Lecture 2: Programming in R

Control Flow, Functions, and Vectorization

Yujin Jeong

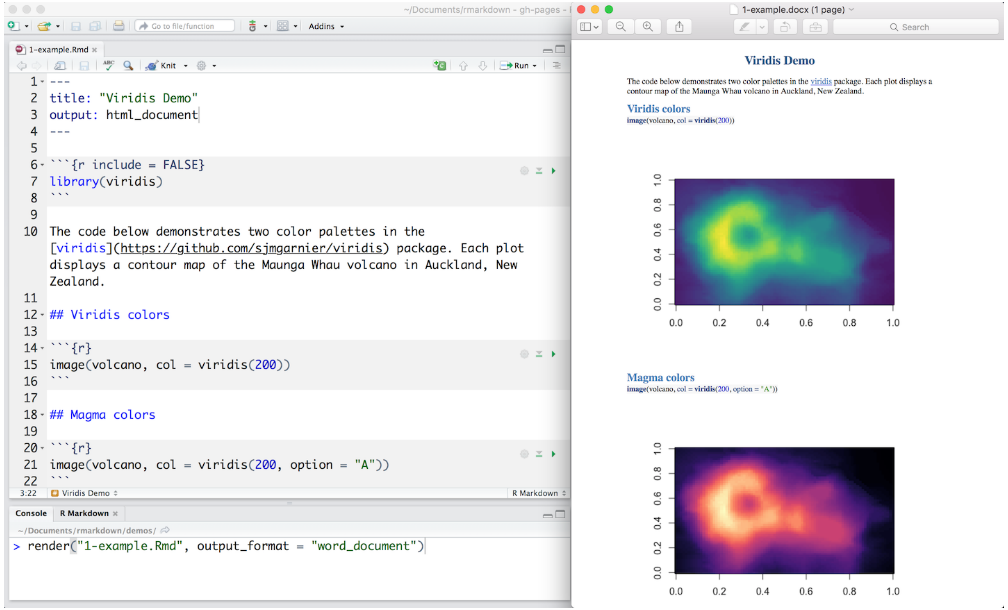
STATS 195

Communicating in R

R Markdown is a document format which allows you to "weave together narrative text and code to produce elegantly formatted output."

R Markdown

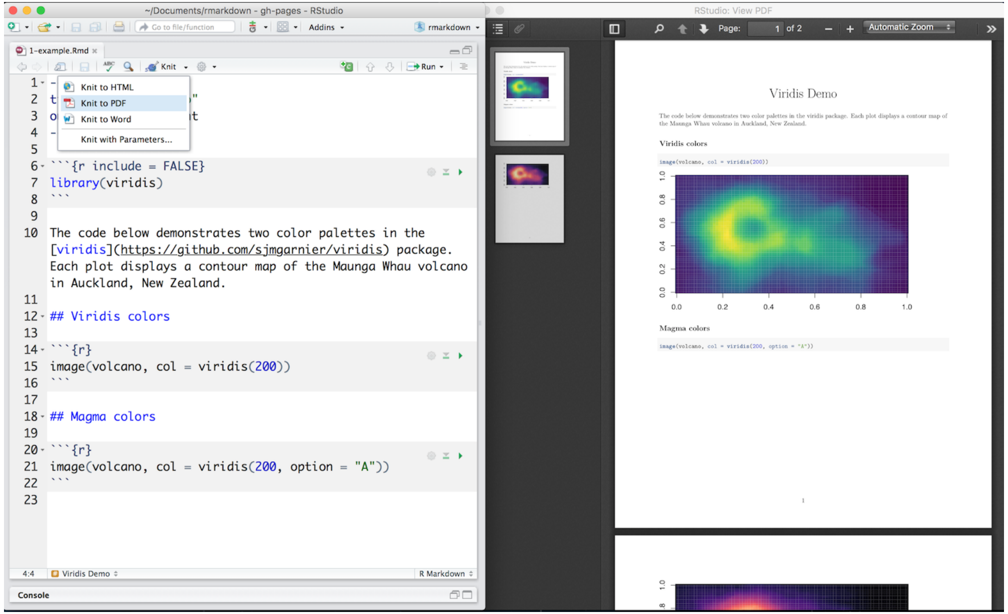
R Markdown files are a plain text files with ".Rmd" extension. The document must contain **YAML header** marked with dashes. You can add both **plain text** and **code chunks.**



Sections and subsections are marked with hashtags. For more details on formatting: R Markdown Cheatsheet

R Markdown

To produce a complete report containing all text, code, and results, **click on "Knit"** in RStudio or **press Cmd/Ctrl + Shift + K**. This will display the report in the viewer pane.



