Automating tasks in R to streamline your workflow

R-Ladies Vancouver workshop

Kim Dill-McFarland, kadm@uw.edu

version December 09, 2020

Contents

Introduction

Do you look at your script and see the same code repeated but with one word changed? Do you wish your script could run thousands of tasks while you take a break? Or does your script already run all those tasks but take forever? In this workshop, we tackle some automation functions (`for`, `apply`, `mutate`) that will help you to streamline code and time to accomplish repetitive tasks in R.

Dr. Kim Dill-McFarland has been teaching data science since her graduate years, including as a teaching fellow at the U. of British Columbia and as a Carpentries certified instructor. She absolutely loves R, using it every day at work as a bioinformatician at the U. of Washington and at home modeling her new kitten's weight gain. Kim strongly believes in open-source tools and the power coding can bring to scientists of all levels and disciplines.

Workshop details Dec 10th, 2020 at 6pm
https://www.meetup.com/rladies-vancouver/events/274737624/
Notes at https://github.com/hawn-lab/workshops_UW_Seattle/tree/master/2020.12.10_automation

Prior to the workshop

- Install R https://mirror.its.sfu.ca/mirror/CRAN/
- Install RStudio (choose the Desktop version) https://www.rstudio.com/products/rstudio/download/#download
- Install tidyverse package with `install.packages("tidyverse")`
- Install penguin data package with `install.packages("palmerpenguins")`
- Install penguin data package with `install.packages("doParallel","foreach")`

Setup

Open RStudio and create a new RScript (File > New File > R Script or Cmd/Crtl+Shift+N). Save the script on your computer. As we work during the workshop, save your code in this script and feel free to add notes by using `#`. See examples below.

Load packages.

```
# Notes can be added like this
library(tidyverse) # or like this!
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(palmerpenguins)
library(doParallel)
```

```
## Loading required package: foreach
```

```
## 
## Attaching package: 'foreach'
```

```
## The following objects are masked from 'package:purrr':
## 
##     accumulate, when
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
library(foreach)
```

Get data. More information on these data can be found at https://github.com/allisonhorst/palmerpenguins

```
# Save data to object in environment
dat <- penguins

# View data
dat
```

```
## # A tibble: 344 x 8
##    species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
##    <fct>   <fct>           <dbl>         <dbl>            <int>       <int>
##  1 Adelie  Torge~           39.1          18.7              181        3750
##  2 Adelie  Torge~           39.5          17.4              186        3800
##  3 Adelie  Torge~           40.3          18                195        3250
##  4 Adelie  Torge~           NA            NA                 NA          NA
##  5 Adelie  Torge~           36.7          19.3              193        3450
##  6 Adelie  Torge~           39.3          20.6              190        3650
##  7 Adelie  Torge~           38.9          17.8              181        3625
##  8 Adelie  Torge~           39.2          19.6              195        4675
##  9 Adelie  Torge~           34.1          18.1              193        3475
## 10 Adelie  Torge~           42            20.2              190        4250
## # ... with 334 more rows, and 2 more variables: sex <fct>, year <int>
```

Here, we see that the data contain information on body size and sex of several penguin species. Our goal will be to change the bill mm measurements to cm and create plots of size by sex for each species.

