

Random Sampling and Loops

More on Visualization

- Find your plot in <https://r-graph-gallery.com/ridgeline-plot.html>
- How we can put two plots together?
- patchwork package

```
# install.packages("patchwork")  
library("patchwork")  
# without patchwork, + works as layering if possible or option chains  
p1 + p2 # side by side  
p1 | p2 # side by side  
p1 / p2 # stack  
(p1 + p2) / p3
```

More on Visualization

- Residual density and normal density
- Option 1: relocate and rescale normal distribution

```
model1 <- lm(Sepal.Length ~ Sepal.Width + Species, data = iris)
res <- residuals(model1)
# coef(model1) # coefficients
# fitted(model1) # fitted values Y_hat
# predict(model1, newdata) # predict with the model

ggplot() +
  geom_density(aes(res)) +
  stat_function(
    fun = dnorm # pdf of normal,
    args = list(mean = mean(res), sd = sd(res))
  )
```

Sampling using `sample()` function

- Base R comes with `sample()`
 - ▶ vector `x` - population
 - ▶ output size
 - ▶ probability vector `prob`
 - ▶ `replace = FALSE` sample with or without replacement

```
sample(x = 1:4, size = 2)
sample(x = 1:4, size = 2, prob = c(0.1, 0.2, 0.3, 0.4))
sample(x = 1:4, size = 10, prob = c(0.1, 0.2, 0.3, 0.4), replace = TRUE)
```

Setting a random seed

- R also has internal RNG (random number generator)
 - ▶ This means that numbers are near random but not completely random
 - ▶ We can control RNG by controlling the *first few inputs* -> this is a (random) seed
- `set.seed()` function.

```
set.seed(63130)
print(rnorm(2)) # draw random numbers from standard normal
```

```
set.seed(63130)
print(rnorm(2)) # draw random numbers from standard normal
```

In-class exercises:

- ① Sample *with* replacement from 1 to 1000 with size of 20
- ② Sample *without* replacement from 1 to 1000 with size of 20

Writing Your Own Functions

- Just like many programming languages you can write your own function in R
- Custom functions can make it easier for you to do repeated jobs
 - ▶ If you have to clean up very similar but different datasets
 - ▶ If you have to repeat regression and saving the output multiple times with different datasets

The Function Constructor

```
my_function <- function(arg1, arg2, ...) {  
  # the body of the function  
  # the jobs to be done  
  arg1[1] <- 2  
  return( list(arg1, arg2) ) # the return a value / list / vector  
}
```

- R Studio provides a cool way to write function:
 - ▶ Select part you want to convert into a function
 - ▶ Then go to Code menu
 - ▶ Extract Function

The Function Constructor

- Function without any arguments

```
roll <- function() {  
  die <- 1:6  
  dice <- sample(die, size = 2, replace = TRUE)  
  s <- sum(dice)  
  return(s) # return value  
}
```

The Function Constructor

- Function with an argument

```
roll <- function(die) {  
  dice <- sample(die, size = 2, replace = TRUE)  
  s <- sum(dice)  
  return(s) # return value  
}
```

- Function with an argument and its default

```
roll <- function(die = 1:6) {  
  dice <- sample(die, size = 2, replace = TRUE)  
  s <- sum(dice)  
  return(s) # return value  
}
```

The Function Constructor

- Function without return()
- Even without return(), the function may return the last assigned / printed values

```
# with cat
roll1 <- function(die = 1:6) {
  dice <- sample(die, size = 2, replace = TRUE)
  s <- sum(dice)
  cat(s) # another way to print
}
```

```
r <- roll1() # NULL
```

```
# with print
roll2 <- function(die = 1:6) {
  dice <- sample(die, size = 2, replace = TRUE)
  s <- sum(dice)
  print(s)
}
```

```
r <- roll2() # a number
```

In-class exercises:

- 1 Write a function called `my_sampling()` that samples without replacement from 1 to 100 inclusive of a given size. The function should take one argument that specifies the size of the returning vector.
- 2 Write a function that randomly samples from the TFR column in the `worldTFR` dataset. The users should be able to modify the size of the sample, and whether to sample with or without replacement.