R Markdown

Chunk output can be customized with options supplied to chunk header. Some non-default options are:

- `` `eval = FALSE` ``: prevents code from being evaluated
- `` `include = FALSE` ``: runs the code, but hides the code and its output in the final document
- `` `echo = FALSE` ``: hides the code, but not the results, in the final document
- `` `message = FALSE` ``: hides messages
- `` `warning = FALSE` ``: hides warning
- `` `results = 'hide'` ``: hides printed outputs
- `` `fig.show = 'hide'` ``: hides plots

Programming in R

Random Numbers, Control Flow, Functions, and Vectorization

Random Numbers

R has a bunch of nice functions for generating vectors of random numbers. This can be really useful if you want to run simulations to test your code. Most of the functions follow a simple naming convention starting with the letter r (for random) followed by a four letter abbreviation for the distribution. For example, if we want numbers from the uniform distribution we use runif:

```
runif(10)
```

```
## [1] 0.7754763 0.2439071 0.6769749 0.8173749 0.7311253 0.7924093 0.5890613 0.2449501 0.4585533
## [10] 0.2750597
```

If we want numbers from the standard normal distribution we use rnorm:

```
rnorm(10, mean = 0, sd = 1)
```

```
## [1] -0.5308581  0.1379041 -2.5566166  1.1773119 -0.1015588 -0.2260640 -1.0720503 -0.7131021
## [9] -0.4680549  0.4183264
```

Random Numbers

Generated random numbers will change every time you rerun it. It will often help to be able to track and reuse the same random numbers repeatedly so you can replicate your results. The way to do this is:

```
set.seed(710)
runif(5)
set.seed(710)
runif(5)
set.seed(45)
runif(5)
```

```
## [1] 0.1644271 0.3176075 0.3017475 0.0196399 0.8549568
## [1] 0.1644271 0.3176075 0.3017475 0.0196399 0.8549568
## [1] 0.63337281 0.31753665 0.24092185 0.37841413 0.35214430
```

You can see the results depend on the seed we chose. Which number you feed into set.seed is not important. set.seed(1) is fine. If you are using any random numbers in a homework or a project, you should set your seed at the beginning of the assignment so that your results can be replicated.

DO NOT SEED HACK!

Control Flow

Control flow is the order in which individual statements, instructions or function calls are evaluated.

Booleans

Booleans are logical data types (TRUE / FALSE) associated with conditional statements, which allow different actions and change control flow.

```
# equal: "=="
5 == 5
```

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

```
# not equal: "!="
5 != 5
```

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

```
# greater than: ">"
5 > 4
```

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

Booleans

You can combine multiple boolean expressions.

```r
TRUE & TRUE
```

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

```r
TRUE & FALSE
```

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

```r
TRUE | FALSE
```

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

```r
!(TRUE)
```

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

Booleans

If you combine 2 vectors of booleans by each element, use `` `&` `` or `` `|` ``:

```
c(TRUE, TRUE) & c(FALSE, TRUE)
```

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

```
c(5 < 4, 7 == 0, 1 < 2) | c(5 == 5, 6 > 2, !FALSE)
```

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

If we use double operators `` `&&` `` or `` `||` ``, only the first elements are compared:

```
c(TRUE, TRUE) && c(FALSE, TRUE)
```

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

Booleans

We can also use `` `all()` `` or `` `any()` `` functions to combine booleans:

```
all(c(TRUE, FALSE, TRUE))
```

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

```
any(c(TRUE, FALSE, TRUE))
```

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

If / else statements

If / else statements decide on whether a block of code should be executed based on the associated boolean expression.

```
if (traffic_light == "green") {
  print("Go.")
}
```

```
if (traffic_light == "green") {
  print("Go.")
} else {
  print("Stay.")
}
```

```
if (traffic_light == "green") {
  print("Go.")
} else if (traffic_light == "yellow") {
  print("Get ready.")
} else {
  print("Stay.")
}
```

