Week 2 R Style and Programming

January 18, 2018

R Programming Style

Google's R Style Guide

While there is not universal agreement on programming style, we will adhere to the concepts in the Google R Style Guide: `https://google.github.io/styleguide/Rguide.xml`

Notation and Naming

- **File Names:** File names should end in .R and be meaningful.
  - Good: predict_ad_revenue.R
  - Bad: foo.R
- **Identifiers:** Don't use underscores or hypens in identifiers.
  - The preferred form for variable names is all lower case letters with words separated with dots (`variable.name`), but `VariableName` is also accepted.
  - Function names begin with capital letters and include no dots (`FunctionName`)

Syntax

- **Spacing:**
  - Place spaces around all operators (`==`, `+`, `...`)
  - Do not place a space before a comma, but always place one after a comma.
  - Place a space before left parenthesis, except in a function call.
- **Assignment:**
  - Use <- not = for assignment.

Operators in R

- Most mathematical operators are self explanatory, but here are a few more important operators.
  - == will test for equality. For example to determine if pi equals three, this can be evaluated with `pi == 3` in R and will return FALSE. Note this operator returns a logical value.
  - & is the AND operator, so `TRUE & FALSE` will return FALSE.
  - | is the OR operator, so `TRUE | FALSE` will return TRUE.
  - ! is the NOT operator, so `! TRUE` will return FALSE.
  - ^ permits power terms, so `4 ^ 2` returns 16 and `4 ^ .5` returns 2.

Exercise: Order of operations

Note that order of operations is important in writing R code.

```
4 - 2 ^ 2
(4 - 2) ^ 2
5 * 2 - 3 ^ 2
! TRUE & pi == 3
! (TRUE | FALSE)
```

Evaluate all four expressions. Note `!` is R's not operator.

Solution: Order of operations

The results of the R code are:

```
4 - 2 ^ 2
```

```
## [1] 0
```

```
(4 - 2) ^ 2
```

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

```
5 * 2 - 3 ^ 2
```

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

```
! TRUE & FALSE
```

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

```
! (TRUE | FALSE)
```

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

Organization: Layout

- **General Layout:** The general layout of an R script should follow as:

1. Author Comment
2. File description comment, including purpose of program, inputs, and outputs
3. `source()` and `library()` statements
4. Function definitions
5. Executed statements

Organization: Commenting

- Comment your code. Entire commented lines should begin with # and then one space.
- Short comments can be placed after code preceded by two spaces, # and then one space.

```
# create plot of housing price by zipcode
plot(Seattle$Price ~ Seattle$Zip,
  rgb(.5,0,0,.7),  # set transparency for points
  xlab='zipode')
```

Organization: Functions

Functions should contain a comments section immediately below the function definition line. These comments should consist of

1. a one-sentence description;
2. a list of the functions arguments, denoted by `Args:`, with a description of each and
3. a description of the return value, denoted by `Returns:`.

The comments should be descriptive enough that the function can be used without reading the function code.

Overview Functions in R

Functions are a way to save elements of code to be used repeatedly.

```
RollDice <- function(num.rolls){
  #
  # ARGS:
  # RETURNS:
  return(sample(6, num.rolls, replace = T))
}
RollDice(2)
```

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

Exercise: Function Descriptions

Document this function with

1. a description,
2. summary of input(s)
3. summary of outputs

```
RollDice <- function(num.rolls){
  #
  # ARGS:
  # RETURNS:
  return(sample(6, num.rolls, replace = T))
}
```

Note for help with functions in R, type `?sample`.

Solution: Function Descriptions


```
RollDice <- function(num.rolls){
  # function that returns rolls of dice
  # ARGS: num.rolls - number of rolls
  # RETURNS: vector of num.rolls of a die
  return(sample(6, num.rolls, replace = T))
}
RollDice(2)
```

```
## [1] 6 3
```

Parting Words

- Use common sense and *be consistent*.
- If you are editing code, take a few minutes to look at the code around you and mimic the style.
- Enough about writing code; the code itself is much more interesting. Have fun!

Built in R Functions

R Functions

- We have seen several standard functions in R. To get more details in R, type `?FunctionName`. This will open up a help window that displays essential characteristics of the function including arguments and values returned. For example, with the `mean` function the following information is shown:

Description: function for the (trimmed) arithmetic mean.

Usage: mean(x, trim = 0, na.rm = FALSE, . . .)

x: An R object.

trim: the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed.

na.rm: a logical value indicating whether NA values should be stripped before the computation proceeds.

Downloading R Packages

- R has a set of built in functions, which we have used thus far.
- R also has a vast repository of "packages" that contain additional, specialized functions. One example is a graphics packaged called `ggplot2` which we will see later in this class.
- Using these external packages requires two steps:
  1. Download the package `install.packages('ggplot2')`. This only needs to be done once.
  2. Load the package `library(ggplot2)`. This needs to be done when opening R.

Writing R Functions

Exercise: Writing and Documenting a Function

Use the defined style guidelines to create an R script that:

1. Takes a state abbreviations as an input
2. Imports a file available at:
   `http://math.montana.edu/ahoegh/teaching/stat408/datasets/HousingSales.csv`
3. Creates a subset of housing sales from that state.
4. Returns a vector with the mean closing price in that state.