Control Statements

- if statement
- for loop
- while loop

if Statement in R

- *if* statement: conditional execution of code
 - ▶ The test condition that evaluates to a logical output (TRUE = 1 / FALSE = 0).
 - ▶ It runs the enclosed code block if the condition evaluates to TRUE.
 - ▶ It skips the code block if the condition evaluates to FALSE.
- use *else if* for the second condition
- use *else* to take care of otherwise cases

```
if (test_expression) {  
  # code  
} else if (test_expression2) {  
} else {  
}
```

if Statement in R

```
a <- 88
b <- 100

if (a > b) {
  print("a is greater than b")
} else {
  print("b is greater than a")
}
```

if Statement in R

- Use in value recoding

```
df <- longley

i <- 3
if (df$Year[i] == 1947) {
  print("Great, it's 1947! I will give it a value great")
  df$Year[i] <- "great"
} else if (df$Year[i] %in% c(1946, 1948)) {
  print("Well, it's not 1947 but It's okay")
  df$Year[i] <- "okay"
} else {
  print("What a disappointment! I feel bad")
  df$Year[i] <- "bad"
}
```


ifelse() function

- Convenient in value recoding and simple if statement job
- `ifelse(condition, value_true, value_false)`

```
ifelse(a < 7, "a is less than 7", "a is greater than 7")
```

```
df$PopMean <- ifelse(df$Population > mean(df$Population), TRUE, FALSE)
```

for Loop in R

- *for* loop repeats tasks for a variable in a sequence

```
for (var in sequence) {  
  # code to repeat  
  var <- 1  
  df["var"] <- 1  
}
```

for Loop in R

```
vec <- c(2, 7)
vec2 <- c(10:17)
for (i in vec) {
  print(vec2[i])
}
```

for Loop in R

- Iterate through the entire dataframe.

```
# This will print out all Years
```

```
for (i in 1:dim(df)[1]) {  
  print(df$Year[i])  
}
```

```
df$ratio <- NA # initialize with NAs
```

```
for (i in 1:dim(df)[1]) {  
  df$ratio[i] <- df$Employed[i] / (df$Unemployed[i] + df$Employed[i])  
}
```

while Loop in R

- *while* loops work similarly to until the test condition becomes FALSE

```
while (test_expression) {  
  code_to_repeat  
}
```

- Never run

```
while (FALSE) {  
  print("F")  
}
```

- Run forever

```
while (TRUE) {  
  print("T")  
}
```

while Loop in R

- A valid use of while loop condition should be changed by the repeated code or time etc.

```
i <- 1 # i is initialized to 1
while (i <= 6) {
  print(i)
  i <- i + 1 # increment i
}
```

for or *while*?

- Personal choice
- You can almost always rewrite *for* with *while* and *while* with *for*

In-class exercises:

- 1 Rewrite the 1947 example. This time use `else if` *twice*
- 2 Write a *for* loop that takes 10 random samples of size 1 without replacement, from the range 1 to 100.
- 3 Do the same thing with *while* loop.

Bootstrap

- We will learn more details about bootstrap in QPM1
- The logic goes:
 - ▶ Assume that the sample we have is a very good representation of the population.
 - ▶ Take random (sub)sample *with replacement* from the sample will be almost same as taking multiple samples from the population
 - ▶ Summarizing the (sub)samples will tell us something about the population

Bootstrap 1

- `sample()` and for loop.

```
worldTFR <- read.csv("./worldTFR.csv")
worldTFR <- na.omit(worldTFR)

B <- 100 # number of bootstrap
# Create numeric vector size of B to store results
bootstrapped <- numeric(B)
set.seed(63130)
for (b in 1:B) {
  # take random index with replacement
  idx <- sample(1:dim(worldTFR)[1], replace=TRUE)
  # subset the data and select the column
  temp <- worldTFR[idx, "GDPpc"]
  bootstrapped[b] <- mean(temp) # Store bth mean
}
```

Bootstrap 2

- boot package
- `boot::boot()` requires a `statistic` argument, which is a function that takes `data` and `index` as arguments -> most times we need to define a custom function

```
# install.packages("boot")
library(boot)

# Define the function first
mean_fn <- function(data, indices) {
  return( mean(data$GDPpc[indices]) )
}

bootstrapped2 <- boot(worldTFR, statistic = mean_fn, R = 100)
```

Bootstrap 2

- boot package also provides useful `boot.ci()` function, which calculates different types of confidence intervals.

```
boot.ci(bootstrapped2)
```

In-class exercises:

- 1 Bootstrap the mean of `MtoF04` in `worldTFR`.

Simulation

- for loop like we did
- Another base R friendly way is using a function `replicate()`

```
set.seed(63130)
R <- 1000
results <- numeric(R) # numeric vector of size 1000
for (i in 1:R) {
  results[i] <- roll()
}

results <- replicate(R, roll())
```