
- Graphics
- Building Models
- Building Tools
- Creating Reports

R is great for every aspect of data science!

Installation

- Download R: https://cran.r-project.org/
- RStudio IDE: https://www.rstudio.com
- Recommended packages:

```
pkgs <- c("data.table", "DT", "rmarkdown", "Hmisc", "knitr",
    "prettydoc", "survival", "scales", "lubridate", "stringi",
    "shiny", "getPass", "ggplot2", "RPostgreSQL", "readxl",
    "flexdashboard")

install.packages(pkgs = pkgs)
```

Along the way, we may make use of other packages, too.

Getting Started With R

Assignments: <- or =

```r
# The hash tag # comments out the remainder of the line
# of code

u <- 2
v = 3
w <- "Hello"
x <- TRUE
# Semi-colons are optional for a single statement on a
# single line.

y <- 31.172
z = -14.13
# Semi-colons are necessary when additional statements
# are included on the same line.
```

Interacting with the Console

You can type out a variable or command in the console to display the result.

```
x <- 3
x
```

```
[1] 3
```

```
y = "Hello"
y
```

```
[1] "Hello"
```

```
47 * (1 + 31/7)
```

```
[1] 255.1429
```

Working with both a source file of code and the console for spot-checking is an excellent way to build and check your code.

Basic Classes of Variables

- **numeric**: numbers such as 43, -1, or 0.23242187987.
- **character**: strings such as “Hello” or “Are you sleeping in class already?”
- **logical**: boolean values TRUE and FALSE.

Others are less used: double, integer, raw, complex, etc.

More specialized forms (e.g. for dates and times) are also available.

Every Basic Variable Is A Vector

- All values in a vector must be of the same basic type.
- You can create longer vectors with various operators.

```
x <- c(1, 2, 3)
x
```

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

```
y <- 1:3
y
```

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

```
seq(from = 1, to = 3, by = 1)
```

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

Vector Arithmetic

You can also use numeric or logical vectors in arithmetic:

```
x <- 1:5
4 * x^2 + 3 * x
```

```
[1]   7  22  45  76 115
```

```
y <- c(TRUE, FALSE)
50 * y + 11
```

```
[1] 61 11
```

Some Variables Are Already Hard Coded For You

```
LETTERS
```

```
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
[18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
pi
```

```
[1] 3.141593
```

```
Inf
```

```
[1] Inf
```

Subsetting Vectors With Indices

```
LETTERS[1:5]
```

```
[1] "A" "B" "C" "D" "E"
```

```
x <- 1:100
x[x <= 8]
```

```
[1] 1 2 3 4 5 6 7 8
```

```
x[x < 5 | x > 95]
```

```
[1]   1   2   3   4  96  97  98  99 100
```

More Complex Data Types

- **matrix**: an (n by m) collection of data. All of the component values must be the same type (numeric, character, or logical).
- **factor**: a classification vector placing all values into one or more categories.
- **list**: a vector that allows for *different data types* in each element. A list also enables objected-oriented programming.

Matrices

```
x <- matrix(data = LETTERS, nrow = 2)
x
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
[1,] "A"  "C"  "E"  "G"  "I"  "K"  "M"  "O"  "Q"  "S"   "U"   "W"   "Y"
[2,] "B"  "D"  "F"  "H"  "J"  "L"  "N"  "P"  "R"  "T"   "V"   "X"   "Z"
```

```
y <- matrix(data = 1:20, nrow = 2, ncol = 10)
y
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    3    5    7    9   11   13   15   17    19
[2,]    2    4    6    8   10   12   14   16   18    20
```

More Matrices

```
z <- matrix(data = c(TRUE, FALSE), nrow = 2)
z
```

```
      [,1]
[1,]  TRUE
[2,] FALSE
```

Vector arithmetic also applies to numeric and logical matrices.

```
4 * z + 1
```

```
     [,1]
[1,]    5
[2,]    1
```

