# fpr

ROSE

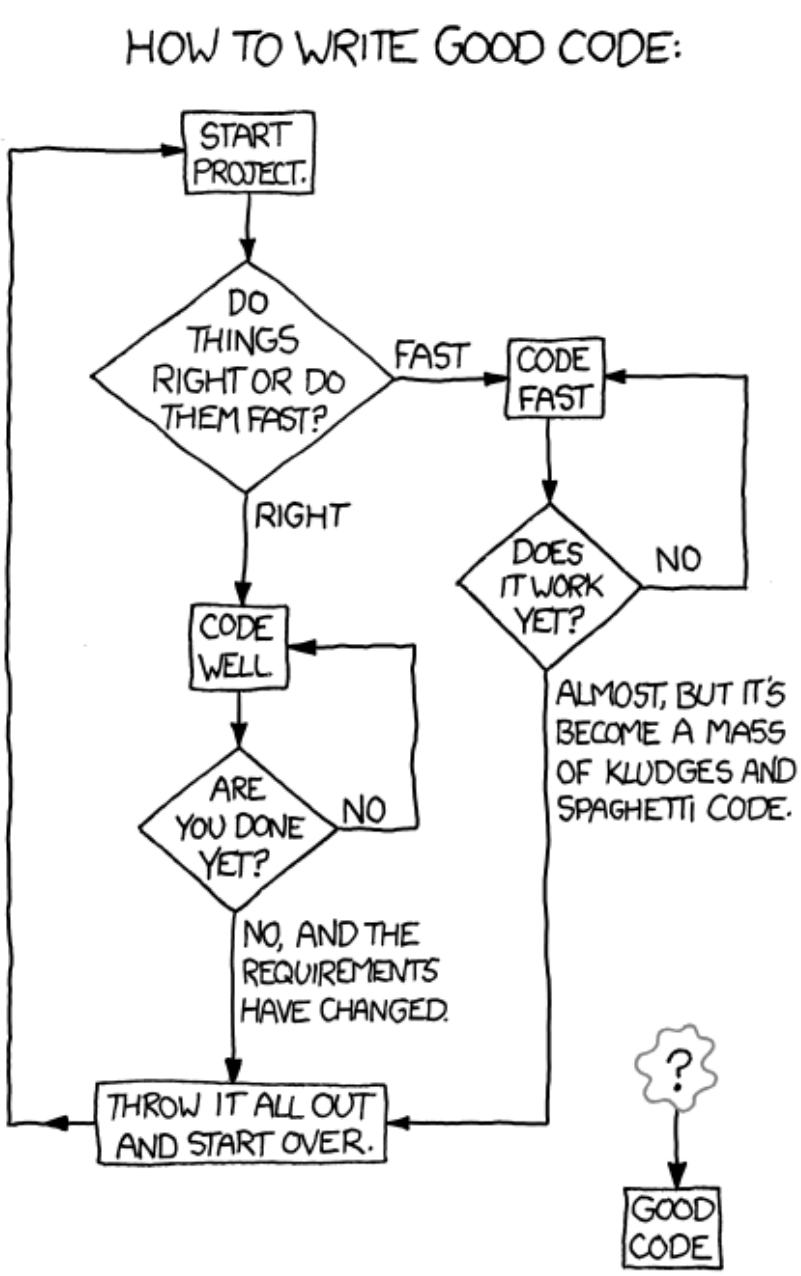
October 22, 2020

## Contents



# 1 Functional Programming with R

---


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
?
THROW IT ALL OUT AND START OVER.
GOOD CODE

## 2 What is Functional Programming?

From Wiki:

- A programming language paradigm in which function definitions are trees of expressions that each return a value.

From me:

At its broadest level, this means that we want to make programs which are compositions of functions for which we can reason about the input and output.