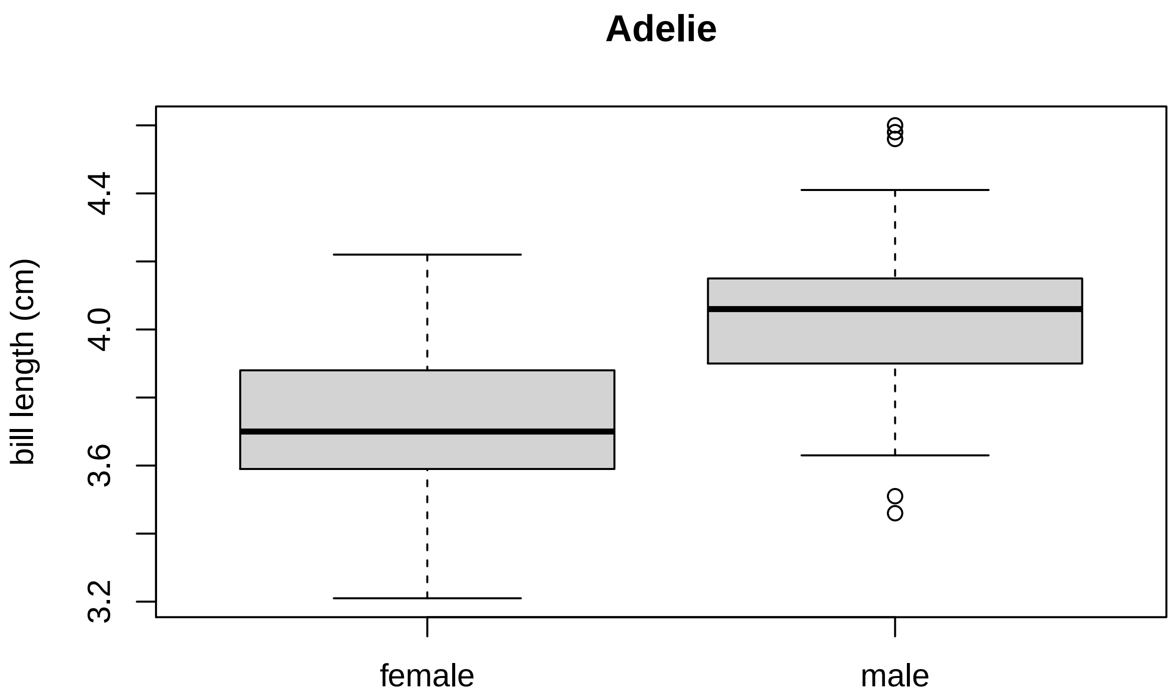
Base R

Before we delve into how to achieve our goal most efficiently, let's see how you would do this in base R.

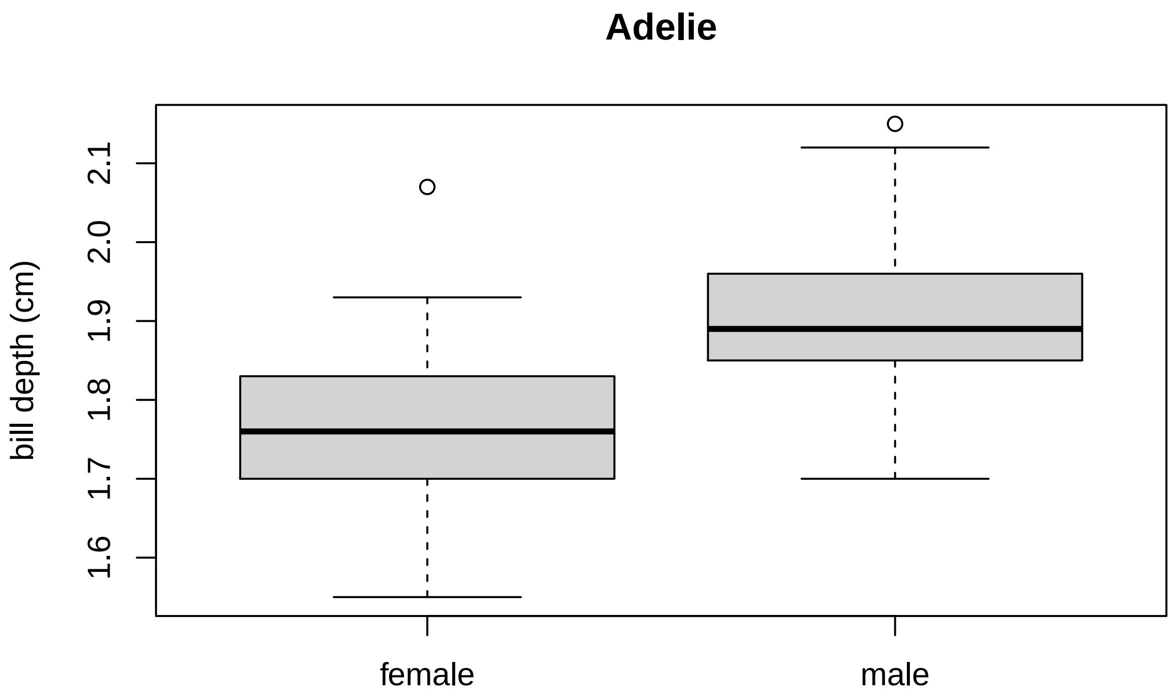
```
# Step 1: Filter data to the first penguin species
## Remember that R calls data frames with [rows, columns]
dat.filter <- dat[dat$species == "Adelie", ]

# Step 2: Convert the mm measurements to cm
bill_length_cm <- dat.filter$bill_length_mm/10
bill_depth_cm <- dat.filter$bill_depth_mm/10

# Step 3: Plot measurements separated by sex
plot(x = dat.filter$sex, y = bill_length_cm,
     ylab = "bill length (cm)", xlab = "", main = "Adelie")
```



```
plot(x = dat.filter$sex, y = bill_depth_cm,
     ylab = "bill depth (cm)", xlab = "", main = "Adelie")
```



