Methods Camp 2025: Day 3

Christina Pao, Sofia Avila, Florencia Torche

September 6, 2025

Good morning!

Quick check-ins: Anything you're looking forward to for week 2 of school?

Outline for Today

1. Functions and libraries
2. Working with lists and apply functions
3. Sampling functions

Functions

- Yesterday, we reviewed how to use a for loop to avoid having to copy/paste code to repeat some process. Functions can provide a more efficient and flexible way to avoid this repetition
- E.g. suppose we had a vector and wanted to find all the outliers. Let's define outliers as values that are more than 2 standard deviations from the mean:

```
set.seed(1)
vec_1 = rnorm(100, mean = 50, sd = 2)
max_val = mean(vec_1) + 2 * sd(vec_1)
min_val = mean(vec_1) - 2 * sd(vec_1)
vec_1[vec_1 > max_val | vec_1 < min_val]
```

```
[1] 45.57060 46.02130 53.96080 54.80324 46.39008 54.34522
```

Functions

- But now imagine you had to do this for multiple vectors. You could just copy/paste the code above:

```
set.seed(1)
vec_2 = rnorm(100, mean = 60, sd = 10)
max_val = mean(vec_2) + 2 * sd(vec_2)
min_val = mean(vec_2) - 2 * sd(vec_2)
vec_2[vec_2 > max_val | vec_2 < min_val]
```

```
[1] 37.85300 40.10648 79.80400 84.01618 41.95041 81.72612
```

```
vec_3 = rnorm(100, mean = 60, sd = 10)
max_val = mean(vec_3) + 2 * sd(vec_3)
min_val = mean(vec_3) - 2 * sd(vec_3)
vec_3[vec_3 > max_val | vec_3 < min_val]
```

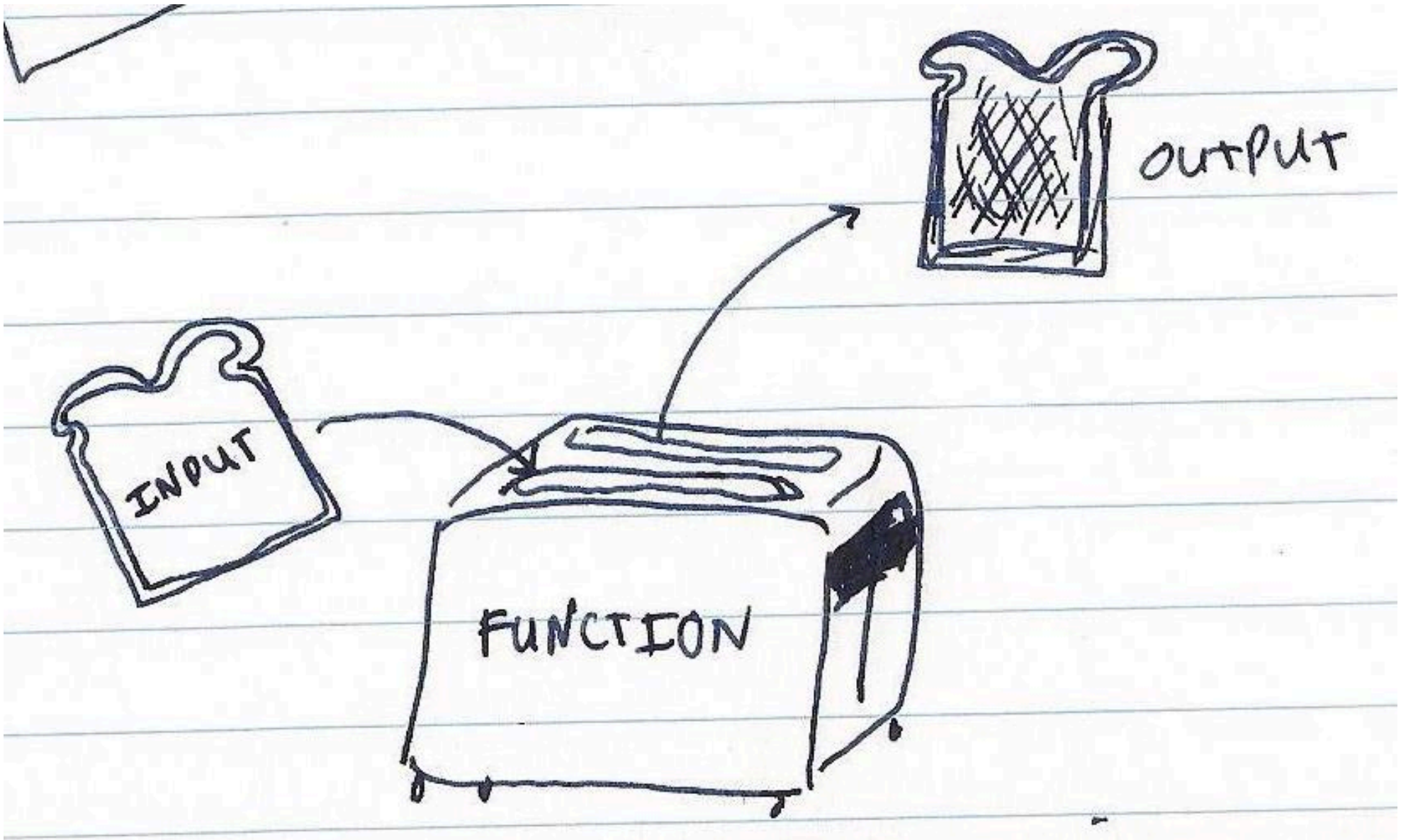
```
[1] 80.87167 82.06102 83.07978 80.75245
```