Factors

```
x <- factor(x = c("red", "blue", "green"))
x
```

```
[1] red   blue  green
Levels: blue green red
```

```
y <- factor(x = c(5, 3, 1))
y
```

```
[1] 5 3 1
Levels: 1 3 5
```

```
z <- factor(x = c(TRUE, FALSE))
z
```

```
[1] TRUE  FALSE
Levels: FALSE TRUE
```

Factor Redactor

Factors are great for categorical data analysis. However, it is very easy to confuse them with characters or numeric vectors. Their behavior can be surprising at times:

```
y
```

```
[1] 5 3 1
Levels: 1 3 5
```

```
y + 1
```

```
[1] NA NA NA
```

```
z <- as.numeric(y)
z
```

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

Factored Out

For this reason, factors are tricky. I personally try to avoid factors whenever possible. However, many pieces of legacy code still make use of them. One recommendation is to convert all factor variables into another type. For numeric conversions, it helps to go through an intermediate character conversion first.

```
x <- factor(x = c("5", "3", "1"))
y <- as.numeric(as.character(x))
y
```

```
[1] 5 3 1
```

This ensures the proper conversion to the intended numeric values rather than the numeric *level* of the variable.

Lists

Lists are the first step toward aggregating different *types* and *amounts* of data into a single overarching form.

```
x <- list(14, "red", c(TRUE, FALSE))
x
```

```
[[1]]
[1] 14

[[2]]
[1] "red"

[[3]]
[1]  TRUE FALSE
```

Lists with Objects

The dollar sign $ is the object operator in R.

```
y <- list(age = c(14, 16), favorite_color = c("red", "yellow"),
    studies_algebra = c(TRUE, FALSE))
y
```

```
$age
[1] 14 16

$favorite_color
[1] "red"    "yellow"

$studies_algebra
[1]  TRUE FALSE
```

Calling Elements from Lists

```
y <- list(age = c(14, 16), favorite_color = c("red", "yellow",
    "green"), studies_algebra = c(TRUE, FALSE))

y[[1]][2]
```

```
[1] 16
```

```
y$age[2]
```

```
[1] 16
```

From Lists To Data Frames

```
  age favorite_color studies_algebra
1  14          white             TRUE
2  18         yellow            FALSE
3  15           blue            FALSE
4  16           pink             TRUE
5  18           blue             TRUE
```

Most real data sets out there have:

```
* Many different observations;
* Many different attributes per observation;
* A tabular form;
* Column names -- and possibly row names, too.
```

In R, a **data frame** is a specialized form of a list. Data frames serve as the primary way to store and utilize data.

Generating A Data Frame

```
students <- data.frame(age = c(14, 18, 15), favorite_color = c("purple",
    "green", "black"), studies_algebra = c(TRUE, FALSE,
    FALSE))
students
```

```
  age favorite_color studies_algebra
1  14         purple            TRUE
2  18          green           FALSE
3  15          black           FALSE
```

More typically, you'll encounter data frames by reading in data from a file.

Data Frame Properties

- All of the nice parts of matrices and lists.
- Can provide a comprehensive view of your data.
- Easily integrated with every kind of analysis.
- You can refer to columns by name or index:

```
students$favorite_color
```

```
[1] purple green  black
Levels: black green purple
```

```
students[, 2]
```

```
[1] purple green  black
Levels: black green purple
```

Data Frame Limitations

- Not optimized for speed or efficiency.
- Character vectors are often converted to factors by default.
- Limited options for queries and segmented analyses.