At this stage, we already see some repeated code and to complete our task for all 3 penguin species, we'd need to copy the above, change the species, and re-run 2 more times! Instead of doing that, we're going to create a loop to run the above code for each species automatically.

for loops

Syntax

The basic syntax of a loop in R is

```
for(something in something.else){
    DO THINGS
}
```

For example, we can print the 3 penguin species to the console.

```
# list the species by-hand
for(i in c("Adelie", "Chinstrap", "Gentoo")){
  print(i)
}
```

```
## [1] "Adelie"
## [1] "Chinstrap"
## [1] "Gentoo"
```

```
# Or have R figure out the species for you!
for(i in unique(dat$species)){
  print(i)
}
```

```
## [1] "Adelie"
## [1] "Gentoo"
## [1] "Chinstrap"
```

In the above, we use `i` to represent the thing we're looping through. This is common practice as `i` stands for iteration, but you can use anything! Just be sure that whatever you use in the `for( )` statement is what you use within the loop `{ }`. For example,

```
for(magic in unique(dat$species)){
  print(magic)
}
```

```
## [1] "Adelie"
## [1] "Gentoo"
## [1] "Chinstrap"
```

Using a loop

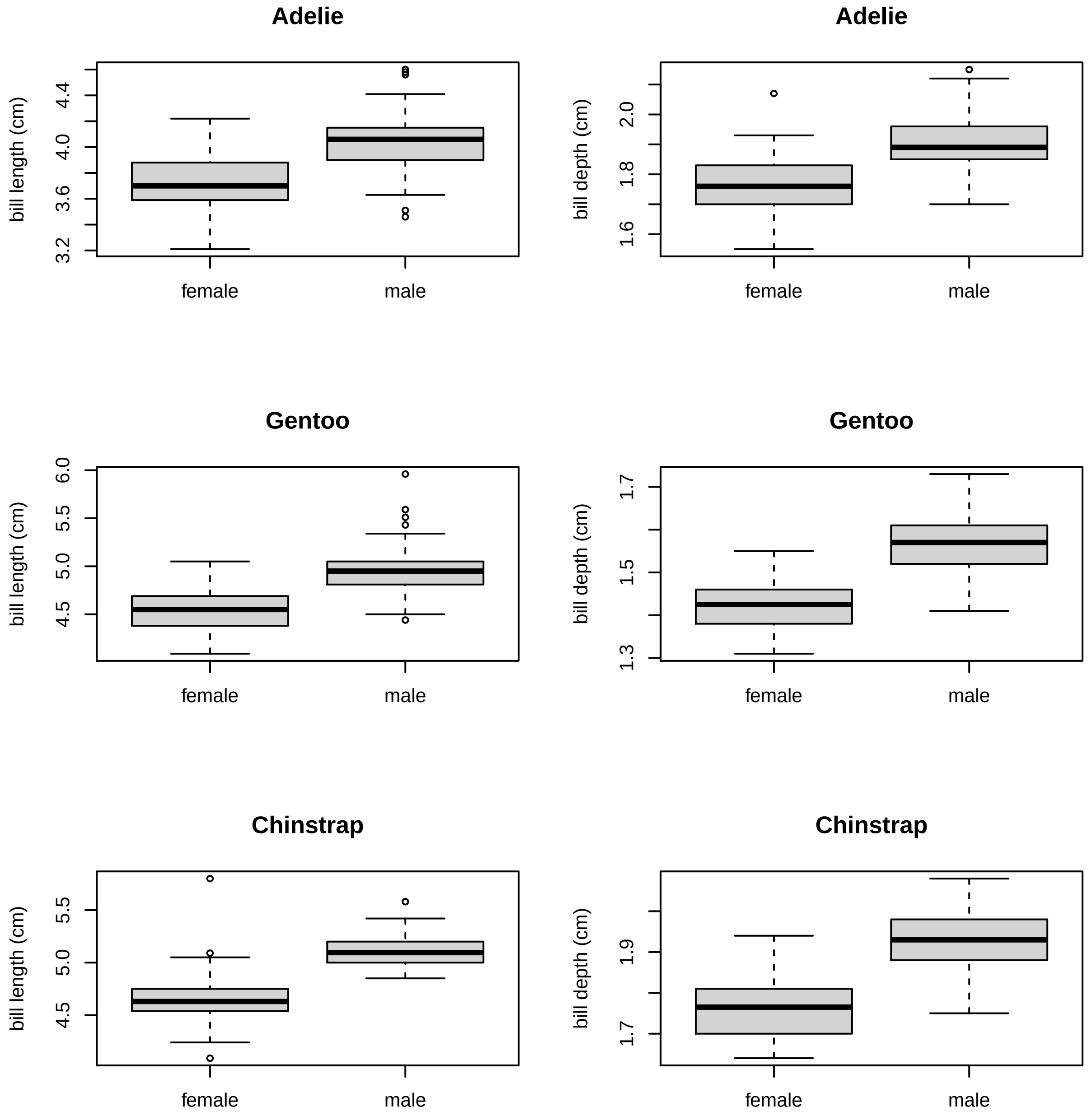
Now, instead of just printing the species name, let's copy-paste all our base R code into the loop. Anywhere we previously put "Adelie", we now replace with `i` to tell the loop to use each unique species.