- But this is inefficient, error-prone, and less readable. Instead, we could define a function to do this for us!

Functions Overview

- Functions are reusable blocks of code that perform a specific task.
- They take inputs (called **arguments**) and return outputs (called **return values**).
- You've already seen this logic in math classes. Functions in programming are like functions in math.
  - E.g. the function $f(x) = x^2$ takes an input x and returns the output x^2.
- My first CS prof in college described functions as toasters: you put in bread, and you get out toast.
  - You can use the toaster again and again with different slices of bread.

Functions Overview



How to approach writing a function

- Ask yourself: **“what problem am I trying to use this function to solve?”**
  - I usually realize I need a function when I find myself copying and pasting code to do the same thing multiple times…
- Once you’ve identified the problem, try writing code outside of the function to deal with a few simple cases of the problem
- Then, see what you can do to generalize the code from step two so that it can handle a variety of versions of the problem

Simple Case: Removing Outliers

We already came up with some code to find outliers from our initial vector:

```
set.seed(1)
vec_1 = rnorm(100, mean = 50, sd = 2)
max_val = mean(vec_1) + 2 * sd(vec_1)
min_val = mean(vec_1) - 2 * sd(vec_1)
vec_1[vec_1 > max_val | vec_1 < min_val]
```

```
[1] 45.57060 46.02130 53.96080 54.80324 46.39008 54.34522
```

- Now, let's try to generalize this code. What should the inputs of our function be?

Function Syntax

Functions usually (but not always)…

1. Take in one or more arguments: the bread
2. Then, they execute some code: e.g. heating up the bread
3. And then return a value: the toast that pops up

```
function_name = function(argument1, argument2...){
  code to execute
  return(value_to_return)
}
```

Our Function

Let's put our code into a function that:

- ☑ takes in a vector
- ☑ a standard deviation cut-off
- ☑ and returns the outliers:

```
find_outliers = function(vec, sd_cutoff){
  max_val = mean(vec) + sd_cutoff * sd(vec)
  min_val = mean(vec) - sd_cutoff * sd(vec)
  return(vec[vec > max_val | vec < min_val])
}

find_outliers(vec_1, 2)
```

```
[1] 45.57060 46.02130 53.96080 54.80324 46.39008 54.34522
```