Functions in R

```r
x <- 1:10
print(x = x)
```

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

```r
sum(x = x, na.rm = TRUE)
```

```
[1] 55
```

```r
mean(x = x, na.rm = TRUE)
```

```
[1] 5.5
```

```r
median(x = x, na.rm = TRUE)
```

```
[1] 5.5
```

More Functions

```
summary(object = x)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   1.00    3.25    5.50    5.50    7.75   10.00 
```

```
c(min(x, na.rm = TRUE), max(x, na.rm = TRUE))
```

```
[1]  1 10
```

```
quantile(x = x, probs = c(0.25, 0.5, 0.75))
```

```
 25%  50%  75% 
3.25 5.50 7.75 
```

Random Number Generators

```
n <- 4
rnorm(n = n, mean = 3, sd = 2)
```

```
[1] 3.560581 2.940210 2.533705 4.440394
```

```
runif(n = n, min = 0, max = 2)
```

```
[1] 1.2649254 1.0301659 0.4702516 0.9073165
```

```
sample(x = 1:n, size = n, replace = TRUE, prob = c(0.1 *
    (1:n)))
```

```
[1] 4 4 4 3
```

Character Functions

```
the.colors <- c("red", "blue", "green")
the.colors.upper <- toupper(x = the.colors)
print(x = the.colors.upper)
```

```
[1] "RED"   "BLUE"  "GREEN"
```

```
tolower(x = the.colors.upper)
```

```
[1] "red"   "blue"  "green"
```

```
library(package = Hmisc)
capitalize(string = the.colors)
```

```
[1] "Red"   "Blue"  "Green"
```

Libraries

The function **capitalize** is contained in the **Hmisc** package. Many packages are not automatically loaded in R. These packages need to be installed (once) and then loaded (once per session).

```
install.packages(pkgs = c("Hmisc", "data.table"))
library(package = "Hmisc")
```

Every library in R was written by an open-source contributor. Maybe someday you will publish your own packages.

More Character Functions

```
the.colors <- c("red", "blue", "green")

paste(the.colors, collapse = " and ")
```

```
[1] "red and blue and green"
```

```
sprintf("%s things", string = the.colors)
```

```
[1] "red things"   "blue things"  "green things"
```

Natural Language Processing Already? grep

```
titles <- c("Dr. Strangelove", "Young Frankenstein",

    "Robin Hood: Men In Tights", "The Princess Bride")
in.indices <- grep(pattern = "in", x = titles)
titles[in.indices]
```

```
[1] "Young Frankenstein"         "Robin Hood: Men In Tights"
[3] "The Princess Bride"
```

```
in.indices.as.word <- grep(pattern = "In ", x = titles, fixed = TRUE)
titles[in.indices.as.word]
```

```
[1] "Robin Hood: Men In Tights"
```

NLP: gsub

```
gsub(pattern = "Dr.", replacement = "Doctor", x = titles)
```

```
[1] "Doctor Strangelove"         "Young Frankenstein"
[3] "Robin Hood: Men In Tights" "The Princess Bride"
```

```
gsub(pattern = "Men In Tights", replacement = "Prince of Thieves",
    x = titles)
```

```
[1] "Dr. Strangelove"                "Young Frankenstein"
[3] "Robin Hood: Prince of Thieves" "The Princess Bride"
```

Developing A Good Style

- Make your code easy to read.
- Use comments to explain your work.
- Keep it organized.
- Make everything you write as *reusable* as possible.
- Be consistent!

Producing elegant and readable code is important. You want others to be able to easily understand your work. But guess who will read your code the most? **You will**, probably by a factor of 100. So the most important thing is creating a style that supports your own development with good habits.

Naming Your Parameters

```r
x <- (1:10)/2

round(x, 2)
```

```
 [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```r
round(x = x, digits = 2)
```

```
 [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

These two calls produce the same results. However, the second call specifically names the function's parameters. This is a good practice. Many functions have a large number of parameters. If you name them, they can be called in any order. Otherwise, the order matters, and the potential for mistakes increases. Consider yourself forewarned!

When It's OK to Slack Off

It's more acceptable not to name the parameter when it's a single one:

```
print(x)
```

```
 [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
mean(x)
```

```
[1] 2.75
```

```
summary(x)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  0.500   1.625   2.750   2.750   3.875   5.000 
```

Reusability With Constants