To make this easier to see in the Rmarkdown, I'm also going to print the plots into a grid using `par(mfrow=c(rows,columns))`.

```
#Make 3 by 2 grid for 6 plots
par(mfrow=c(3,2))

for(i in unique(dat$species)){
  # Step 1: Filter data to the penguin species
  dat.filter <- dat[dat$species == i, ]

  # Step 2: Convert the mm measurements to cm
  bill_length_cm <- dat.filter$bill_length_mm/10
  bill_depth_cm <- dat.filter$bill_depth_mm/10
  # Step 3: Plot measurements separated by sex
  plot(x = dat.filter$sex, y = bill_length_cm,
      ylab = "bill length (cm)", xlab = "", main = i)
  plot(x = dat.filter$sex, y = bill_depth_cm,
      ylab = "bill depth (cm)", xlab = "", main = i)
}
```



Now, the loop doesn't run any faster than if we'd copy-pasted and input each species by-hand. However, it has saved us 10 lines of code and with this framework, we could scale up to 100s of species without changing anything except the size of the grid made with `par( )`.

Next, we'll explore methods for improving the code inside the loop.

apply functions

Syntax

R's `apply` functions allow you to modify multiple pieces of data at once. They work on most standard data types with some specialized versions such as `lapply` for lists. The standard version, `apply`, uses the function you give it on all rows (`MARGIN = 1`) or all columns (`MARGIN = 2`). These functions are *extremely* fast and efficient.