In our case:

**data %>% transformation$_1$() %>% transformation$_2$() %>% ... %>% transformation$_n$() -> output**

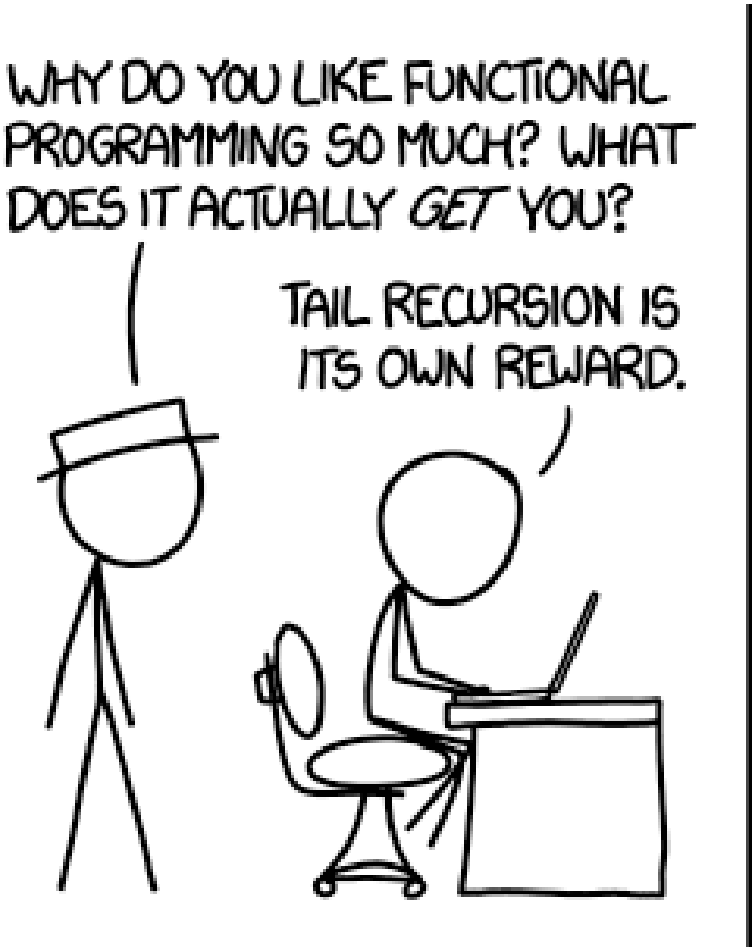
Some of the techniques and methodologies that hallmark functional programming languages are:

1. immutable data
2. distinction between pure and impure functions
3. functions as first-class objects
4. functional style is (usually) a declarative style using function composition

### 2.1 Why Bother?

Functional programs:

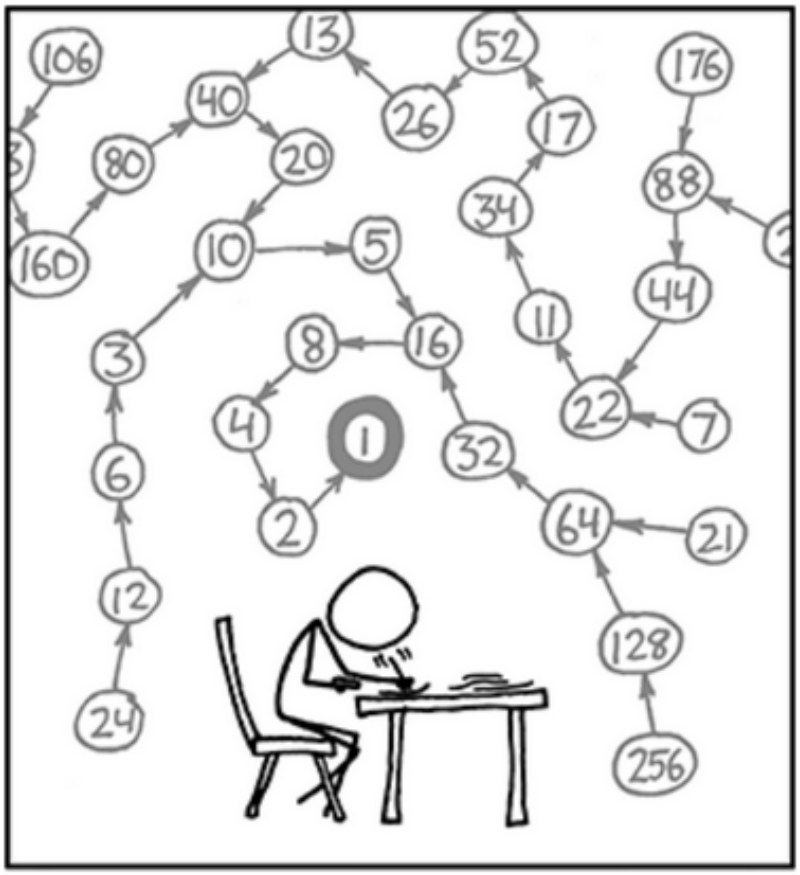
- tend to be less error prone and more concise
- favor high amounts of abstraction, which can increase modularity
- is a great fit for data analysis
- **Most importantly, it is quite fun**