```
largest.number <- 10
x <- 1:largest.number
y <- LETTERS[x]
z <- tolower(x = y)
print(x = data.frame(Upper_Case = y, Lower_Case = z))
```

```
   Upper_Case Lower_Case
1           A          a
2           B          b
3           C          c
4           D          d
5           E          e
6           F          f
7           G          g
8           H          h
9           I          i
10          J          j
```

Writing Functions

```
# Compute the mean of the values of x that are greater
# than the threshold.
filtered.mean <- function(x, lower.threshold) {
    y <- x[x > lower.threshold]
    the.filtered.mean <- mean(x = y, na.rm = TRUE)
    return(value = the.filtered.mean)
}
filtered.mean(x = c(1:10, 2:14, 82), lower.threshold = 5)
```

```
[1] 14.13333
```

```
filtered.mean(x = c(1:10, 2:14, 82), lower.threshold = 82)
```

```
[1] NaN
```

Returning Objects From Functions

```
filtered.mean.2 <- function(x, threshold, return.threshold = TRUE) {
    y <- x[x > threshold]
    the.filtered.mean <- mean(x = y, na.rm = TRUE)
    if (return.threshold == TRUE) {
        return(list(filtered.mean = the.filtered.mean, threshold = threshold))
    } else {
        return(value = list(filtered.mean = the.filtered.mean))
    }
}
x <- filtered.mean.2(x = c(1:10, 2:14, 82), threshold = 5)
print(x)
```

```
$filtered.mean
[1] 14.13333

$threshold
[1] 5
```

Adding Fields To A Data Frame in Functions

```
offer.mortgage <- function(dat, cash.threshold, income.threshold) {
    dat$offer_mortgage <- (dat$cash_on_hand >= cash.threshold &
        dat$income >= income.threshold)

    return(value = dat)
}
cash_on_hand = c(14000, 50000)
income = c(40000, 62000)
offer.mortgage(dat = data.frame(cash_on_hand = cash_on_hand,
    income = income), cash.threshold = 40000, income.threshold = 50000)
```

```
  cash_on_hand income offer_mortgage
1        14000  40000          FALSE
2        50000  62000           TRUE
```

Deep Dive #1: Creating a Vector Efficiently

Coding Challenge: Create a vector x of 100,000 elements. For each index i, the **i**th value of the vector should be equal to:

$$x[i] = 1 + i^2.$$

Let's consider several different approaches to constructing this vector with speed in mind.

Approach #1: Looping with Concatenation

```
n <- 10^5

toc1 <- Sys.time()
a1 <- c()

for (i in 1:n) {
    a1 <- c(a1, 1 + i^2)
}
tic1 <- Sys.time()
time1 <- tic1 - toc1
print(time1)
```

```
Time difference of 27.65309 secs
```

Approach #2: Looping with Defined Memory

```
toc2 <- Sys.time()
a2 <- numeric(length = n)

for (i in 1:n) {
    a2[i] <- 1 + i^2
}
tic2 <- Sys.time()

time2 <- tic2 - toc2
print(time2)
```

```
Time difference of 0.01123214 secs
```

This is a big improvement already. Can we do better?

Approach #3: Vector Operations

```
toc3 <- Sys.time()

a3 <- 1 + (1:n)^2

tic3 <- Sys.time()