Using `apply`

In our loop, we can use `apply` to change multiple mm columns to cm. Since we're working on columns, we use `apply` with `MARGIN = 2`. Note that we are working with `dat.filter` because that is what will be used within the loop at this step.

```
dat.filter.cm <- apply(dat.filter, MARGIN = 2, function(x) x/10)
```

```
## Error in x/10: non-numeric argument to binary operator
```

We get an error here because `apply` is tying to divide all columns by 10 and we have non-numeric data like 'species'. So, we must specify the columns we want to modify. If you wanted to transform all columns, you wouldn't need to modify the above.

```
# Using index numbers
## shorter code but more likely to result in errors if you're not careful
dat.filter.cm <- apply(dat.filter[c(3,4)],
                       MARGIN = 2, function(x) x/10)

# Using column names
## longer code but more exact and no errors if your column order changes
dat.filter.cm <- apply(dat.filter[c("bill_length_mm","bill_depth_mm")],
                       MARGIN = 2, function(x) x/10)
```

This takes two short lines of code for step 2 and replaces with 1 longer line. This doesn't seem like much now but remember, we're only working with two columns at the moment. If we scaled up in the old loop, it would mean copy-pasting the mm to cm conversion for every new variable. In the `apply` loop, we'd just need to add the new variable names to the vector. Moreover, `apply` is much faster and for larger data sets, there would be a noticeable improvement in speed.

The tidyverse

Syntax

The tidyverse (https://www.tidyverse.org/) is a suite of packages for data manipulation and plotting. Its function `mutate` can be used to accomplish many of the same things as `apply`. This is not quite as fast as `apply`, but as you'll see below, the tidyverse's pipe function allows us to further clean-up our code.

Though we will not use all of these in this workshop, here is a list of some commonly used tidyverse functions.

- `select` a subset of variables (columns)
- `filter` out a subset of observations (rows)
- `rename` variables
- `arrange` the observations by sorting a variable in ascending or descending order
- `mutate` all values of a variable (apply a transformation)
- `group_by` a variable and `summarise` data by the grouped variable
- `*_join` two data frames into a single data frame

`mutate` multiple variables

As the name implies, `mutate` changes variables. You can modify one at a time like so, and we see the new column at the end of the data frame.

```
#Convert bill length to cm
dat.mutate <- mutate(dat.filter, bill_length_cm = bill_length_mm/10)

#list columns mutated data
colnames(dat.mutate)
```

```
## [1] "species"           "island"            "bill_length_mm"
## [4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
## [7] "sex"               "year"              "bill_length_cm"
```

In our case, we want to mutate multiple columns. Similar to our base R method, we could do this with multiple `mutate` functions. However, we can also do this with a single line of code by using the `across( )` modifier which tells `mutate` to change multiple columns at once (similar to `apply`).

Not that the ~ denotes a function with `.x` being a placeholder for the variables being altered by that function.

```
#Convert bill length AND depth to cm
dat.mutate <- mutate(dat.filter, across(c(bill_length_mm, bill_depth_mm),
                                        .fns = ~.x/10))

#list columns in mutated data
colnames(dat.mutate)
```

```
## [1] "species"           "island"            "bill_length_mm"
## [4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
## [7] "sex"               "year"
```

Oh no! We see the same column names but in fact, the data have changed.

```
dat.filter$bill_length_mm[1:5]
```

```
## [1] 46.5 50.0 51.3 45.4 52.7
```

```
dat.mutate$bill_length_mm[1:5]
```

```
## [1] 4.65 5.00 5.13 4.54 5.27
```

This is because mutate does not automatically rename the columns. The `across( )` modifier is relatively new in the tidyverse and still being developed. Currently, you can add a prefix or suffix to the new column names (called `{.col}`) but cannot modify the original name. Here, let's add "_cm" to the names so we can see the new data.

```
#Convert bill length AND depth to cm. Rename columns
dat.mutate <- mutate(dat.filter, across(c(bill_length_mm, bill_depth_mm),
                                 .fns = ~.x/10, .names = "{.col}_cm"))

#list columns in mutated data
colnames(dat.mutate)
```

```
##  [1] "species"           "island"            "bill_length_mm"
##  [4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
##  [7] "sex"               "year"              "bill_length_mm_cm"
## [10] "bill_depth_mm_cm"
```

These variable names are not particularly beautiful but since we're not saving the data frame nor using them to label the plot, it doesn't really matter. If you wanted to change the names, check out tidyverse's `rename(`

) function.

