List:
Base R: lapply, sapply
plyr: l*ply
tidyverse: map

Array:
Base R: apply
plyr: a*ply
tidyverse: summarize

Subset of a data frame:
Base R: tapply; split + lapply
plyr: d*ply
tidyverse: group-by + summarize

Purrr and a Bit of Dplyr

Statistical Computing, STA3005

Monday Jan 22, 2023

Last chapter: Data frames and apply

- Data frames are a representation of the "classic" data table in R: rows are observations/cases, columns are variables/features
- Each column can be a different data type (but must be the same length)
- Factors represent a vector by categories (`levels`) and integer indices
- `subset()` : function for extracting rows of a data frame meeting a condition
- `split()` : function for splitting up rows of a data frame, according to a factor variable
- `apply()` : function for applying a given routine to rows or columns of a matrix or data frame
- `lapply()` : similar, but used for applying a routine to elements of a vector or list
- `sapply()` : similar, but will try to simplify the return type, in comparison to `lapply()`
- `tapply()` : function for applying a given routine to groups of elements in a vector or list, according to a factor variable

Part I: Three types of implicit iterations

Common iteration tasks

Here's a basic breakdown for common iteration tasks that we encounter in R: we iterate over

- elements of a list
- dimensions of an array (e.g., rows/columns of a matrix)

- subsets of a data frame induced by one or more factors

For simplicity, we usually implement the apply family of functions in base R: `lapply()`, `sapply()`, `apply()`, `tapply()`, etc, instead of using explicit iterations (`for` or `while` loop). Besides base R, we introduce two alternative ways: using `plyr` or `tidyverse`.

Why do we look anywhere else?

Because some alternatives offer better **consistency**

- With the apply family of functions, there are some inconsistencies in both the **interfaces** to the functions, as well as their **outputs**
- This can both slow down learning and also lead to inefficiencies in practice (frequent checking and post-processing of results)

However, the world isn't black-and-white: base R still has its advantages, and the best thing you can do is to be informed and well-versed in using all the major options!

Part II: `plyr` package

The `plyr` package used to be one of the most popular (most downloaded) R packages of all-time. It was more popular in the late 2000s and early 2010s. It provides extremely useful family of apply-like functions.

- Advantage over the built-in `apply()` family is its **consistency**
- All `plyr` functions are of the form `**ply()`
- Replace `**` with characters denoting types:
  - First character: input type, one of `a` (array), `d` (data frame), `l` (list)
  - Second character: output type, one of `a`, `d`, `l`, or `_` (drop)

Installing and loading packages

Before introducing how to use the `plyr` package, we should install it first by

```
install.packages("plyr")
```

You can also select “Tools” –> “Install Packages” from the RStudio menu.

Now we’ll load the package and check the installation.

```
library(plyr)
```

a*ply() : input is an array

The signature for all `a*ply()` functions is:

```
a*ply(.data, .margins, .fun, ...)
```

- `.data` : an array
- `.margins` : index (or indices) to split the array by
- `.fun` : the function to be applied to each piece
- `...` : additional arguments to be passed to the function