For Loops

For loops repeat a process a pre-set number of times. As it does each loop, it iterates the input index (i in the below code) over a sequence of values (1 through 10 in the below code).

```
for (i in 1:10) {
  print(i)
}
```

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

For Loops

The index need not interate over just integers, it can be anything:

```
x <- c(7, 35, 29.8)
for (num in x) {
  print(num)
}
```

```
## [1] 7
## [1] 35
## [1] 29.8
```

The `` `seq_along` `` creates a sequence of numbers from 1 to the object's length and can be used as:

```
for (i in seq_along(x)) {
  print(c(i, x[i]))
}
```

```
## [1] 1 7
## [1] 2 35
## [1] 3 29.8
```

While Loops

While loops repeat the loop until a stopping condition is met. If your stopping condition is not guarenteed to be met, the loop will run forever (or until your computer runs out of memory or you halt the process).

```
i = 1
while (i <= 10) {
  print(i)
  i = i + 1
}
```

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

In this case, since we know how many times we want to run the loop, a for loop is definitely more appropriate.

While Loops

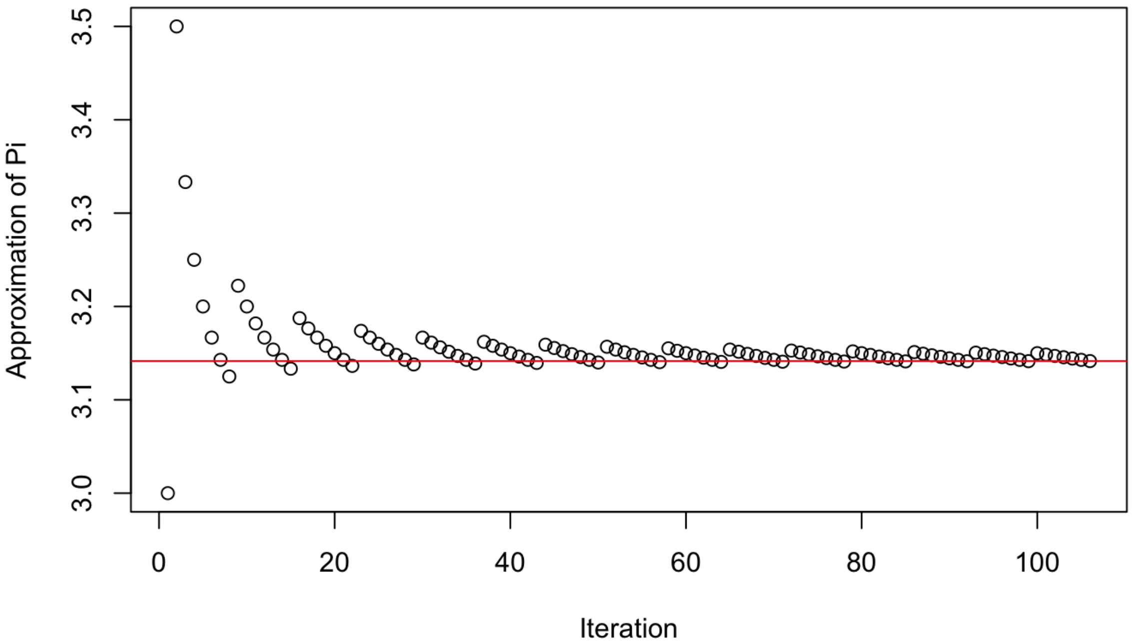
Let's look at a case where a while loop is needed. Below we'll try to approximate Pi using fractions.

```
tol<-1e-4 # how much approximation error are we willing to tolerate
err<-1 # initializing our error, just needs to be above the tolerance to begin
target<-pi-3 # taking just the bit of Pi after the decimal for now
numerator<-0 # initializing our numerator
denominator<-1 # initializing our denominator
approximation<-numerator/denominator # our approximation is just the ratio
dif<-approximation-target # The difference between our ratio and the target, note this value is signed

while(err > tol){
  denominator<-denominator+1 # update our denominator
  if(dif < 0){numerator<-numerator+1} # update our numerator if we were below the target
  approximation<-c(approximation,numerator/denominator) # Note we do this step strangly just to keep track of the appro
  dif<-tail(approximation,1)-target # update the difference
  err<-abs(dif) # taking the absolute value of the error
}

plot(3+approximation,xlab="Iteration",ylab="Approximation of Pi")
abline(h=pi,col="red")
```

