Using loops and map

Abhijit Dasgupta

BIOF 339

Where we are

- Got a start on plotting and creating panelled graphs with ggplot2
- Can modify a data set somewhat
    - `dplyr` verbs (mutate, filter, select, separate, unite)
    - joins
    - pivot_longer/wider

Repetitive copying

For the last homework you had to do the same operations on multiple files and data sets:

- Had to do same processing on multiple data sets
- Had to do same graphs from multiple data sets

So you had to copy and paste code, changing the file name or the data name.

This is generally dangerous

You're better off having the computer do it!!

This week is a bit complicated. Learn by doing, taking the code apart and understand what each part does.

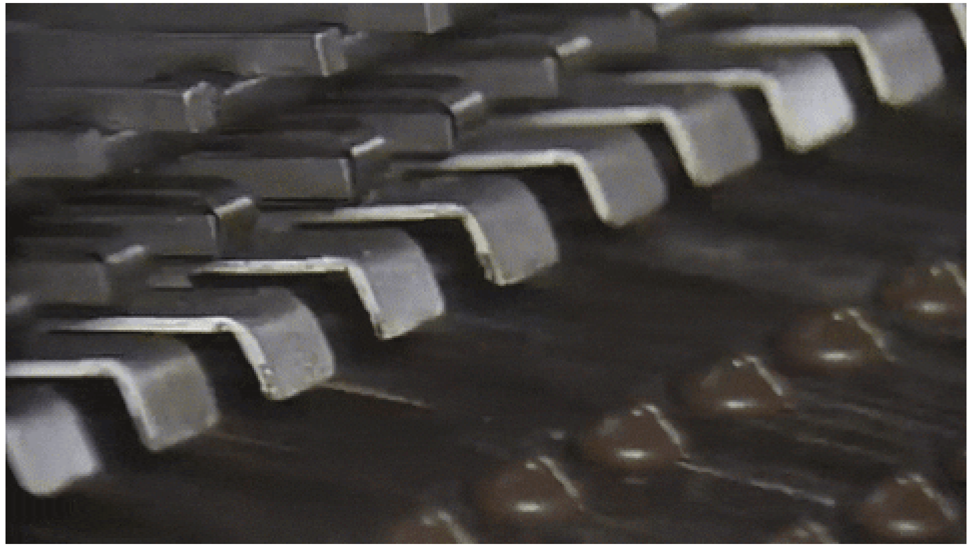
For loops

For loops

For-loops are a computational structure that allows you to do the same thing repeatedly over a loop with some index.

The basic structure is

```
for (variable in vector) {
  <code to execute for each iteration>
}
```



For loops

Using numeric indices

```r
library(fs)
sites <- c('Brain','Colon','Esophagus','Lung','Oral'
dats <- list() # Initialize empty list
for(i in 1:length(sites)){
  dats[[i]] <-
    read_csv(path('../data',
                         paste0(sites[i], '.csv'
                    skip=4)
}
```

Using names

```r
library(fs)
sites <- c('Brain','Colon','Esophagus','Lung','Oral'
dats <- list() # Initialize empty list
for(n in sites){
  dats[[n]] <-
    read_csv(path('../data',
                  paste0(n, '.csv')),
             skip=4)
}
```

Why lists? Because lists can keep any R object, and they're a great way to store a bunch of data that we want to operate on in similar ways

For loops can work on vectors or lists, so basically something array-like where you can count things either by name or by index