## 3 Immutable Data

Immutable data means that we have an emphasis on not actively **updating** variables. Once a variable x is assigned, it shall remain that assignment until it has no use. This is to prevent bugs down the road, as it allows us to evaluate our code through a substitution model (i.e. we can replace function calls with their values).

### 3.1 Example : Collatz Conjecture




THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

```
## Collatz Conjecture
hailstone <- function(n) {
  stopifnot(n >= 1)
  if (n == 1) return(1)
  if (n %% 2 == 0) return(c(n, hailstone(n / 2)))
```

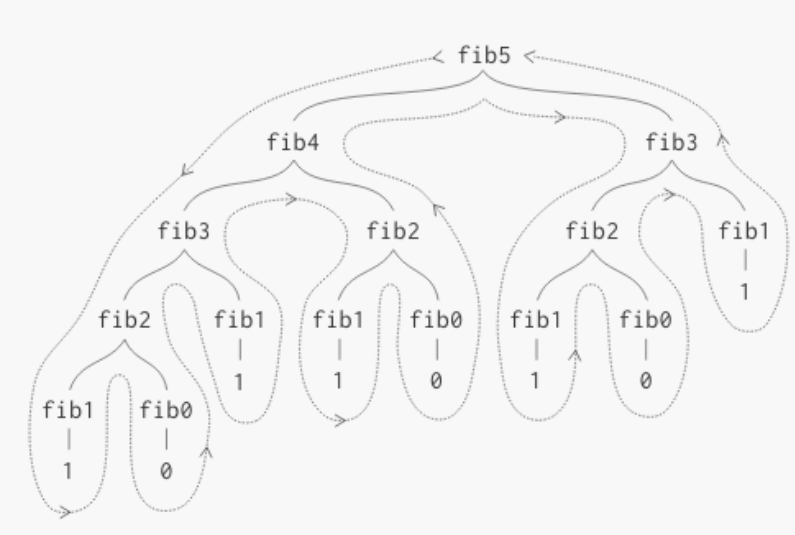
```
  else return(c(n, hailstone(3 * n + 1)))
}

(x <- hailstone(12))
```

Here we see a recursive function with only 1 state variable n. What makes this important is that for whatever n we give it, each time we run the algorithm for that same n we get the same result.

When we unfold this operation, we get something like

```
## for hailstone(4)
hailstone(4)
c(4, hailstone(2))
c(4, c(2, hailstone(1)))
c(4, 2, 1)
```

Here is a different illustrative example of the tree structure with the Fibonacci sequence:




- In contrast, imperative languages are a sequence of statements which change the state of a program. There is a focus on having variables (data structures) which are **updated.**

```
hailstone_imp <-function(x, n) {
```

```
  stopifnot(n >= 1)
  ## define counter for while loop
  i <- 1
  while (n > 1)
  {
    x[i] <- n
    i <- i + 1
    n <- ifelse(n %% 2, 3*n + 1, n/2)
  }
  x[i] = 1
  return(x)
}

## define a vector to hold results
y <- c()
(y <- y %>% hailstone_imp(12))

## in another language we might see
## y = c()
## y = y.hailstone_imp(12)
```

Here we have a few state variables:

- i
- n
- our vector y

What if I want the results of the hailstone sequence at a later time, but I don't remember the state of y and assume it is ok?

