Automation

Main ideas

- Use `for` to iterate code
- Use `map_*()` functions to iterate across data frames
- Iteration best practices

Packages

```
library(tidyverse)
library(broom)
```

Notes

`for` loops

- One tool for reducing duplication is functions. Another tool is iteration via `for` loops.
- This will help if you need to do the same thing to multiple inputs.
- For example, we can iterate through elements of a vector and evaluate code based on each vector element's value.

Motivating example

Let's create a small tibble `x`.

```
x <- tibble(
  col_a = c(3, -1, 0, 10),
  col_b = c(2, -2, 2, -2),
  col_c = c(8, sqrt(131), log(4), 33),
  col_d = 1:4
)
```

Suppose you want to compute the mean of each column in `x`.

A first attempt might be as follows.

```
mean_col_a <- x %>%
  pull(1) %>%
  mean()

mean_col_b <- x %>%
  pull(2) %>%
  mean()

mean_col_c <- x %>%
  pull(3) %>%
  mean()
```

```
mean_col_d <- x %>%
  pull(4) %>%
  mean()

c(mean_col_a, mean_col_b, mean_col_c, mean_col_d)
```

```
#> [1]  3.00000  0.00000 13.45795  2.50000
```

```
x %>%
  summarise(
    mean_col_a = mean(col_a),
    mean_col_b = mean(col_b),
    mean_col_c = mean(col_c),
    mean_col_d = mean(col_d)
  )
```

```
#> # A tibble: 1 x 4
#>   mean_col_a mean_col_b mean_col_c mean_col_d
#>        <dbl>      <dbl>      <dbl>      <dbl>
#> 1          3          0       13.5        2.5
```

Or a non-tidyverse way may be as follows.

```
mean(x$col_a)
```

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

```
mean(x$col_b)
```

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

```
mean(x$col_c)
```

```
#> [1] 13.45795
```

```
mean(x$col_d)
```

```
#> [1] 2.5
```

This is still a lot of copied code. Imagine if we had a tibble with 1,000 columns.

How can we automate this process?

`for`-loop construction and syntax

Looking at our previous code, we see that the only variation is with regards to the column index being pulled. A `for` loop can easily automate our process from the previous slide.

First, create an output object. This is where we will save our results.

Second, define the loop sequence. Here `i` is a looping variable, in each run of the loop `i` will be assigned a different value in the vector `c(1, 2, 3, 4)`.

Third, add the loop's body. This is the code that does the work.

```
# allocate output object, call it results
results <- numeric(4)

# define loop and looping sequence
for (i in c(1, 2, 3, 4)) {
  # print(i)
```

```r
  cat("The value of i is", i, "\n")
  results[i] <- x %>%
    pull(i) %>%
    mean()
  cat("The mean is", results[i], "\n")
  Sys.sleep(3)
}
```

```
#> The value of i is 1
#> The mean is 3
#> The value of i is 2
#> The mean is 0
#> The value of i is 3
#> The mean is 13.45795
#> The value of i is 4
#> The mean is 2.5
```

```r
results <- numeric(length(x))
for (i in 1:length(x)) {
  results[i] <- x %>%
    pull(i) %>%
    mean()
}
```

This is a small example, but it's easy to see the benefits of using a `for` loop if we needed to scale this computation to 100 columns. All we would have to change is our loop sequence.

Example

Let's create a function called `length_unique()`. It will determine the number of unique values in a vector.

```r
length_unique <- function(x) {
  length(unique(x))
}
```

```r
a <- c(1, 1, 1, 1, 5, 2, 2, 2, 0)
length(a)
```

```
#> [1] 9
```

```r
length_unique(a)
```

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

Write a loop that outputs the number of unique values in each column in a data frame. Test your loop out on the `storms` data frame in `dplyr`.

```r
output <- numeric(length(storms))

for (j in seq(length(storms))) {
  output[j] <- storms %>%
    pull(j) %>%
    length_unique()
}

output
```

```
#>  [1] 198  41  10  31  24 403 856   3   7  31 124 109  35
```

`for`-loop best practices

1. Always initialize your result vector with as many elements as you will have results before you begin your loop.
2. Avoid using loops when vectorization works. For example, in R we do not need a loop to sum all the elements of `c(4, 1, -3, 0, 44, 9)`, just use `sum()`.