Our Function: Default Arguments

- We can also set default values for arguments.
  - This is useful if we want to use the function with a common value for an argument, but still allow users to change it if they want.
- Also prevents errors if the user forgets to specify an argument.

```
find_outliers = function(vec, sd_cutoff = 2){
  max_val = mean(vec) + sd_cutoff * sd(vec)
  min_val = mean(vec) - sd_cutoff * sd(vec)
  return(vec[vec > max_val | vec < min_val])
}

find_outliers(vec_1)  # uses default sd_cutoff of 2
```

```
[1] 45.57060 46.02130 53.96080 54.80324 46.39008 54.34522
```

```
find_outliers(vec_1, 2.5)  # uses sd_cutoff of 3
```

```
[1] 45.57060 54.80324
```

Our Function: Making it more robust

- We can also add checks to make sure the inputs are valid –> helps prevent errors given unexpected inputs!
- Usually not something to spend a lot of time on if you're just writing a function for yourself, but good practice if you're writing a function for others to use.
- How might we make our function more robust?

```r
find_outliers = function(vec, sd_cutoff = 2){
  if(!is.numeric(vec)){
    stop("vector must be numeric!")
  }
  if(!is.numeric(sd_cutoff) || length(sd_cutoff) != 1){
    stop("standard deviation cut-off must be a single numeric value!")
  }
  max_val = mean(vec) + sd_cutoff * sd(vec)
  min_val = mean(vec) - sd_cutoff * sd(vec)
  return(vec[vec > max_val | vec < min_val])
}

string_vec = c("a", "b", "c")
find_outliers(string_vec)
```

```
Error in find_outliers(string_vec): vector must be numeric!
```

```r
find_outliers(vec_1, c(2, 3))
```

```
Error in find_outliers(vec_1, c(2, 3)): standard deviation cut-off must be a single numeric value!
```

Your Turn

We've been hired by Tacoria to help streamline their ordering process. Customers can choose a bowl or burrito. By default, the base of the bowl/burrito is chicken but, if customers are vegan or vegetarian, they swap this with beans. They also add cheese to all orders unless the customer is vegan.

Write a function, `make_tacoria_order()` that takes in two variables:

- bowl_burrito: a string variable (`bowl` or `burrito`)
- dietary_restriction: a string variable (`vegan`, `vegetarian`, `none`)

Use information from these variables to return a string with the customer's food order. The order should be in the format: `A [bowl|burrito] with [chicken|beans]. [Add cheese|No cheese].`

Bonus: can you add default values? How can you make your function more robust? Note: don't do any string processing for now.

Your turn

```
1 make_tacoria_order('bowl','none')
```

```
[1] "A bowl with chicken. Add cheese."
```

```
1 make_tacoria_order('bowl','vegetarian')
```

```
[1] "A bowl with beans. Add cheese."
```

```
1 make_tacoria_order('bowl','vegan')
```

```
[1] "A bowl with beans. No cheese."
```

```
1 make_tacoria_order('burrito','none')
```

```
[1] "A burrito with chicken. Add cheese."
```

```
1 make_tacoria_order('burrito','vegetarian')
```

```
[1] "A burrito with beans. Add cheese."
```

```
1 make_tacoria_order('burrito','vegan')
```

```
[1] "A burrito with beans. No cheese."
```

Multiple Return Objects

- In the functions we've seen so far, we return a single object (e.g. a vector of outliers, a single string, a data frame, etc.).
- However, sometimes we might want to return multiple objects from a function.
- We can do this by returning a list!
- The syntax for a list:

```
my_list = list(object1_name = object1, object2_name = object2, ...)
```

Multiple Return Objects

Let's modify our `find_outliers` function to return both the outliers and the mean and standard deviation of the vector:

```
find_outliers = function(vec, sd_cutoff = 2){
  mean_vec = mean(vec)
  sd_vec = sd(vec)
  max_val = mean_vec + sd_cutoff * sd_vec
  min_val = mean_vec - sd_cutoff * sd_vec
  outliers = vec[vec > max_val | vec < min_val]

  return(list('outliers' = outliers,
              'vector_mean' = mean_vec,
              'vector_sd' = sd_vec))
}
```

Multiple Return Objects

```
find_outliers = function(vec, sd_cutoff = 2){
  mean_vec = mean(vec)
  sd_vec = sd(vec)
  max_val = mean_vec + sd_cutoff * sd_vec
  min_val = mean_vec - sd_cutoff * sd_vec
  outliers = vec[vec > max_val | vec < min_val]

  return(list('outliers' = outliers,
              'vector_mean' = mean_vec,
              'vector_sd' = sd_vec))
}
```