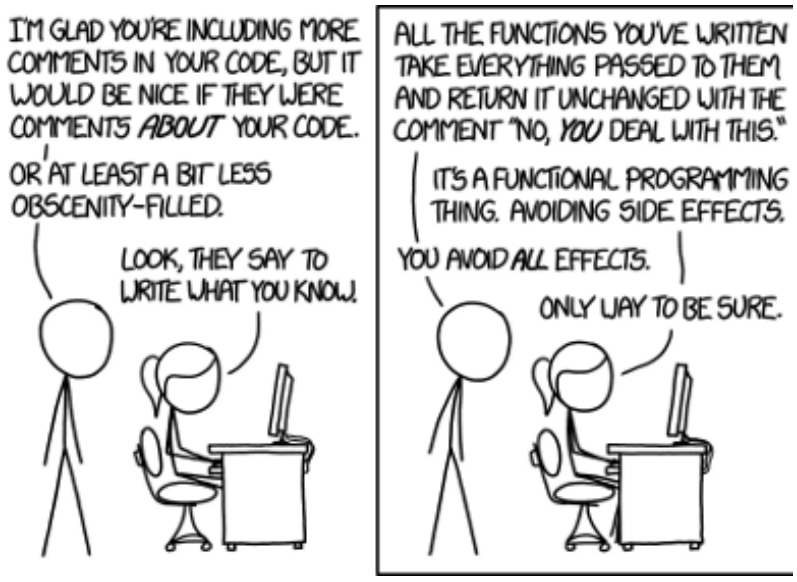
```
(y <- y %>% hailstone_imp(4))
```

We see that our biggest problem is that state of the vector y. When we pass y to the hailstone sequence function, we get a bug. We assumed something about the state, but didn't have a way to guarantee that it would work correctly. This leads to us needing to spend more time writing unit tests and defensively programming.

### 3.2 Side Quest:

```
1:10000 %>%
  tibble("hailstone_length" = map(., ~ length(hailstone(.x))), "n" = .) %>%
  unnest(cols = c(hailstone_length)) %>%
  ggplot(aes(x = n, y = hailstone_length)) +
  geom_point(shape = 5, color = "mediumpurple") +
  ggdark::dark_theme_gray() +
  xlab("Number") + ylab("Hailstone Sequence Length") +
  ggtitle("Sequence Length ~ Number")
```

## 4 Pure and Impure Functions




### 4.1 Why do we care about whether the function has state variables?

- A function that is pure will always return the exact same results when given the same input.
- A function that is impure often relies on some value that exists in the environment and may not return the same result given the same input. This might be due to many different things, like:

- a variable having been updated or overwritten by something
- corrupted data
- the operating system state (the classic "it worked on my machine")
- the time
- the seed for RNG

### 4.2 Example : Pure / Impure Processes

```
## pure
(1:10 %>%
  Filter(f = function(x) x %% 2 == 0) %>%
  Map(f = function(x) x * 10) %>%
  Reduce(f = function(a, b) a + b) -> result)

## impure
num_list <- 1:10
result <- 0

for (i in num_list) {
  if (num_list[i] %% 2 == 0) {
    result <- result + num_list[i] * 10
  }
}

result
```

In our first example, our process does not depend on the value of result.

In our second example, our process requires us to reset the value of result to 0 before proceeding, otherwise we get the wrong answer. Thus for the same values of "$\text{num}_{\text{list}}$" (1:10), we can get different answers depending on the value of result.

This is relatively benign here, but it scales very poorly and makes reasoning about a program difficult and directly linked to its current state. When we wish to think about what our program will do depending on our input, we now much consider what state each of the variables is in.

Instead of dealing with this problem of state head-on, it is worthwhile to try to isolate reduce the impurity of your functions by separating

and/or explicitly notating side effects. This means using pure functions when you can, and using **persistent data structures** if possible. Persistent data structures preserve the previous version of themselves when modified.