While Loops



Approximation of Pi
3.0
3.1
3.2
3.3
3.4
3.5
0
20
40
60
80
100
Iteration

Functions

Functions

We’ve already seen a number of functions in R! For example:

```
is.character("123")
```

The function `` `is.character` `` takes the input given to it in the parentheses and returns `` `TRUE` `` or `` `FALSE` ``, depending on whether the input is of type character or not.

We can see what a function does by typing in `` `?` `` followed by the function name in the R console.

```
?is.character
```

We can write our own functions as well, and we will do that all the time. Why?

- Functions will make your code easier to understand.
- For repeated tasks, changes can be made once by editing a function instead of editing many distant chunks of code.

Functions

To define a function, you assign a variable name to a function object.

- Function take arguments, mandatory and optional.
- State what you want to return with `` `return()` ``, otherwise it will return the last expression it evaluates.

```r
# Computes mean and standard deviation of a vector, and optionally prints the results.
summarize_data <- function(x, print = FALSE){
  center <- mean(x)
  spread <- sd(x)
  if (print) {
    print(paste("Mean =", center, ", SD =", spread))
  }
  return(list(mean = center, sd = spread))
}
```

```r
summarize_data(rnorm(n = 500, mean = 4, sd = 1))
```

```
## $mean
## [1] 3.999052
## $sd
## [1] 0.9774615
```

Functions

Intermediate function steps are not saved to the global environment:

```
donothing <- function(x){
  x.new1 <- x^2
  return(x)
}
returnsomething <- function(x){
  x.new2 <- x^2
  return(x.new2)
}
```

```
x2 <- donothing(5); x2
x.new1
```

```
## [1] 5
## Error: object 'x.new1' not found
```

```
x2 <- returnsomething(5); x2
x.new2
```

```
## [1] 25
## Error: object 'x.new2' not found
```

Vectorization

Vectorization

Let's compare a loop to a built-in vectorized function:

```
n <- 1e6
x <- seq_len(n)

# Method 1: Loop
start1 <- Sys.time()
y1 <- rep(0,n)
for (i in 1:n) {
  y1[i] <- x[i]^2
}
finish1 <- Sys.time()

# Method 2: Built-in vectorized function
start2 <- Sys.time()
y2 <- x^2
finish2 <- Sys.time()
```

```
finish1-start1
## Time difference of 0.1533 secs
finish2-start2
## Time difference of 0.008269072 secs
```

Apply

When working with a function that is not vectorized, there is a family of R functions that allows to avoid using loops explicitly to the same effect.

- The apply family functions: `` `lapply` ``, `` `vapply` ``, `` `sapply` ``, `` `apply` ``, and a few other less often used versions are sometimes much faster than the equivalent for loops.
- They do generally allow for more compact code.

Apply

The function `apply` operates on matrices/arrays. In the following example, we obtain column sums of the matrix X. Note than MARGIN = 1 indicates rows and MARGIN = 2 indicates columns.

```
X <- matrix(sample(15), nrow = 3, ncol = 5); X
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   12    9    4   15    8
## [2,]   11    7    5   13   14
## [3,]    6    1   10    3    2
```

```
apply(X, MARGIN = 2, FUN = sum)
```

```
## [1] 29 17 19 31 24
```

The function `apply` can be used with user-defined functions.

```
apply(X, 2, function(x) sum(10*x + 2))
```

```
## [1] 296 176 196 316 246
```

Apply

The function `lapply()` is used to repeatedly apply a function to elements of a sequential object such as a vector, list, or data-frame (applies to columns). The output is a list with the same number of elements as the input object.

```
lapply(1:3, function(x) x^2)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
```

The function `sapply()` is the same as `lapply()` but returns a "simplified" output.

```
sapply(1:3, function(x) x^2)
```

```
## [1] 1 4 9
```