More tidyverse functions

As mentioned above, you can use `apply` or `mutate` for many of the same things. In our code, using the tidyverse is helpful because we also have a `filter` in step 1 that can be combined with `mutate` using a pipe (`%>%`). This is beyond the scope of this workshop but here's an example of how the loop would look with pipes.

```r
#Make 3 by 2 grid for 6 plots
par(mfrow=c(3,2))

for(i in unique(dat$species)){
  dat.filter <- dat %>%
    # Step 1: Filter data to the penguin species
    filter(species == i) %>%
    # Step 2: Convert the mm measurements to cm
    mutate(across(c(bill_length_mm, bill_depth_mm),
                                  .fns = ~.x/10, .names = "{.col}_cm"))

  # Step 3: Plot measurements separated by sex
  plot(x = dat.filter$sex, y = dat.filter$bill_length_mm_cm,
      ylab = "bill length (cm)", xlab = "", main = i)
  plot(x = dat.filter$sex, y = dat.filter$bill_depth_mm_cm,
      ylab = "bill depth (cm)", xlab = "", main = i)
}
```

Nested loops

Another way to tackle our plots is to put another `for` loop within the one we currently have. This will loop through each unique set of species and variables we want to plot. And since we're now working with 1 variable at a time, we don't need to use `apply` or `mutate(across())` anymore.

There are also a couple other loop tricks involved like

- descriptive loop iteration names especially when nesting
- `get( )` to extract data instead of printing the character string
- `gsub( )` statements to create automatic labels

```r
#Make 3 by 2 grid for 6 plots
par(mfrow=c(3,2))

for(penguin.species in unique(dat$species)){
  for(variable in c("bill_length_mm", "bill_depth_mm")){
    dat.filter <- dat %>%
      # Step 1: Filter data to the penguin species
      filter(species == penguin.species) %>%
      # Step 2: Convert the mm measurements to cm
      mutate(variable.cm = get(variable)/10)

    ## Make y-label from variable name
    y.label <- gsub("mm", "(cm)", variable)
    y.label <- gsub("_", " ", y.label)
    # Step 3: Plot measurements separated by sex
    plot(x = dat.filter$sex, y = dat.filter$variable.cm,
      ylab = y.label, xlab = "", main = penguin.species)
  }
}
```

foreach loops

Syntax

`foreach` loops are very similar to `for` loops except they can run in parallel. This means if your single `for` loop takes 10 minutes, it will take about 5 minutes to run with `foreach` on 2 processors.

First, we tell R how many processors it can use. It is important to not use all of your processors and to keep in mind how much RAM each processor needs. R is a RAM hog, so it is easy to accidentally crash R when you run in parallel. The data in this workshop are very small so that's not a concern today. But let's just run on 2 processors as an example.

```
registerDoParallel(cores=2)
```

Then the `foreach` loop is setup as

```
foreach(something = something.else) %dopar% {
    DO THINGS
}
```

Using `foreach`

Let's turn our first `for` loop into a parallel `foreach` loop.

Because we are working with plots, we cannot simply switch out the `for` syntax with `foreach`. This is because `foreach` does not play well with RStudio's 'Plots' pane. The easiest solution to this is to save the plots to your computer, as shown below.

Of note, if our loop output was a data frame or matrix, this issue would not arise and the `foreach` loop would save each processor's output in a list object in your main R environment.

```
foreach(i = unique(dat$species)) %dopar% {
  # Step 1: Filter data to the penguin species
  dat.filter <- dat[dat$species == i, ]

  # Step 2: Convert the mm measurements to cm
  bill_length_cm <- dat.filter$bill_length_mm/10
  bill_depth_cm <- dat.filter$bill_depth_mm/10

  # Step 3: Plot measurements separated by sex
  #Save to PDF named as penguin species
  pdf(paste(i, ".pdf", sep=""))
  #Make 1 by 2 grid for 2 plots
  par(mfrow=c(1,2))
  plot(x = dat.filter$sex, y = bill_length_cm,
      ylab = "bill length (cm)", xlab = "", main = i)
  plot(x = dat.filter$sex, y = bill_depth_cm,
       ylab = "bill depth (cm)", xlab = "", main = i)
  dev.off()
}
```

```
## [[1]]
## pdf
##   2
##
## [[2]]
## pdf
##   2
##
## [[3]]
## pdf
##   2
```

Concluding thoughts

Why do some people hate `for` loops? If you spend time reading on the subject, you'll come across some very strong arguments against loops. This is because they are slow. Despite this, though, I've found them to be very useful in my work. If you're doing a couple 1000 or fewer processes, it is unlikely that you will reach time scales that will effect your life overmuch. I have a lot of loop scripts that take minutes to hours and I don't feel the need to further optimize them - I love an excuse for a tea break!