Lists

Directly using lists has efficiency advantages. `rio` can load all the datasets into a list, for example.

```
dats <- rio::import_list(path('../data', paste0(sites,'.csv')))
names(dats)
```

```
[1] "Brain"      "Colon"      "Esophagus" "Lung"       "Oral"
```

```
str(dats[['Brain']])
```

```
'data.frame':    43 obs. of  10 variables:
 $ Year of Diagnosis   : chr  "1975-2016" "1975" "1976" "1977" ...
 $ All Races,Both Sexes: num  6.59 5.85 5.82 6.17 5.76 6.12 6.3 6.51 6.42 6.31 ...
 $ All Races,Males     : num  7.88 6.84 7.14 7.76 6.79 7.42 7.58 8.07 7.93 7.6 ...
 $ All Races,Females   : num  5.51 5.01 4.68 4.89 4.91 5.01 5.24 5.2 5.24 5.19 ...
 $ Whites,Both Sexes   : num  7.22 6.21 6.18 6.6 6.1 6.6 6.81 6.9 6.92 6.88 ...
 $ Whites,Males        : num  8.61 7.31 7.51 8.26 7.19 8.03 8.2 8.44 8.57 8.2 ...
 $ Whites,Females      : num  6.04 5.28 5.03 5.27 5.19 5.37 5.65 5.63 5.64 5.74 ...
 $ Blacks,Both Sexes   : num  4.08 4.14 3.32 3.55 3.86 3.69 3.14 5.02 3.71 2.75 ...
 $ Blacks,Males        : num  4.79 4.31 5.37 5.17 4.34 4.19 3.35 7.24 4.4 3.79 ...
 $ Blacks,Females      : chr  "3.51" "3.88" "-" "2.47" ...
 - attr(*, "filename")= chr "../data/Brain.csv"
```

A note on `rio::import` for reading CSV files

The function `rio::import` reads CSV files using `data.table::fread`, and then converts the resulting `data.table` object into a `data.frame` object.

`fread` is not only really fast, but also makes some great automatic choices.

- It looks for and tries to omit non-standard header rows (so we don't need `skip=4`)
- It automatically tries to figure out the right number of rows to import
- With the `check.names=TRUE` option, it fixes issues with column names to make them conformant with R

Using `rio::import` solves a lot of troublesome things in importing regular text files (CSV, TSV, etc), and is recommended

Lists

```
dats <- rio::import_list(path('../data', paste0(sites,'.csv')),
                         setclass = 'tbl',      # Output as tibble
                         check.names = TRUE)    # Check and fix names
str(dats[['Brain']])
```

```
tibble [43 × 10] (S3: tbl_df/tbl/data.frame)
 $ Year.of.Diagnosis   : chr [1:43] "1975-2016" "1975" "1976" "1977" ...
 $ All.Races.Both.Sexes: num [1:43] 6.59 5.85 5.82 6.17 5.76 6.12 6.3 6.51 6.42 6.31 ...
 $ All.Races.Males     : num [1:43] 7.88 6.84 7.14 7.76 6.79 7.42 7.58 8.07 7.93 7.6 ...
 $ All.Races.Females   : num [1:43] 5.51 5.01 4.68 4.89 4.91 5.01 5.24 5.2 5.24 5.19 ...
 $ Whites.Both.Sexes   : num [1:43] 7.22 6.21 6.18 6.6 6.1 6.6 6.81 6.9 6.92 6.88 ...
 $ Whites.Males        : num [1:43] 8.61 7.31 7.51 8.26 7.19 8.03 8.2 8.44 8.57 8.2 ...
 $ Whites.Females      : num [1:43] 6.04 5.28 5.03 5.27 5.19 5.37 5.65 5.63 5.64 5.74 ...
 $ Blacks.Both.Sexes   : num [1:43] 4.08 4.14 3.32 3.55 3.86 3.69 3.14 5.02 3.71 2.75 ...
 $ Blacks.Males        : num [1:43] 4.79 4.31 5.37 5.17 4.34 4.19 3.35 7.24 4.4 3.79 ...
 $ Blacks.Females      : chr [1:43] "3.51" "3.88" "-" "2.47" ...
 - attr(*, "filename")= chr "../data/Brain.csv"
```

purrr::map

map

map is like a for-loop, but strictly for lists. It is more efficient than for-loops. The basic template is:

```
map(<list>, <function>, <function arguments>)
```