The state problem is slightly alleviated by using closures

# 5 Scopes and Closures

**"An object is data with functions. A closure is a function with data." — John D. Cook**

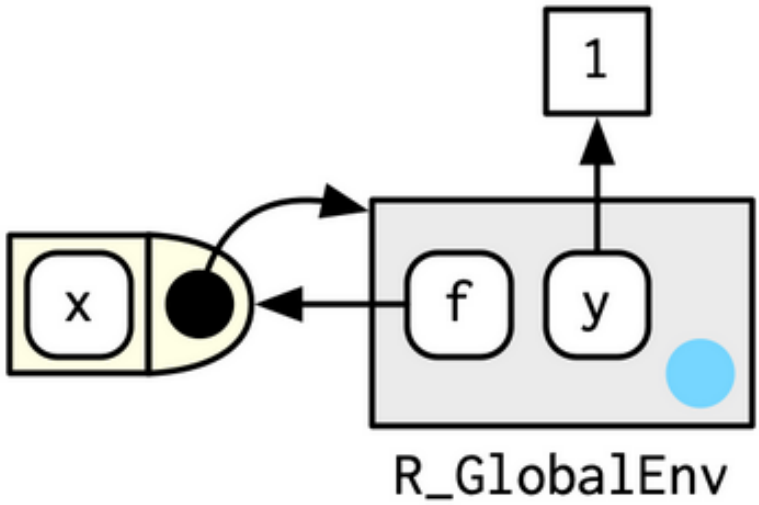
A **scope** is something functions or expressions are associated with that tells them what values variables refer to. It is used to figure out what environment expressions are evaluated in. A variable name can be used in many places in a program – the scope helps R figure out what value you wanted.

A **closure** is a function with an associated scope. In this case, all functions in R are closures. By default they have access to the local environment of the function and the global environment. Variables defined in the scope of the function remain in the scope of the function.

As an example:

```
y <- 1
f <- function(x) x + y
```

Has the following environment:




- How can we use closures to alleviate the problem we had before?

```
do_stuff <- function (num_list){
  result <- 0 ## explicit assignment

  for (i in num_list) {
    if (num_list[i] %% 2 == 0) {
      result <- result + num_list[i] * 10
    }
  }
  result
}

result

(do_stuff(1:10))
```

Now we have effectively made a new environment to contain our result variable. Whenever we call this function, our result **inside** the function scope is set to 0. This clears the problem without affecting our result variable in the global environment.

This seems very common sense, but it begins to break down once you have reliance on objects. An object typically has internal variables which has getters and setters that change that state.

A traditional object looks something like:

```
object_name <- object(
  ## state variables
  var1 <- 1

  ## methods
  get_var1 <- function() {
    return(var1)
  }

  set_var1 <- function(new_val) {
    var1 <<- new_val
  }

  some_kinda_action <- function() {
    var1 * 2
  }
)
```

Let's see why this could be problematic:

```
## make object
object_name <- R6Class(classname = "something",
                       list(
                         ## state variables
                         var1 = 1,
                         ## methods
                         get_var1 = function() {
                           return(self$var1)
                         },
                         set_var1 = function(num) {
                           self$var1 <- num
                           print("ok")
                         },
                         some_important_action = function() {
                           self$var1 * 2
                         }
                       ))
```

```
## construct object
object_example <- object_name$new()

## run some
object_example$get_var1()
object_example$some_important_action()

object_example$get_var1()
object_example$set_var1(2)
object_example$some_important_action()

object_example$get_var1()
```

We see that our computation is having the same problem we saw with the unscoped loop – we rely on the state variables and this could lead to bugs.

## 5.1 Can we write programs using only pure functions?

Yes, but it is quite restrictive. Here are some impure functions:

- data i/o
- writing/printing to the console
- declaring variables
- plotting
- getting system time
- random number generation
- reading from a production database
- calling a system command that is impure

## 5.2 So what do we do to avoid bugs?

Some good rules of thumb: —

- avoid updating variables once they are defined
- Use closures instead of global variables for more safety

- don't update any data without certainty that it won't break things later. Use mutate sparingly.
  - When you do use mutate, mutate to a new column instead of updating
- actively notate/separate your side effect functions from your computation functions (example later)
- Use persistent / immutable data structures if possible
- chain together pipelines of pure functions and make your compositions human understandable

# 6 First Class & Higher Order Functions

## 6.1 First Class Functions

Functions are first class citizens in both R and Python.

