R Functions

Why functional programming?

Vanilla cupcakes

Ingredients:

1. Flour
2. Sugar
3. Baking powder
4. Unsalted butter
5. Milk
6. Egg
7. Vanilla

Directions:

1. Preheat oven to 350°F
2. Put the flour, sugar, baking powder, salt, and butter in a free standing electric mixer with a paddle attachment, beat on slow speed until sandy consistency is obtained
3. Whisk ingredients 5-7 together
4. Spoon batter, bake for 20 minutes

Source: The hummingbird bakery cookbook

Chocolate cupcakes

Ingredients:

1. Cocoa
2. Sugar
3. Baking powder
4. Unsalted butter
5. Milk
6. Egg
7. Vanilla

Directions:

1. Preheat oven to 350°F
2. Put the cocoa, sugar, baking powder, salt, and butter in a free standing electric mixer with a paddle attachment, beat on slow speed until sandy consistency is obtained Whisk ingredients 6-8 together
3. Spoon batter, bake for 20 minutes

Source: The hummingbird bakery cookbook

Chocolate cupcakes

Ingredients:

1. Cocoa
2. Sugar
3. Baking powder
4. Unsalted butter
5. Milk
6. Egg
7. Vanilla

Directions:

1. Preheat oven to 350°F
2. Put the cocoa, sugar, baking powder, salt, and butter in a free standing electric mixer with a paddle attachment, beat on slow speed until sandy consistency is obtained Whisk ingredients 6-8 together
3. Spoon batter, bake for 20 minutes

Source: The hummingbird bakery cookbook

Vanilla cupcakes

1. Rely on domain knowledge

Ingredients:

1. Flour
2. Sugar
3. Baking powder
4. Unsalted butter
5. Milk
6. Egg
7. Vanilla

Directions:

1. Preheat oven to 350°F
2. Put the flour, sugar, baking powder, salt, and butter in a free standing electric mixer with a paddle attachment, beat on slow speed until sandy consistency is obtained
3. Whisk ingredients 5-7 together
4. Spoon batter, bake for 20 minutes

Source: The hummingbird bakery cookbook

Vanilla cupcakes

1. Rely on domain knowledge

Ingredients:

1. Flour
2. Sugar
3. Baking powder
4. Unsalted butter
5. Milk
6. Egg
7. Vanilla

Directions:

1. Preheat
2. Mix, whisk, and spoon
3. Bake

Vanilla cupcakes

2. Use variables

Ingredients:

1. Flour
2. Sugar
3. Baking powder
4. Unsalted butter
5. Milk
6. Egg
7. Vanilla

Directions:

1. Preheat
2. Mix, whisk, and spoon
3. Bake

Vanilla cupcakes

2. Use variables

Ingredients:

1. Flour
2. Sugar
3. Baking powder
4. Unsalted butter
5. Milk
6. Egg
7. Vanilla

Directions:

1. Preheat
2. Mix dry ingredients, whisk wet ingredients, and spoon
3. Bake

Cupcakes

3. Extract out common code

Directions:

1. Preheat
2. Mix dry ingredients, whisk wet ingredients, and spoon
3. Bake

Vanilla:	**Chocolate:**
1. Flour ⟷	1. Cocoa
2. Sugar	2. Sugar
3. Baking powder	3. Baking powder
4. Unsalted butter	4. Unsalted butter
5. Milk	5. Milk
6. Egg	6. Egg
7. Vanilla	7. Vanilla

`for` loops are like pages in the recipe book

```
> out1 <- vector("double", ncol(mtcars))

for(i in seq_along(mtcars)) {
  out1[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}

> out2 <- vector("double", ncol(mtcars))

for(i in seq_along(mtcars)) {
  out2[[i]] <- median(mtcars[[i]], na.rm = TRUE)
}
```

`for` loops are like pages in the recipe book

```
> out1 <- vector("double", ncol(mtcars))

for(i in seq_along(mtcars)) {
  out1[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}

> out2 <- vector("double", ncol(mtcars))

for(i in seq_along(mtcars)) {
  out2[[i]] <- median(mtcars[[i]], na.rm = TRUE)
}
```

- Emphasizes the objects, pattern of implementation
- Hides actions

`for` loops are like pages in the recipe book

```
> out1 <- vector("double", ncol(mtcars))

for(i in seq_along(mtcars)) {
  out1[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}

> out2 <- vector("double", ncol(mtcars))

for(i in seq_along(mtcars)) {
  out2[[i]] <- median(mtcars[[i]], na.rm = TRUE)
}
```

- Emphasizes the objects, pattern of implementation
- Hides actions

Functional programming is like the meta-recipe

```
> library(purrr)

> means <- map_dbl(mtcars, mean)

> medians <- map_dbl(mtcars, median)
```

- Give equal weight to verbs and nouns
- Abstract away the details of implementation

Let's practice!