A failure to allocate result object

```
results <- NULL
x <- rnorm(n = 100000)

system.time({
for (i in 1:length(x)) {
  results <- c(results, x[i] ^ 2)
}
})
```

```
#>    user  system elapsed
#>  16.505   7.569  24.291
```

```
results <- numeric(100000)
x <- rnorm(n = 100000)

system.time({
for (i in 1:length(x)) {
  results[i] <- x[i] ^ 2
}
})
```

```
#>    user  system elapsed
#>   0.009   0.000   0.009
```

`purrr` and `map_*()`

The `purrr` package has functions that facilitate automation and mostly eliminate your need to write `for`-loops.

Recall our small data frame `x`.

```
x <- tibble(
  col_a = c(3, -1, 0, 10),
  col_b = c(2, -2, 2, -2),
  col_c = c(8, sqrt(131), log(4), 33),
  col_d = 1:4
)
```

Instead of a `for`-loop to compute the mean of each column, we could use `map_dbl()`.

```
map_dbl(x, mean)
```

```
#>    col_a    col_b    col_c    col_d
#>  3.00000  0.00000 13.45795  2.50000
```

What is this doing?

```
c(mean(x$col_a), mean(x$col_b), mean(x$col_c), mean(x$col_d))
```

```
#> [1]  3.00000  0.00000 13.45795  2.50000
```

Function `map_dbl()` expects each function call to result in a single value (vector of length 1). Depending on what you expect your output to be you can use some similar variants:

- `map_lgl()` for logical results
- `map_chr()` for character results
- `map_dbl()` for double results
- `map_int()` for integer results

Example

Suppose you want to apply a function to each column of your data frame where it only modifies the data frame. For example, suppose you want to standardize your variables before fitting a linear model. Function `modify()` from `purrr` can help you do just that.

Fit a linear model with `mpg` as the response and `hp` and `wt` as predictors from `mtcars`. Standardize all variables.

```
standardize <- function(x) {
  (x - mean(x)) / sd(x)
}
```

```
mtcars %>%
  select(mpg, hp, wt) %>%
  modify(standardize) %>%
  lm(mpg ~ -1 + hp + wt, data = .) %>%
  tidy()
```

```
#> # A tibble: 2 x 5
#>   term  estimate std.error statistic     p.value
#>   <chr>    <dbl>     <dbl>     <dbl>       <dbl>
#> 1 hp      -0.361     0.101     -3.58 0.00120
#> 2 wt      -0.630     0.101     -6.23 0.000000727
```

Practice

(1) Write a `for`-loop to check the type of each variable in `gapminder` from the `gapminder` package. Function `typeof()` can be used to check the type.

```
library(gapminder)

types <- character(length = length(gapminder))

for (i in 1:length(gapminder)) {
  types[i] <- gapminder %>% pull(i) %>% typeof()
}
```

(2) Redo the previous exercise, but this time use a `map_*()` variant.

```
map_chr(gapminder, typeof)
```

```
#>   country continent      year   lifeExp       pop gdpPercap
#> "integer" "integer" "integer"  "double" "integer"  "double"
```

(3) Write a function that returns the number of `NA` values in a vector. Use a `purrr` function to then check how many `NA` values exist for each variable in a data frame. Test your code on the `tibble` below.

```
df <- tibble(
  x = c(6, NA, 9, 10, 4, 1),
  y = c("a", "z", NA, NA, NA, "e"),
```

```r
  z = rnorm(6)
)
```

```r
count_na <- function(x) {
  sum(is.na(x))
}

map_dbl(df, count_na)
```

```
#> x y z
#> 1 3 0
```