Note that this resembles:

```
apply(X, MARGIN, FUN, ...)
```

Examples of a*ply()

```
head(aaply(state.x77, 1, mean)) # Get back array
```

```
##    Alabama     Alaska    Arizona   Arkansas California
##   7261.819  71676.601  15039.031   7202.570  22854.839  1
```

```
head(adply(state.x77, 1, mean)) # Get back data frame
```

```
##           X1        V1
## 1    Alabama  7261.819
## 2     Alaska 71676.601
## 3    Arizona 15039.031
## 4   Arkansas  7202.570
## 5 California 22854.839
## 6   Colorado 13937.558
```

```r
head(alply(state.x77, 1, mean)) # Get back list
```

```
## $`1`
## [1] 7261.819
##
## $`2`
## [1] 71676.6
##
## $`3`
## [1] 15039.03
##
## $`4`
## [1] 7202.57
##
## $`5`
## [1] 22854.84
##
## $`6`
## [1] 13937.56
```

```r
mean.sd = function(x) c("mean"=mean(x), "sd"=sd(x))
head(aaply(state.x77, 1, mean.sd)) # Get back array
```

```
##
## X1               mean        sd
##   Alabama     7261.819  17629.67
##   Alaska     71676.601 199923.19
##   Arizona    15039.031  39784.17
##   Arkansas    7202.570  18123.15
##   California 22854.839  54439.39
##   Colorado   13937.558  36339.05
```

```r
head(adply(state.x77, 1, mean.sd)) # Get back data frame
```

```
##           X1      mean        sd
## 1    Alabama  7261.819  17629.67
## 2     Alaska 71676.601 199923.19
## 3    Arizona 15039.031  39784.17
## 4   Arkansas  7202.570  18123.15
```

```
## 4   Arkansas  7202.570  18123.15
## 5 California 22854.839  54439.39
## 6   Colorado 13937.558  36339.05
```

```
head(alply(state.x77, 1, mean.sd)) # Get back list
```

```
## $`1`
##      mean        sd
##  7261.819 17629.674
##
## $`2`
##     mean       sd
##  71676.6 199923.2
##
## $`3`
##     mean       sd
## 15039.03 39784.17
##
## $`4`
##     mean       sd
##  7202.57 18123.15
##
## $`5`
##     mean       sd
## 22854.84 54439.39
##
## $`6`
##     mean       sd
## 13937.56 36339.05
```

`l*ply()` : input is a list

The signature for all `l*ply()` functions is:

```
l*ply(.data, .fun, ...)
```

- `.data` : a list
- `.fun` : the function to be applied to each element
- `...` : additional arguments to be passed to the function

Note that this resembles:

```
lapply(X, FUN, ...)
```

Examples of l*ply()

注：要特别注意 coercion

```
my.list = list(nums=rnorm(1000), lets=letters,
               pops=state.x77[,"Population"])
laply(my.list, range) # Get back array
```

```
##      1                    2
## [1,] "-3.42838834843137" "3.11487007123291"
## [2,] "a"                  "z"
## [3,] "365"                "21198"
```

```
ldply(my.list, range) # Get back data frame
```

```
##    .id                 V1                V2
## 1 nums -3.42838834843137 3.11487007123291
## 2 lets                  a                 z
## 3 pops                365             21198
```

都是 char

```
llply(my.list, range) # Get back list
```

```
## $nums
## [1] -3.428388  3.114870
##
## $lets
## [1] "a" "z"
##
## $pops
## [1]   365 21198
```

```
laply(my.list, summary)
```

(会报错)

```
## Error: Results must have one or more dimensions.
```

```
# Doesn't work! Outputs have different types/lengths
ldply(my.list, summary)
```

```
## Error in list_to_dataframe(res, attr(.data, "split_labels
```

```
# Doesn't work! Outputs have different types/lengths
llply(my.list, summary) # Works just fine
```

```
## $nums
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -3.42839 -0.71356  0.05133  0.02188  0.68876  3.11487
##
## $lets
##    Length     Class      Mode
##        26 character character
##
## $pops
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     365    1080    2838    4246    4968   21198
```

d*ply() : the input is a data frame

The signature for all `d*ply()` functions is:

```
d*ply(.data, .variables, .fun, ...)
```

- `.data` : a data frame
- `.variables` : variable (or variables) to split the data frame by
- `.fun` : the function to be applied to each piece
- `...` : additional arguments to be passed to the function

Note that this resembles:

```
tapply(X, INDEX, FUN, ...)
```

Examples of d*ply()

```r
state.df = data.frame(state.x77, Region=state.region,
                      Division=state.division)
# Get back array
daply(state.df, .(Region), function(df) mean.sd(df$Frost))
```