This means that functions can take other functions as arguments or return them as results.

Arguably the most important uses of first class functions is for **functionals**. A functional takes a function as an input and returns a vector as an output.

```
## map replaces loops
map(1:10, ~ .x * 2)

## map essentially does this on a list
apply_it <- function(data, operation) {
  operation(data)
}

apply_it(1:10, function(x) exp(x))

## try a bunch of operations
list("exp" = exp,
     "times_two" = (function(x) x * 2),
     "square_root" = sqrt,
     "times_pi" = (function(x) x * pi)) %>%
  map(., ~ apply_it(1:10, .x))
```

```
## filter filters
seq(from = 0.1, to = 1.5, by = 0.1) %>%
  Filter(f = function(x) x^2 < x)

## reduce does aggregations
reduce(1:10, `+`)

list("exp" = exp,
     "times_two" = (function(x) x * 2),
     "square_root" = sqrt,
     "times_pi" = (function(x) x * pi)) %>%
  map(., ~ apply_it(1:10, .x) %>% reduce(`+`))
```

This map filter reduce paradigm is very powerful.

From SICP: Richard Waters (1979) developed a program that automatically analyzes traditional Fortran programs, viewing them in terms of maps, filters, and accumulations. He found that fully 90 percent of the code in the Fortran Scientific Subroutine Package fits neatly into this paradigm.

### 6.2 Higher Order Functions

A higher order function is a function that returns a function. A great example of one is a derivative.

Here are some other very useful higher order functions

```
## safely wraps a function to capture errors
try_it <- function(x) {
  if (x %% 2 == 0) stop()
  else x
}

1:10 %>% map(try_it)

stry <- safely(try_it)

1:10 %>% map(stry) %>% map(., ~ extract2(.x, "result"))
1:10 %>% map(stry) %>% map(., ~ extract2(.x, "error"))
```

Good uses for this kind of thing are web scraping, or just computations with many parts that might lead to failure

Here is another example of a higher order function, an **object**

```
higher_fn_person_obj <- function(first_name, last_name) {
  fname <- first_name
  lname <- last_name

  get_full_name <- function() {
    paste(fname, lname)
  }

  set_first_name <- function(new_fname) {
    fname <<- new_fname
  }

  set_last_name <- function(new_lname) {
    lname <<- new_lname
  }

  dispatch <- function(fn) {
    switch(fn,
           "get_full_name" = get_full_name,
           "set_first_name" = set_first_name,
           "set_last_name" = set_last_name,
           return("No dispatch found"))
  }
  dispatch
}

person_obj <- higher_fn_person_obj("Michael", "Rose")

person_obj("get_full_name")()
person_obj("set_first_name")("leahciM")
person_obj("set_last_name")("esoR")
person_obj("get_full_name")()
```

This example is a clearer reason as to why people call object oriented programming a message passing paradigm. They are passing messages to the stateful variables inside of an enclosed environment.

### 6.3 Declarative Programming

When we say code is declarative, we mean it focuses on what we are trying to do and **not how** we go about it.

SQL is a very declarative language:

```
SELECT *
FROM table
WHERE table.insurance_score > 600
LIMIT 10;
```


we don't necessarily care about how select, from, where, and limit were implemented as long as they do what we expect. We have abstracted that code from sight.