For example, if we want to take out the first row of each dataset and make sure all the variables are numeric, we could do:

```
dats <- map(dats,
            function(d){
              d %>% slice(-1) %>%  # remove first row which contains summary data
                mutate(across(everything(), as.numeric))
            })
str(dats[['Brain']])
```

```
tibble [42 × 10] (S3: tbl_df/tbl/data.frame)
 $ Year.of.Diagnosis   : num [1:42] 1975 1976 1977 1978 1979 ...
 $ All.Races.Both.Sexes: num [1:42] 5.85 5.82 6.17 5.76 6.12 6.3 6.51 6.42 6.31 6.12 ...
 $ All.Races.Males     : num [1:42] 6.84 7.14 7.76 6.79 7.42 7.58 8.07 7.93 7.6 7.18 ...
 $ All.Races.Females   : num [1:42] 5.01 4.68 4.89 4.91 5.01 5.24 5.2 5.24 5.19 5.2 ...
 $ Whites.Both.Sexes   : num [1:42] 6.21 6.18 6.6 6.1 6.6 6.81 6.9 6.92 6.88 6.49 ...
 $ Whites.Males        : num [1:42] 7.31 7.51 8.26 7.19 8.03 8.2 8.44 8.57 8.2 7.64 ...
 $ Whites.Females      : num [1:42] 5.28 5.03 5.27 5.19 5.37 5.65 5.63 5.64 5.74 5.49 ...
 $ Blacks.Both.Sexes   : num [1:42] 4.14 3.32 3.55 3.86 3.69 3.14 5.02 3.71 2.75 4.53 ...
 $ Blacks.Males        : num [1:42] 4.31 5.37 5.17 4.34 4.19 3.35 7.24 4.4 3.79 5.34 ...
 $ Blacks.Females      : num [1:42] 3.88 NA 2.47 3.51 3.23 2.92 3.16 3.05 1.84 3.88 ...
 - attr(*, "filename")= chr "../data/Brain.csv"
```

map

map is like a for-loop, but strictly for lists. It is more efficient than for-loops. The basic template is:

```
map(<list>, <function>, <function arguments>)
```

For example, if we want to take out the first row of each dataset and make sure all the variables are numeric, we could do:

```
dats <- map(dats, function(d){
  d %>% slice(-1) %>%  # remove first row
    mutate_all( as.numeric)
})
str(dats[['Brain']])
```

The argument for the function inside the map function is an element of the list. In this case, it is a data frame.

The output of map is a list the same length as the input list.

Here, we are overwriting the original data. This is fine if you're sure about the transformations, but normally you might want to save the result with a different name first until you're sure of what you're doing

map

I don't like the names with dots, say 😄. I can just apply a function to each data set to fix that.

```
dats <- map(dats, janitor::clean_names)
str(dats[['Oral']])
```

```
tibble [42 × 10] (S3: tbl_df/tbl/data.frame)
 $ year_of_diagnosis    : num [1:42] 1975 1976 1977 1978 1979 ...
 $ all_races_both_sexes: num [1:42] 13.2 13.3 12.7 13.4 14 ...
 $ all_races_males      : num [1:42] 21.2 21 20.1 20.9 21.9 ...
 $ all_races_females    : num [1:42] 7.09 7.39 6.94 7.71 7.98 7.91 7.91 7.93 7.24 7.86 ...
 $ whites_both_sexes    : num [1:42] 13.3 13.2 12.6 13.2 13.7 ...
 $ whites_males         : num [1:42] 21.7 21.1 19.9 20.7 21.6 ...
 $ whites_females       : num [1:42] 6.94 7.38 7 7.57 7.72 7.62 7.95 7.85 7.28 7.64 ...
 $ blacks_both_sexes    : num [1:42] 13.4 15.2 14.5 15.9 18.5 ...
 $ blacks_males         : num [1:42] 20.2 23.8 23.9 26 28.2 ...
 $ blacks_females       : num [1:42] 8.23 8.37 6.77 8.18 10.77 ...
 - attr(*, "filename")= chr "../data/Oral.csv"
```