Verify your functions works by running it twice using "MT" and "NE" as inputs.

Solution: Writing and Documenting a Function


```
SummarizeHousingCosts <- function(state){
  # computes average sales price in a state
  # ARGS: state abbr, such as 'MT' or 'CA'
  # RETURNS: vector with average sales price that each state
  housing.data <- read.csv(
    'http://math.montana.edu/ahoegh/teaching/stat408/datasets/HousingSales.csv')
  location <- subset(housing.data, State == state)
  mean.price <- mean(location$Closing_Price)
  return(mean.price)
}
```

```
SummarizeHousingCosts('MT')
```

```
## [1] 164608
```

```
SummarizeHousingCosts('NE')
```

```
## [1] 152050
```

Format of an R function

Here is an example (trivial) R function.

```
SquareRoot <- function(value.in){
  # function takes square root of value.
  # Args: value.in - numeric value
  # Returns: the square root of value.in
  return(value.in ^ .5)
}
```

Square Root Function

Now consider running the function for a few values.

```
SquareRoot(9)
```

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

```
SquareRoot(25)
```

```
## [1] 5
```

Now what happens with `SquareRoot(-1)`?

Square Root Function


```
SquareRoot(-1)
```

```
## [1] NaN
```

What should happen?

Errors in R functions

Here is an example (trivial) R function.

```
SquareRoot <- function(value.in){
  # function takes square root of value.
  # Args: value.in - numeric value
  # Returns: the square root of value.in
  if (value.in < 0) stop('argument less than zero')
  return(value.in ^ .5)
}
```

Square Root Function


```
SquareRoot(-1)
```

This returns:

```
> SquareRoot(-1)
Error in SquareRoot(-1) :
  argument less than zero
```

Exercise: Functions Part 2

Now write a function that;

1. Takes daily snowfall total in inches as input
2. Takes day of week as input
3. Returns whether to ski or stay home.

Also include and the `stop()` function for errors. Test this function with two settings:

- snowfall = 15, day = "Sat"
- snowfall = -1, day = "Mon"

Solution: Functions Part 2


```
ToSki <- function(snowfall, day){
  # determines whether to ski or stay home
  # ARGS: snowfall in inches, day as three letter
  #       abbrwith first letter capitalized
  # RETURNS: string stating whether to ski or not
  if (snowfall < 0) stop('snowfall should be greater
       than or equal to zero inches')
  if (day == 'Sat') {
    return('Go Ski')
  } else if (snowfall > 5) {
    return('Go Ski')
  } else return('Stay Home')
}
```

Solution: Functions Part 2 cont..


```
ToSki(snowfall = 15, day = "Sat")
```

```
## [1] "Go Ski"
```

```
ToSki(-1, 'Mon')
```

```
## Error in ToSki(-1, "Mon"): snowfall should be greater
##          than or equal to zero inches
```

Matrix Style Operations

colMeans, rowSums

R contains a set of built in functions for taking the mean and sum of matrices that have been optimized for speed.

```
mat1 <- matrix(1:4,ncol=2,nrow=2)
rowMeans(mat1)
```

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

```
colSums(mat1)
```

```
## [1] 3 7
```

Apply

For generic functions, the set of apply commands are extremely useful. They provide a mechanism for matrix style operations similar to the built in `rowMeans()` type of functions.

The apply function has three arguments:

- matrix
- margin (rows=1, columns=2)
- function

```
apply(mat1,2,mean)
```

```
## [1] 1.5 3.5
```

Aggregate

Another useful function is aggregate which can be used to compute summary statistics of dataset by a particular group. Aggregate also has three essential elements.

1. An R object
2. list of groups
3. function

```
aggregate(Loblolly$height,by=list(Loblolly$age),mean)
```

```
##   Group.1         x
## 1       3  4.237857
## 2       5 10.205000
## 3      10 27.442143
## 4      15 40.543571
## 5      20 51.468571
## 6      25 60.289286
```

Exercise: Aggregate

Earlier we wrote a function to compute the average housing price for two states, now use aggregate to compute this for all the states in the housing data set.

Solution: Aggregate


```
housing.prices <- read.csv(
  'http://math.montana.edu/ahoegh/teaching/stat408/datasets/HousingSales.csv',
  stringsAsFactors = F)
housing.prices.state <- aggregate(housing.prices$Closing_Price,
                                  by=list(housing.prices$State),mean)
head(housing.prices.state)
```

```
##   Group.1        x
## 1      CA 380006.9
## 2      CO 716647.2
## 3      CT 301400.9
## 4      DC 510675.0
## 5      FL 241977.1
## 6      GA 182596.3
```

Tables for R Markdown

Kable function

Note that output from R can often be hard to read. Luckily there are several options for creating nicely formatted tables. One, which we will use, is the kable function.

Kable function


```
library(knitr)
kable(aggregate(Loblolly$height,
      by=list(Loblolly$age),mean), digits=3,
      caption= 'Average height of loblolly pine by age',
      col.names = c('Tree Age','Height (ft)'))
```

Table 1: Average height of loblolly pine by age

Tree Age	Height (ft)
3	4.238
5	10.205
10	27.442
15	40.544
20	51.469
25	60.289