Here is an example of declarative functional composition:

```
## read in data
readRDS("data/ops_tbls_08sep20.rds") %>%
  ## join tables and make dataset target variable
  make_target_var(auto, "Umbrella") %>%
  (~ base_data) %>%
  ## data preprocessing
  get_recipe_prepped_data(recipe_list = preprocessors) %>%
  ## xgboost preprocessing
  get_xgb_matrices() %>>%
  ## store variable for use later
  (~ boost_mats) %>%
  ##  fit models
  get_xgb_models(scale_pos_weight_ratio = get_ideal_ratio_scale(base_data)) %>>%
  ## store variable for use later
```

```
  (~ xgb_mods) %>>%
  ## look at results
  get_xgb_results(prepped_data = boost_mats)
```

The flow here is:

**read in data %>% clean and set up target %>% preprocess for model %>% fit model %>% check results**

We can abstract this further into a function:

```
make_xgb_model <- function(data, target, preprocessors) {
  data %>%
    ## join tables and make dataset target variable
    make_target_var(auto, target) %>>%
    (~ base_data) %>%
    ## data preprocessing
    get_recipe_prepped_data(recipe_list = preprocessors) %>%
    ## xgboost preprocessing
    get_xgb_matrices() %>>%
    ## store variable for use later
    (~ boost_mats) %>%
    ##  fit models
    get_xgb_models(scale_pos_weight_ratio = get_ideal_ratio_scale(base_data)) %>>%
    ## store variable for use later
    (~ xgb_mods) %>>%
    ## look at results
    get_xgb_results(prepped_data = boost_mats)
}

## then we can map over a list of targets
readRDS("data/ops_tbls_08sep20.rds") %>%
  ## make a model for every unique endorsement in parallel
  future_map(names(extract2(., "endorsements")),
             ~ make_xgb_model(data = .,
                              target = .y,
                              preprocessors)) -> xgb_end_mods
```

Another example (article summarizer):

```
title_path <- "/html/body/div[1]/div[1]/main/div[1]/article/div[2]/div/header/div
```

```
body_path <- "/html/body/div[1]/div[1]/main/div[1]/article/div[2]/div/div/div[1]"

web_pages <- c("https://www.insurancejournal.com/news/international/2020/10/12/586
               "https://www.insurancejournal.com/news/national/2020/10/01/584756.h
               "https://www.insurancejournal.com/news/midwest/2020/09/21/583399.ht
               "https://www.insurancejournal.com/news/west/2020/09/16/582772.htm",
               "https://www.insurancejournal.com/news/national/2020/08/31/580793.h
               "https://www.insurancejournal.com/news/national/2020/08/12/578840.h
               "https://www.insurancejournal.com/news/national/2020/08/10/578554.h
               "https://www.insurancejournal.com/news/national/2020/08/07/578252.h
               "https://www.insurancejournal.com/news/midwest/2020/07/28/577147.ht
               "https://www.insurancejournal.com/news/national/2020/07/28/577121.h

web_pages %>%
  ## scrape each web page's title and body
  map(get_webpage_sentences, title_path, body_path) %>>%
  ## set results as scraped
  (~ scraped) %>%
  ## grab just the sentences
  map(., ~ extract2(.x, "sentences")) %>%
  ## run sentences through word vector embedding and pagerank
  map(all_together_tokens) %>%
  ## set names of list to titles
  set_names(scraped %>% map(~ extract2(.x, "title"))) -> sums_out_bert

## or
get_summary <- function(page, title_path, body_path) {
  page %>%
    get_webpage_sentences(title_path, body_path) %>>%
    (~ scraped) %>%
    map(., ~ extract2(.x, "sentences") %>%
             all_together_tokens()) %>%
    set_names(scraped %>% map(~ extract2(.x, "title")))
}

web_pages %>%
  map(., ~ get_summary(.x, title_path, body_path)) -> sums_out_bert
```

A good way to get into this habit is to have a separate file for functions, and then read those into your main program file. This is very common

in other languages like C.

# 7 Great Libraries for Functional Programming in R

purrr (map, reduce, filter) pipeR (explicit side effects) rlist (list manipulation functions) dplyr! (you've probably been doing FP all along) magrittr (pipes and pipe assignment)

The jelly to FP's peanut butter is **Metaprogramming**. This is a way in which we can program our programming language and treat code as data. In R a great library for metaprogramming is rlang. There are also **a lot** of base functions for metaprogramming.