Note that `janitor::clean_names` takes a data.frame/tibble as its first argument (as all tidyverse functions), and `dats` is a list of tibbles. So `map` applies the `clean_names` function to each tibble in the list, and returns the result as a list

map

Now let's split up by sexes

```
dats_all <- map(dats, select, year_of_diagnosis, ends_with('sexes'))
dats_male <- map(dats, select, year_of_diagnosis, ends_with('_males'))
dats_female <- map(dats, select, year_of_diagnosis, ends_with('females'))
str(dats_all[['Esophagus']])
```

```
tibble [42 × 4] (S3: tbl_df/tbl/data.frame)
 $ year_of_diagnosis   : num [1:42] 1975 1976 1977 1978 1979 ...
 $ all_races_both_sexes: num [1:42] 4.14 4.3 4.06 4.12 4.42 4.27 4.14 4.26 4.29 4.18 ...
 $ whites_both_sexes   : num [1:42] 3.55 3.72 3.33 3.41 3.73 3.54 3.31 3.46 3.57 3.52 ...
 $ blacks_both_sexes   : num [1:42] 10.9 10.7 12 13.1 12.9 ...
 - attr(*, "filename")= chr "../data/Esophagus.csv"
```

Here I used the form `map(<list>, <function>, <function arguments>)`.

Earlier I had used `map(<list>,<function definition>)` and `map(<list>, <function>)` with no (i.e., default) arguments.

Note, `map` assumes that each element of the list is the **first** argument of the function, and so you only have to specify from the 2nd argument onwards

map

Let's make the column headers of each dataset reflect the site, so that when we join we can keep the sites separate

```r
for(n in sites){
  names(dats_all[[n]]) <- str_replace(names(dats_all[[n]]), 'both_sexes',n)
  names(dats_male[[n]]) <- str_replace(names(dats_male[[n]]), 'male',n)
  names(dats_female[[n]]) <- str_replace(names(dats_female[[n]]), 'female',n)
}
names(dats_all[['Esophagus']])
```

```
[1] "year_of_diagnosis"   "all_races_Esophagus" "whites_Esophagus"
[4] "blacks_Esophagus"
```

map

When we joined these data sets, we had to repeatedly use `left_join` to create the final data set.

```
joined_all <- dats_all[['Brain']]
for(n in setdiff(names(dats2_all), 'Brain')){
  joined_all <- joined %>% left_join(dats_all[['n']]
}
```

There is a shortcut to this repeated operation of a function with two inputs as applied to a list successively.

```
joined_all <- Reduce(left_join, dats_all)
joined_male <- Reduce(left_join, dats_male)
joined_female <- Reduce(left_join, dats_female)
```

```
tibble [42 × 16] (S3: tbl_df/tbl/data.frame)
 $ year_of_diagnosis  : num [1:42] 1975 1976 1977 1978 1979 ...
 $ all_races_Brain    : num [1:42] 5.85 5.82 6.17 5.76 6.12 6.3 6.51 6.42 6.31 6.12 ...
 $ whites_Brain       : num [1:42] 6.21 6.18 6.6 6.1 6.6 6.81 6.9 6.92 6.88 6.49 ...
 $ blacks_Brain       : num [1:42] 4.14 3.32 3.55 3.86 3.69 3.14 5.02 3.71 2.75 4.53 ...
 $ all_races_Colon    : num [1:42] 59.5 61.3 62.4 62 62.4 ...
 $ whites_Colon       : num [1:42] 60.2 62.2 63.2 62.8 63 ...
 $ blacks_Colon       : num [1:42] 56.9 55 60.8 62.2 58.6 ...
 $ all_races_Esophagus: num [1:42] 4.14 4.3 4.06 4.12 4.42 4.27 4.14 4.26 4.29 4.18 ...
 $ whites_Esophagus   : num [1:42] 3.55 3.72 3.33 3.41 3.73 3.54 3.31 3.46 3.57 3.52 ...
 $ blacks_Esophagus   : num [1:42] 10.9 10.7 12 13.1 12.9 ...
 $ all_races_Lung     : num [1:42] 52.2 55.4 56.7 57.8 58.6 ...
 $ whites_Lung        : num [1:42] 51.9 54.6 55.9 57.2 58 ...
 $ blacks_Lung        : num [1:42] 64.5 72.3 73.6 74.4 74.5 ...
 $ all_races_Oral     : num [1:42] 13.2 13.3 12.7 13.4 14 ...
 $ whites_Oral        : num [1:42] 13.3 13.2 12.6 13.2 13.7 ...
 $ blacks_Oral        : num [1:42] 13.4 15.2 14.5 15.9 18.5 ...
 - attr(*, "filename")= chr "../data/Brain.csv"
```