Now, we can call the function and access the individual objects in the list using the $ operator:

```
result = find_outliers(vec_1, 2)
result$outliers
```

```
[1] 45.57060 46.02130 53.96080 54.80324 46.39008 54.34522
```

```
result$vector_mean
```

```
[1] 50.21777
```

Your Turn

Let's continue our unpaid Tacoria technical consultantship. Now we will write a function that interacts with the customer to take in their food preferences and dietary restrictions. Write a function that asks users whether they want a bowl or burrito and whether they're vegan or vegetarian. Return their responses as a list.

Hint 1: The function `npr = readline(prompt = 'On a scale of 1–10, how likely are you to recommend methods camp to a friend?')` takes in user input from the console and saves it into the variable `npr`.

Hint 2: This function will interact with the function you wrote above. How can you process the strings so they match what `make_tacoria_order()` expects? You could even use a `while` loop…

Your Turn

Libraries and Functions

- A lot of what we'd been doing the past two days has been using functions from libraries.
  - We were using these every time we had the syntax `()`.
- R has a lot of built-in functions (e.g. `mean()`) but we also imported **libraries**
- I think of libraries as basically a collection of functions that others have written to make our lives easier.
- If a function is built-in, then we can just automatically call it as soon as we open R:

```
1 mean(c(1, 2, 3, 4, 5))
```

```
[1] 3
```

Libraries and Functions

- When a function is **not** built-in, R doesn't *"know"* about it until we load the library that contains it.
  - This might require installing the library first.
- For example, the `cowsay` library is not built-in, so if I try the following, R will throw an error:

```
1 say("Functions help us cowculate almost anything!", by = "cow")
```

```
Error in say("Functions help us cowculate almost anything!", by = "cow"): could not find function "say"
```

Libraries and Functions

- To use the `cowsay` library, we first need to install it (if we haven't already).
- Then, we can either load it with `library(cowsay)` or use the `cowsay::` syntax to call the function directly without loading the library.
  - What is the `::` operator, you ask? What's the difference between these strategies, you ask?
  - Let's talk about that!

Detour: the `::` operator

- We can imagine that, every time we install a library, it's like we're giving the R program in our computer a new library card.
- In this analogy, functions are like books in the library.
- When I load libraries at the beginning of my code, I'm telling R where to find the books I want to read.
  - Without this machinery, R would be **a lot slower** since it would have to search through all the libraries every time we call a function.
- The `::` operator allows us to call a function from a specific library without loading the entire library into our R session.
  - It is just a more specific way of telling R where to find the function we want to use.

Detour: the `::` operator

- Using the `::` is particularly useful to avoid conflicts with other libraries that might have functions with the same name. E.g., the following code works:

```
1 mtcars %>%
2   select(mpg) %>%
3   head(1)
```

```
           mpg
Mazda RX4   21
```

- But the `MASS` library also has a `select()` function, which conflicts with `dplyr`'s:

```
1 library(MASS)
2 mtcars %>%
3   select(mpg)
```

```
Error in select(., mpg): unused argument (mpg)
```

So we can use `dplyr::select()` to avoid this conflict:

```
1 library(MASS)
2 mtcars %>%
3   dplyr::select(mpg) %>%
4   head(1)
```

```
           mpg
Mazda RX4   21
```

Detour: `::` operator summary

- The `::` operator allows us to call a function from a specific library without loading the entire library.
- It is useful to avoid conflicts with other libraries that might have functions with the same name.
- Other use cases:
  - If you're only using one or two functions from a library, it can be more efficient to use `::` instead of loading the entire library –> keeps your R session cleaner and faster.
  - It also makes your code more readable, as it clearly indicates which library the function comes from.