(注意格式)

```
##
## Region              mean       sd
##   Northeast     132.7778 30.89408
##   South          64.6250 31.30682
##   North Central 138.8333 23.89307
##   West          102.1538 68.87652
```

dim=(4,2)

```r
# Get back df
ddply(state.df, .(Region), function(df) mean.sd(df$Frost))
```

```
##          Region     mean       sd
## 1     Northeast 132.7778 30.89408
## 2         South  64.6250 31.30682
## 3 North Central 138.8333 23.89307
## 4          West 102.1538 68.87652
```

dim=(4,3)

```r
# Get back list
dlply(state.df, .(Region), function(df) mean.sd(df$Frost))
```

```
## $Northeast
##      mean        sd
## 132.77778  30.89408
##
## $South
##     mean       sd
## 64.62500 31.30682
##
## $`North Central`
##      mean        sd
## 138.83333  23.89307
##
## $West
```

```
##      mean        sd
## 102.15385  68.87652
##
## attr(,"split_type")
## [1] "data.frame"
## attr(,"split_labels")
##          Region
## 1     Northeast
## 2         South
## 3 North Central
## 4          West
```

Splitting on two (or more) variables

The function `d*ply()` makes it very easy to split on two (or more) variables: we just specify them, separated by a "," in the `.variables` argument

```
# First create a variable that indicates
# whether the area is big or not
state.df$AreaBig = state.df$Area > 50000
# Now use (say) ddply() to compute the mean and sd Frost,
# for each region, but separately over big and small areas
ddply(state.df, .(Region, AreaBig),
      function(df) mean.sd(df$Frost))
```

```
##          Region AreaBig     mean        sd
## 1     Northeast   FALSE 132.7778 30.894084
## 2         South   FALSE  76.1000 28.512960
## 3         South    TRUE  45.5000 27.833433
## 4 North Central   FALSE 123.0000  1.414214
## 5 North Central    TRUE 142.0000 25.113078
## 6          West   FALSE   0.0000        NA
## 7          West    TRUE 110.6667 64.401205
```

```
# We can also create factor variables on-the-fly with I()
ddply(state.df, .(Region, I(Area > 50000)),
      function(df) mean.sd(df$Frost))
```