That being said, if you use loops, it is important to put effort into optimizing the code within the loop. For example, the nested `for` loop at the end of this workshop takes longer to run than the single loops with `apply` or `mutate` in them. Thus, if you can leverage `apply/mutate` (or other specialized R functions) to accomplish your task, that will generally be faster.

How do I choose when to `apply` vs `mutate`? Mostly, this comes down to personal style so don't worry if how you code something is different that how someone else does! Below are some of the things I consider when choosing which road to go down.

- a single function on many rows and/or columns -> `apply`
- unless I'm already working in the tidyverse and have several pre/post filtering steps -> `mutate(across( ))` piped with other tidyverse functions
- multiple functions on different rows and/or columns -> `mutate`
  - While `apply` functions would be faster, I find the automatic renaming done in `mutate` necessary for my sanity when doing several transformations.
  - That said, if my script is taking too long, I'll switch to `apply` after validating a subset of the data with `mutate`

Where can I learn more?

- Dr. Pat Schloss explores intro to advanced R topics on Riffomonas, particularly his on-going Code Club (https://www.riffomonas.org/code_club/) which talked about `for` loops the last couple weeks, especially https://www.youtube.com/watch?v=L2QsTX3q0vM&feature=youtu.be
- Software Carpentry has lessons in R, Git, command line, and other data science areas (https://software-carpentry.org/). A lesson on loops is at http://swcarpentry.github.io/r-novice-inflammation/15-supp-loops-in-depth/index.html

R session

```
sessionInfo()
```

```
## R version 4.0.2 (2020-06-22)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Catalina 10.15.6
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRblas.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
##  [1] doParallel_1.0.15    iterators_1.0.12     foreach_1.5.0
##  [4] palmerpenguins_0.1.0 forcats_0.5.0        stringr_1.4.0
##  [7] dplyr_1.0.2          purrr_0.3.4          readr_1.4.0
## [10] tidyr_1.1.2          tibble_3.0.4         ggplot2_3.3.2
## [13] tidyverse_1.3.0
##
## loaded via a namespace (and not attached):
##  [1] tidyselect_1.1.0 xfun_0.18        haven_2.3.1      colorspace_1.4-1
##  [5] vctrs_0.3.4      generics_0.0.2   htmltools_0.5.0  yaml_2.2.1
##  [9] utf8_1.1.4       blob_1.2.1       rlang_0.4.8      pillar_1.4.6
## [13] glue_1.4.2       withr_2.3.0      DBI_1.1.0        dbplyr_1.4.4
## [17] modelr_0.1.8     readxl_1.3.1     lifecycle_0.2.0  munsell_0.5.0
## [21] gtable_0.3.0     cellranger_1.1.0 rvest_0.3.6      codetools_0.2-16
## [25] evaluate_0.14    knitr_1.30       fansi_0.4.1      broom_0.7.1
## [29] Rcpp_1.0.5       scales_1.1.1     backports_1.1.10 jsonlite_1.7.1
## [33] fs_1.5.0         hms_0.5.3        digest_0.6.26    stringi_1.5.3
## [37] grid_4.0.2       cli_2.1.0        tools_4.0.2      magrittr_1.5
## [41] crayon_1.3.4     pkgconfig_2.0.3  ellipsis_0.3.1   xml2_1.3.2
## [45] reprex_0.3.0     lubridate_1.7.9  assertthat_0.2.1 rmarkdown_2.5
## [49] httr_1.4.2       rstudioapi_0.11  R6_2.4.1         compiler_4.0.2
```