Libraries and Functions

So, applying what we learned, we can use:

```
1 library(cowsay)
2 say("Functions help us cowculate almost anything!", by = "cow")
```

```
 _____________________________________________
< Functions help us cowculate almost anything! >
 ---------------------------------------------
      \
       \

        ^__^
        (oo)\ ________
        (__)\         )\ /\
             ||------w|
             ||      ||
```

Libraries and Functions

We can peer under the hood of the functions we import. This has been really useful for me when, for example, I'm using a function from a library, and it keeps crashing (and the error message is not helpful).

Let's look at the `say` function from the `cowsay` library: . . .

```
1 cowsay::say
```

```
function (what = "Hello world!", by = "cow", type = NULL, what_color = NULL,
    by_color = what_color, length = 18, fortune = NULL, width = 60,
    ...)
{
    stopifnot(`what must be length 1` = has_length(what, 1))
    if (crayon::has_color() == FALSE && (!is_null(what_color) ||
        !is_null(by_color))) {
        inform(c("Colors cannot be applied in this environment :(",
            "Try using a terminal or RStudio/Positron/etc."))
        what_color <- NULL
        by_color <- NULL
    }
    else {
        what_color <- check_color(what_color)
        by_color <- check_color(by_color)
    }
    if (is_null(type)) {
        if (interactive()) {
            type <- "message"
        }
        else {
            type <- "print"
        }
```

Your Turn

Take five minutes to explore the library `snakecase`, which has functions for converting strings between different naming conventions (e.g., snake_case, camelCase, etc.). Do this in three ways:

1. Visit the GitHub page for the package. Their GitHub follows a similar structure to a lot of other packages, so it's a good format to get familiar with. Skim the README file. Then, go into the R folder and look at the code for the different function.
2. In R, use the `help()` or `?` function to look up the documentation for the `to_any_case()` function. This is a good way to get a quick overview of what a function does and how to use it.
3. Now, run `snakecase::to_any_case` to see the code for the function. This is a good way to see how the function is implemented and what it does under the hood.

Beyond `for` Loops?

- `for` loops are fine! But they can be:
  - Verbose
  - Hard to debug or nest
  - Not vectorized (often slower since vectorized functions apply to a whole vector in one go)
- Alternatives offer:
  - Cleaner syntax
  - Safer output types
  - Built-in parallelism (in some cases!)

Option 1: `purrr::map()` Family

Part of the **tidyverse**, `purrr::map()` is like a loop that:

- Takes a vector/list
- Applies a function to each element
- Returns a result (e.g., list, numeric, data frame)

Example: Using `map()`

Let's say we have a vector of names and we want to convert them to uppercase. Using `for` loops, we might do something like this:

```
names <- c("Taylor", "Travis", "Beyonce")
# Base R for loop
output <- vector("list", length(names))
for (i in seq_along(names)) {
  output[[i]] <- toupper(names[i])
}
```

- But with `purrr::map()`, we can do this in a single line:

```
# With purrr
map(names, toupper)
```

```
[[1]]
[1] "TAYLOR"

[[2]]
[1] "TRAVIS"

[[3]]
[1] "BEYONCE"
```

`map()`: Specifying Output Type

By default, `map()` returns a list. But you can specify the output type using `map_*()` functions:

```r
x <- list(1, 2, 3)

map_dbl(x, function(n) n^2)
```

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

```r
map_chr(x, as.character)
```

```
[1] "1" "2" "3"
```

```r
# → ["1", "2", "3"]
```

Benefits:

- Fewer surprises!
- Easier to debug and document
- Pairs well with pipes

`map_dfr()`: Combining Data Frame Rows

- One of the most useful `map_*` functions is `map_dfr()`, which combines results into a single data frame.
- Recall when we were using `for` loops to open `csv` files, saving each one as a data frame, and then combining them into one big data frame?
- `map_dfr()` does this for us automatically!

```r
files = list.files(path = "data/ces_by_state", pattern = "*.csv", full.names = TRUE)
ces_data <- map_dfr(files, read.csv)
head(ces_data,3)[1:6]
```

```
  year state         county state_fips county_fips zipcode
1 2024    CA Alameda County          6           1   94541
2 2023    CA Alameda County          6           1   94587
3 2024    CA Alameda County          6           1   94702
```