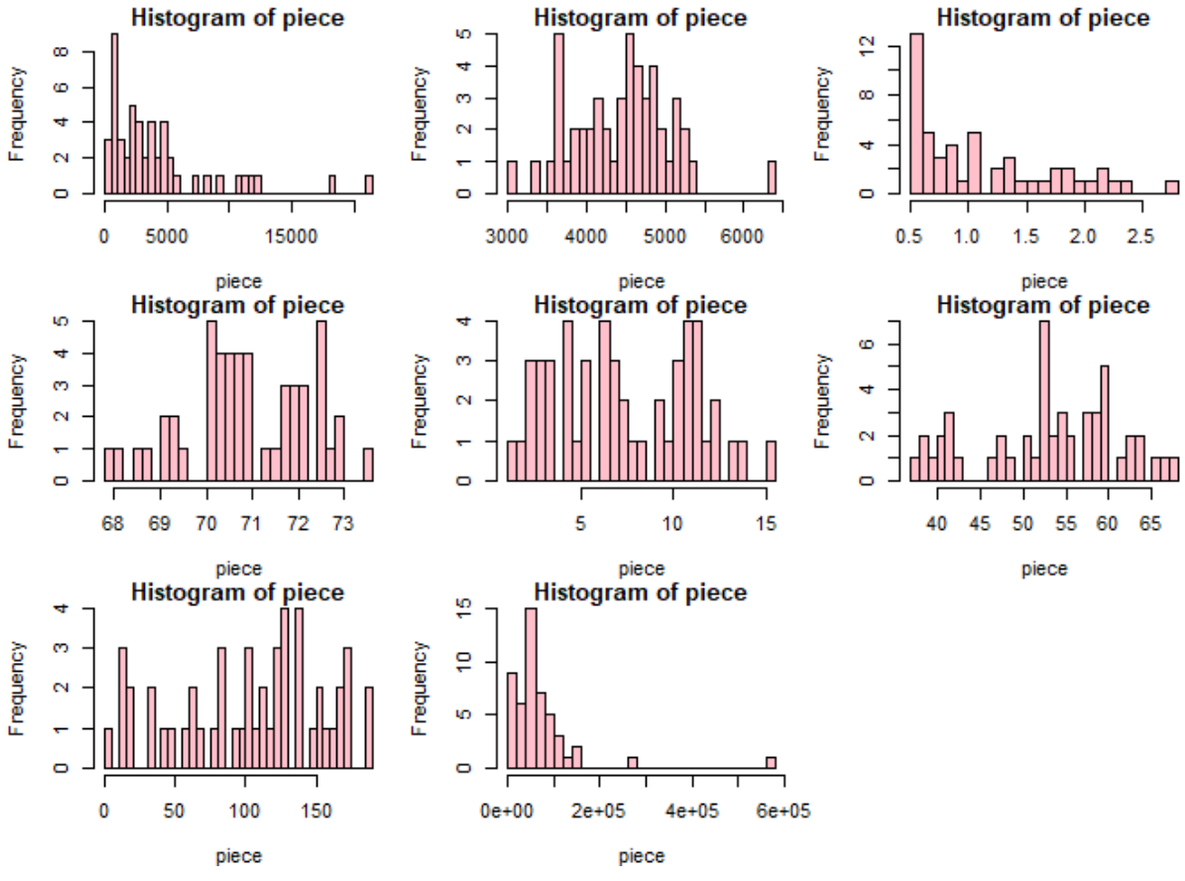
Boolean factor

```
##          Region I(Area > 50000)     mean        sd
## 1     Northeast           FALSE 132.7778 30.894084
## 2         South           FALSE  76.1000 28.512960
## 3         South            TRUE  45.5000 27.833433
## 4 North Central           FALSE 123.0000  1.414214
## 5 North Central            TRUE 142.0000 25.113078
## 6          West           FALSE   0.0000        NA
## 7          West            TRUE 110.6667 64.401205
```

The fourth option for *

The fourth option for * is _ : the function `a_ply()` (or `l_ply()` or `d_ply()`) has no explicit return object, but still runs the given function over the given array (or list), possibly producing side effects

```
par(mfrow=c(3,3), mar=c(4,4,1,1))
a_ply(state.x77, 2, hist, breaks=30, col="pink")
```



Summary of plyr

- All plyr functions are of the form **ply() :
  - First character: input type, one of a , d , l

- Second character: output type, one of `a`, `d`, `l`, or `_` (drop)

Unfortunately, `plyr` is no longer under active development and that development is now happening elsewhere (mainly in the `tidyverse`). Nevertheless, some people still like it.

Part III: `purrr`

`tidyverse` package

The `tidyverse` is a coherent collection of packages in R for data science (and `tidyverse` itself is actually a package that loads all its constituent packages). Packages include:

数据整理

- **Data wrangling**: `dplyr`, `tidyr`, `readr`
- **Iteration**: `purrr`
- **Visualization**: `ggplot2`

We'll cover `purrr` and a bit of `dplyr` in the following. Next chapter we will do more `dplyr`, and some `tidyr`.

What is `purrr`?