time3 <- tic3 - toc3
print(time3)
```

```
Time difference of 0.002598047 secs
```

Speed Comparison

Relative to the fastest method, here are the multipliers for each approach:

Show 10 entries Search:

Type	Looping with Concatenation	Looping with Defined Memory	Vector Operations
Overall Seconds	27.6531	0.0112	0.0026
Multiplier	10643.7995	4.3233	1

Showing 1 to 2 of 2 entries Previous 1 Next

R Can Be Many Things

The immediate conclusions are obvious to those who've used R – vectorized operations are better than for loops (where possible). However, the broader conclusions are worthy of more consideration.

- R can be very fast and simple when the best practices are used.
- R can be maddeningly slow in the worst case scenario.
- Sometimes your first instinct is good enough, but optimizing for speed can pay off in other circumstances.

I have made my share of mistakes that have led to worst-case scenarios. These situations have led me to investigate and refine my approach. Ultimately I have resolved to write code that gets to the best result **as often as possible** while recognizing that my process is imperfect.

Deep Dive 2: Writing Good Functions

In a prior class, I gave out the following homework problem:

Question: Write a function called even.or.odd. Given a vector of numeric inputs that are whole numbers, the function should return a character vector specifying whether each input is **even** (divisible by 2) or **odd** (not divisible by 2). To demonstrate your work, show the results of this function on the vector 1:5.

One Student's Solution

I received the following (paraphrased) answer, which was quite interesting:

```
the.values <- 1:5
the.results <- c()

even.or.odd.1 <- function(x) {

    for (x in the.values) {
        if (x%%2 == 1) {
            the.results <- c(the.results, "odd")
        }
        if (x%%2 == 0) {
            the.results <- c(the.results, "even")
        }
    }
    return(the.results)
}

even.or.odd.1(x = 1:5)
```

```
[1] "odd"  "even" "odd"  "even" "odd"
```

Right for the Wrong Reasons

- For the vector 1:5, the function provided the proper classification for each of the values.
- However, unbeknownst to the student, the TA had a few other test cases to check:

```
even.or.odd.1(x = 2)
```

```
[1] "odd"  "even" "odd"  "even" "odd"
```

```
even.or.odd.1(x = 0 * (1:100))
```

```
[1] "odd"  "even" "odd"  "even" "odd"
```

No matter the input, the function always returned the same sequence!

Let's examine all of the issues with the code:

Problem 1: Unused Inputs

- The input **x** is not used in the function!
- Every function should have results that depend upon the inputs.
- If the result does not depend on an input, then that input is most likely unnecessary.

Problem 2: Overwriting

- The **x** in the for loop overwrites the **x** from the input!

The moment you create a loop like **for(x in the.values)**, R creates a variable called x. It overwrites whatever else was available for x at that time. (This is true within the *environment* that you're working in. A function has its own environment. If there is an x like x = 3 outside of the function, then it remains.) So if you feed your even.or.odd function a variable x like x = 1:5, then this for loop would overwrite that value with a new value – in this case, each successive value of **the.values**. That is how the input of the function was lost.

Problem 3: for loops are slow in R.

There are times when for loops are necessary. Unfortunately, R wasn't built to optimize their speed.

To understand why, it helps to remember that R is an **interpreted language**. Its **interpreter** program takes the R code that you have written and translates it into C. In many ways, this is a good thing. R's translations take care of all of the more granular aspects of classic programming like working with memory.

However, for loops in R are not translated into for loops in C (which are quite fast). They are translated into a large number of coding statements (which are slow to translate).

Problem 4: Global Variables

- **the.results** starts as a global variable rather than a locally defined variable.

- In reality, the input **x** should have been the value used in place of **the.results**

It's better to define local variables inside of a function. Using global variables can be OK, but you have to make sure that you know exactly what their values are. The best practice is to only use global variables in a function if their values are considered universal for the program (things that won't change in the middle of the code).

Problem 5: Looping with Concatenation

This is the same issue we discussed in Deep Dive 1. This method of building a numeric vector can be painfully slow.

Problem 6: For Loops Instead of Vectorized Operations

This could all be accomplished far more simply:

```
even.or.odd <- function(x) {
    the.outcomes <- c("even", "odd")
    the.true.false <- x%%2
    the.results <- the.outcomes[1 + (the.true.false)]
    return(the.results)
}
even.or.odd(x = 1:5)
```

```
[1] "odd"  "even" "odd"  "even" "odd"
```

```
even.or.odd(x = 2)
```

```
[1] "even"
```

```
even.or.odd(x = 2:4)
```

```
[1] "even" "odd"  "even"
```