`map_dfc()`: Combining Data Frame Columns

- Similar to `map_dfr()`, but combines results into a data frame with columns instead of rows.
- I don't use this one as often, but it's handy to know.
- E.g. suppose we wanted to calculate average support for repealing the affordable care act (`repeal_affordable_care_act`) for each ideology (`ideo`) and save each result as a column in a new data frame:

```
get_support = function(curr_ideo){
  repeal_aca_support = ces_data %>%
    filter(ideo == curr_ideo) %>%
    summarise(mean_support = mean(repeal_affordable_care_act, na.rm = TRUE)) %>%
    # rename the column to the current ideology
    rename(!!curr_ideo := mean_support)
  return(repeal_aca_support)
}

ideo_levels <- unique(ces_data$ideo)
support_df <- map_dfc(ideo_levels, get_support)
support_df
```

```
  very_liberal  moderate   liberal very_conservative conservative
1   0.08928571 0.3203125 0.1325301           0.71875    0.6666667
```

Option 2: `foreach`

- The `foreach` package uses syntax that should feel quite similar to the `for` loop, but it has some powerful features.
- The basic syntax is as follows:

```
library(foreach)

foreach(i = 1:3) %do% {
  return(i^2)
}
```

```
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 9
```

`foreach()` Combining Results

- One of the useful features of `foreach` is that it can combine results from each iteration.
- You can specify which function to use for combining results with the `.combine` argument.
- E.g. if we want a simple vector, we can specify `.combine = c`:

```
1 i_sq = foreach(i = 1:3, .combine = c) %do% {
2   i^2
3 }
4 i_sq
```

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

`foreach()` Combining Results

- If we want to combine results into a data frame, we can use `.combine = rbind` or `.combine = cbind`.
- Let's rewrite the code we used above with `map_dfr` but using `foreach`:

```
files = list.files(path = "data/ces_by_state", pattern = "*.csv", full.names = TRUE)
ces_data <- foreach(file = files, .combine = rbind) %do% {
  read.csv(file)
}
head(ces_data,3)[1:6]
```

```
  year state         county state_fips county_fips zipcode
1 2024    CA Alameda County          6           1   94541
2 2023    CA Alameda County          6           1   94587
3 2024    CA Alameda County          6           1   94702
```

`foreach()` Parallel Processing

- One of the most powerful features of `foreach` is that it can be easily modified to run in parallel using the `%dopar%` operator.
- What is parallel processing?
  - Running multiple tasks at the same time, which can significantly speed up computations, especially for large datasets or complex calculations.
  - Imagine you were working on a big project that involved opening ten different datasets and doing some complicated calculations on each one.
  - One way you could speed that up is to work with nine coauthors or RAs, have each of them process one dataset on their computer, and then combine the results at the end.
  - We can achieve a similar effect with parallel processing–almost like splitting your computer into multiple computers that can work on different tasks at the same time.

`foreach()` Parallel Processing

- We will practice using `foreach` with parallel processing to run a bootstrap simulation to estimate standard errors.
- The bootstrap works by repeatedly resampling the data with replacement and calculating the statistic of interest (e.g., mean, median, etc.) for each resample.
- In the code below, we conduct 1000 bootstrap iterations. We see that it takes about 2.6 seconds to run:

```
set.seed(1)
# make random data
n = rnorm(100000, mean = 50, sd = 10)
# number of bootstrap iterations
n_iter = 1000
system.time({
  boot_means = foreach(i = 1:n_iter, .combine = c) %do% {
    # sample with replacement
    curr_sample = sample(n, length(n), replace = TRUE)
    # calculate the mean of the current sample
    boot_mean = mean(curr_sample)
    return(boot_mean)
  }
})
```

```
   user  system elapsed 
  2.864   0.075   2.958 
```

`foreach()` Parallel Processing

- Now, let's run the same code but using parallel processing with `%dopar%`.
- To do this, we need to:
  - Determine how many cores our computer has (like how many mini-computers that can run their own tasks.
  - Register a parallel backend. The `doParallel` package is a common choice for this.