Functions can be arguments too

Removing duplication with arguments

```
> f1 <- function(x) abs(x - mean(x)) ^ 1
> f2 <- function(x) abs(x - mean(x)) ^ 2
> f3 <- function(x) abs(x - mean(x)) ^ 3
```

Removing duplication with arguments

```
> f1 <- function(x) abs(x - mean(x)) ^ power
> f2 <- function(x) abs(x - mean(x)) ^ power
> f3 <- function(x) abs(x - mean(x)) ^ power
```

Removing duplication with arguments

```
> f1 <- function(x, power) abs(x - mean(x)) ^ power
> f2 <- function(x, power) abs(x - mean(x)) ^ power
> f3 <- function(x, power) abs(x - mean(x)) ^ power
```

Functions can be arguments too

```
col_median <- function(df) {
  output <- numeric(length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
}
```

```
col_sd <- function(df) {
  output <- numeric(length(df))
  for (i in seq_along(df)) {
    output[i] <- sd(df[[i]])
  }
  output
}
```

```
col_mean <- function(df) {
  output <- numeric(length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
```

Functions can be arguments too

```
col_median <- function(df) {
  output <- numeric(length(df))
  for (i in seq_along(df)) {
    output[i] <- fun(df[[i]])
  }
  output
}
```

```
col_sd <- function(df) {
  output <- numeric(length(df))
  for (i in seq_along(df)) {
    output[i] <- fun(df[[i]])
  }
  output
}
```

```
col_mean <- function(df) {
  output <- numeric(length(df))
  for (i in seq_along(df)) {
    output[i] <- fun(df[[i]])
  }
  output
}
```

Functions can be arguments too

```
col_median <- function(df, fun) {
  output <- numeric(length(df))
  for (i in seq_along(df)) {
    output[i] <- fun(df[[i]])
  }
  output
}
```

```
col_sd <- function(df, fun) {
  output <- numeric(length(df))
  for (i in seq_along(df)) {
    output[i] <- fun(df[[i]])
  }
  output
}
```

```
col_mean <- function(df, fun) {
  output <- numeric(length(df))
  for (i in seq_along(df)) {
    output[i] <- fun(df[[i]])
  }
  output
}
```

Functions can be arguments too

```
col_summary <- function(df, fun) {
  output <- numeric(length(df))
  for (i in seq_along(df)) {
    output[i] <- fun(df[[i]])
  }
  output
}
```

```
> col_summary(df, fun = median)
> col_summary(df, fun = mean)
> col_summary(df, fun = sd)
```

Let's practice!

Introducing purrr

Passing functions as arguments

```
> sapply(df, mean)
          a          b          c          d 
 0.0643872 -0.1630165 -0.1057590  0.0406435 
```

```
> col_summary(df, mean)
[1]  0.0643872 -0.1630165 -0.1057590  0.0406435
```

```
> library(purrr)
> map_dbl(df, mean)
          a          b          c          d 
 0.0643872 -0.1630165 -0.1057590  0.0406435 
```

Every map function works the same way

```
map_dbl(.x, .f, ...)
```

1. Loop over a vector `.x`
2. Do something to each element `.f`
3. Return the results

The map functions differ in their return type

There is one function for each type of vector:

- `map()` returns a list
- `map_dbl()` returns a double vector
- `map_lgl()` returns a logical vector
- `map_int()` returns a integer vector
- `map_chr()` returns a character vector

Different types of vector input

```
map(.x, .f, ...)
```

.x is always a vector

```
> df <- data.frame(a = 1:10, b = 11:20)
> map(df, mean)
$a
[1] 5.5

$b
[1] 15.5
```

Data frames, iterate over columns

Different types of vector input

```
> l <- list(a = 1:10, b = 11:20)
> map(l, mean)
$a
[1] 5.5

$b
[1] 15.5
```

Lists, iterate over elements

Different types of vector input

```
> vec <- c(a = 1, b = 2)
> map(vec, mean)
$a
[1] 1

$b
[1] 2
```

Vectors, iterate over elements

Advantages of the map functions in purrr

- Handy shortcuts for specifying `.f`
- More consistent than `sapply()`, `lapply()`, which makes them better for programming (Chapter 5)
- Takes much less time to solve iteration problems

Let’s practice!

Shortcuts for specifying `.f`

Specifying .f

```
> map(df, summary)
```

An existing function

```
> map(df, rescale01)
```

An existing function you defined

```
> map(df, function(x) sum(is.na(x)))
```

An anonymous function defined on the fly

```
> map(df, ~ sum(is.na(.)))
```

An anonymous function defined using a formula shortcut

Shortcuts when .f is [[

```
> list_of_results <- list(
      list(a = 1, b = "A"),
      list(a = 2, b = "C"),
      list(a = 3, b = "D")
  )


> map_dbl(list_of_results, function(x) x[["a"]])
[1] 1 2 3
```

An anonymous function

```
> map_dbl(list_of_results, "a")
[1] 1 2 3
```

Shortcut: string subsetting

```
> map_dbl(list_of_results, 1)
[1] 1 2 3
```

Shortcut: integer subsetting

A list of data frames

```
> cyl <- split(mtcars, mtcars$cyl)
> str(cyl)
List of 3
 $ 4:'data.frame': 11 obs. of  11 variables:
  ..$ mpg : num [1:11] 22.8 24.4 22.8 32.4 30.4 33.9 21.5 ...
  ..$ cyl : num [1:11] 4 4 4 4 4 4 4 4 4 4 ...

  ...

 $ 6:'data.frame': 7 obs. of  11 variables:
  ..$ mpg : num [1:7] 21 21 21.4 18.1 19.2 17.8 19.7 ...
  ..$ cyl : num [1:7] 6 6 6 6 6 6 6 ...

  ...

 $ 8:'data.frame': 14 obs. of  11 variables:
  ..$ mpg : num [1:14] 18.7 14.3 16.4 17.3 15.2 10.4 10.4 14.7 ...
  ..$ cyl : num [1:14] 8 8 8 8 8 8 8 8 8 8 ...
```

Split the data frame `mtcars` based on the unique values in the `cyl` column

A list of data frames

```
> cyl[[1]]
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Goal

- Fit regression to each of the data frames in `cyl`
- Quantify relationship between `mpg` and `wt`

```
# Slopes for regressions on mpg on weight for each cylinder class
        4         6         8 
-5.647025 -2.780106 -2.192438
```

Let’s practice!