map

Next, we want to separate the races from the sites, after a `gather`. The `all_races` will pose a problem if we split on `_`. Let's fix that.

```r
names(joined_all) <- str_replace(names(joined_all), 'all_races','allraces')
names(joined_male) <- str_replace(names(joined_male), 'all_races','allraces')
names(joined_female) <- str_replace(names(joined_female), 'all_races','allraces')
```

map

Now, for each of these, we need to gather then separate. We'll put the data sets in a list first

```
joined <- list('both'=joined_all, 'male'=joined_male, 'female'=joined_female)
joined <- map(joined,
             function(d){
               d %>%
                pivot_longer(names_to='variable', values_to = 'rate',
                             cols=c(-year_of_diagnosis)) %>%
                 separate(variable, c('race','site'), sep='_')
             })
str(joined[['both']])
```

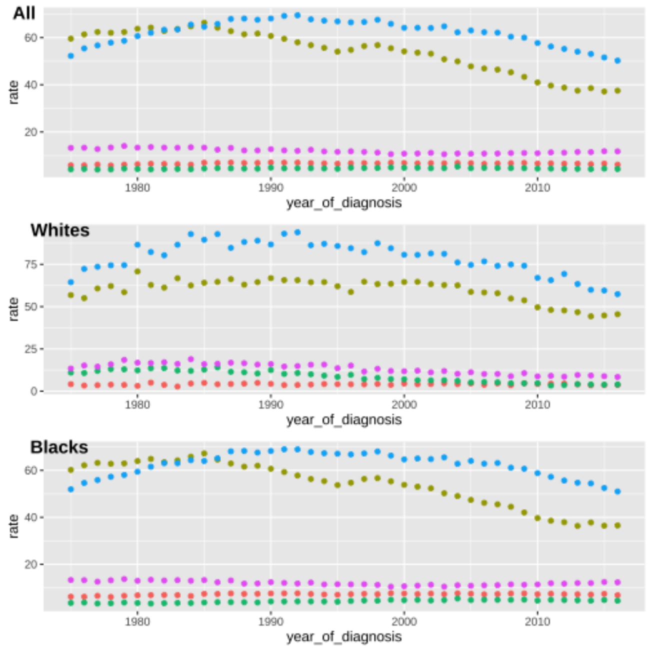
```
tibble [630 × 4] (S3: tbl_df/tbl/data.frame)
 $ year_of_diagnosis: num [1:630] 1975 1975 1975 1975 1975 ...
 $ race             : chr [1:630] "allraces" "whites" "blacks" "allraces" ...
 $ site             : chr [1:630] "Brain" "Brain" "Brain" "Colon" ...
 $ rate             : num [1:630] 5.85 6.21 4.14 59.54 60.2 ...
```

Okay, this is voodoo 😈 Not really. Grab one of the datasets and work out what you need. Since you'll be doing the same to all of the datasets, you use `map` on the list of datasets

Final graphing

Now we're in a position to do the graphing.

```
pltlist <- joined[['both']] %>% group_split(race) %>%
  map(function(d) {ggplot(d,
                          aes(x = year_of_diagnosis,
                              y = rate,
                              color=site))+
  geom_point(show.legend = F) })
cowplot::plot_grid(plotlist=pltlist, ncol=1,
                   labels=c('All','Whites','Blacks'))
```



I'm using quite advanced R here, but hopefully you'll learn by example.

`group_split` splits the dataset by the values of the grouping variable into a list

(Yes, your homework asked for a different panel placement)