`foreach()` Parallel Processing

Code below tells us how many cores our computer has available for parallel processing:

```
parallel::detectCores()
```

```
[1] 12
```

Next, we need to register a parallel backend. This allows R to know how many cores to use for parallel processing.

```
library(doParallel)
registerDoParallel(cores = 4) # choose number less number of cores on your computer
```

- Lastly, we run our code using `%dopar%` instead of `%do%`. We see the process only took 0.81 seconds (~69% reduction)

```
set.seed(1)
system.time({
  boot_means = foreach(i = 1:n_iter, .combine = c) %dopar% {
    curr_sample = sample(n, length(n), replace = TRUE)
    boot_mean = mean(curr_sample)
    return(boot_mean)
  }
})
```

```
   user  system elapsed 
  2.778   0.173   0.811 
```

Summary: why use `foreach`?

- Feels like a `for` loop but is **more flexible**
- Can run **in parallel** with `%dopar%`
- Builds results more cleanly and efficiently

Your Turn

The last task our Tacoria bosses are asking us to do is to help them simulate earnings from a typical lunch period. Specifically, the restaurant wants to simulate customer orders for **100 lunch periods**. Here are there stats:

- Each lunch period (11am-2pm), Tacoria serves between 100 and 150 customers
- From their data, it seems around 10% of their customers are vegetarian, 5% are vegan, and the rest have no dietary restrictions.
- Bowls are about twice as popular as burritos.
- Their bean burrito is $8 and their chicken one is $11. Bowls cost $2 more and cheese is free.

Your Turn

We will simulate 100 lunch periods. For each period, we will calculate the total amount of money earned. Make sure to save the data into a vector, `tacoria_earnings`. Simulate the data three ways:

1. Using a **for loop**
2. Using **foreach**
3. Using **map** from `purrr` in combination with **replicate**

Hint: instead of writing the simulation 3 times from scratch, are there parts you can make into a function?

Helper Function

```
get_order = function(){
  dietary_restriction = sample(c('vegan','vegetarian','none'),
                               size=1,
                               prob=c(0.05,0.1,0.85))
  bowl_burrito = sample(c('bowl','burrito'),size=1,prob=c(2/3,1/3))
  cost = ifelse(dietary_restriction == 'none',11,8)
  if(bowl_burrito == 'bowl') cost = cost + 2
  return(cost)
}
```

For Loops

```
n_lunches = 100
tacoria_earnings = numeric(n_lunches)
for(i in 1:n_lunches){
  n_customers = sample(x=200:250,size=1)
  earnings = 0
  for(j in 1:n_customers){
    earnings = earnings + get_order()
  }
  tacoria_earnings[i] = earnings
}
mean(tacoria_earnings)
```

```
[1] 2683.64
```

For Each

```
tacoria_earnings = foreach(i = seq(n_lunches), .combine=c) %do% {
  n_customers = sample(x=200:250,size=1)
  curr_earnings = foreach(i = seq(n_customers), .combine=sum) %do% {
    get_order()
  }
}
mean(tacoria_earnings)
```

```
[1] 2675
```

Map

```
tacoria_earnings = map_dbl(1:n_lunches, function(i){
  n_customers = sample(x=200:250,size=1)
  curr_earnings = replicate(n_customers,get_order())
  return(sum(curr_earnings))
})

mean(tacoria_earnings)
```

```
[1] 2660.38
```

Discussion

- Did all three methods give similar result?
- Which approach feels easiest to read and write?
- Are there other methods you like to use?

Questions?

```
cowsay::say("Functions and foreach loops are udderly amazing!", by = "cow")
```

```
 ________________________________________________
< Functions and foreach loops are udderly amazing! >
 ------------------------------------------------
      \
       \

        ^__^
        (oo)\ ________
        (__)\         )\ /\
             ||------w|
             ||      ||
```

```
cowsay::say("I would have to agree... they're quite eggcellent...", by = "chicken")
```

```
 ____________________________________________________
< I would have to agree... they're quite eggcellent... >
 ----------------------------------------------------
      \
       \

         _
       _/ }
      `>' \
      `|   \
       |   /'-.     .-.
        \'     ';`--' .'
         \'.    `'-./
          '.`-..-;`
            `;-..'
            _| _|
            /` /` [nosig]
```