`purrr` is a package that is part of the `tidyverse`. It offers a family of functions for iterating (mainly over lists) that can be seen as alternatives to base R's family of apply functions

- Compared to base R, they are more consistent
- Compared to `plyr`, they can often be faster

Below we install `tidyverse` which gives us the packages we need (`purrr` and `dplyr`).

```
install.packages("tidyverse")
```

Here, we load the `tidyverse` package.

```
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.
```

```
## Warning: package 'ggplot2' was built under R version 4.2.
```

```
## Warning: package 'tibble' was built under R version 4.2.3
```

```
## Warning: package 'tidyr' was built under R version 4.2.3
```

```
## Warning: package 'readr' was built under R version 4.2.3
```

```
## Warning: package 'dplyr' was built under R version 4.2.3
```

```
## Warning: package 'forcats' was built under R version 4.2.
```

```
## Warning: package 'lubridate' was built under R version 4.
```

```
## ── Attaching core tidyverse packages ──────────────────────
## ✔ dplyr     1.1.3     ✔ readr     2.1.4
## ✔ forcats   1.0.0     ✔ stringr   1.5.0
## ✔ ggplot2   3.4.3     ✔ tibble    3.2.1
## ✔ lubridate 1.9.3     ✔ tidyr     1.3.0
## ✔ purrr     1.0.1
## ── Conflicts ────────────────────────────────────────── t
## ✖ dplyr::arrange()   masks plyr::arrange()
## ✖ purrr::compact()   masks plyr::compact()
## ✖ dplyr::count()     masks plyr::count()
## ✖ dplyr::desc()      masks plyr::desc()
## ✖ dplyr::failwith()  masks plyr::failwith()
## ✖ dplyr::filter()    masks stats::filter()
## ✖ dplyr::id()        masks plyr::id()
## ✖ dplyr::lag()       masks stats::lag()
## ✖ dplyr::mutate()    masks plyr::mutate()
## ✖ dplyr::rename()    masks plyr::rename()
## ✖ dplyr::summarise() masks plyr::summarise()
## ✖ dplyr::summarize() masks plyr::summarize()
## ℹ Use the ]8;;http://conflicted.r-lib.org/ conflicted pac
```

类似于 C++，可以使用 namespace :: function name 来使用存在 conflict 的 function，如 dplyr::filter( )

Note: Loading the `tidyverse` package after `plyr` will cause namespace overlapping issues. In particular, some functions in `dplyr` will mask the corresponding functions with the same names in `plyr`. Better to just load only what you need.

The map family

map_xx 要求子返回一个 single value，而不是 vector

`purrr` offers a family of **map functions**, which allow you to apply a function across different chunks of data (primarily used with lists). Offers an alternative base R's apply functions. Summary of functions:

- `map()`: apply a function across elements of a list or vector
- `map_dbl()`, `map_lgl()`, `map_chr()`: same, but return a vector of a particular data type → logical
- `map_dfr()`, `map_dfc()`: same, but return a data frame

apply( ) 为 organize by row ← by row　　by column

`map()`: list in, list out

The `map()` function is an alternative to `lapply()`. It has the following simple form: `map(x, f)`, where `x` is a list or vector, and `f` is a function. It always returns a list

```
my.list = list(nums=seq(0.1,0.6,by=0.1), chars=letters[1:12]
               bools=sample(c(TRUE,FALSE), 6, replace=TRUE))
map(my.list, length)
```

```
## $nums
## [1] 6
##
## $chars
## [1] 12
##
## $bools
## [1] 6
```

```
# Base R is just as easy
lapply(my.list, length)
```

```
## $nums
## [1] 6
##
## $chars
## [1] 12
##
## $bools
## [1] 6
```

`map_dbl()` : list in, numeric out

The `map_dbl()` function is an alternative to `sapply()`. It has the form: `map_dbl(x, f)`, where `x` is a list or vector, and `f` is a function that returns a numeric value (when applied to each element of `x`)

Similarly:

- `map_int()` returns an integer vector
- `map_lgl()` returns a logical vector
- `map_chr()` returns a character vector

```
map_dbl(my.list, length)
```

```
##  nums chars bools
##     6    12     6
```

```
map_chr(my.list, length)
```

```
## Warning: Automatic coercion from integer to character was
## ℹ Please use an explicit call to `as.character()` within
## Call `lifecycle::last_lifecycle_warnings()` to see where
## generated.
```

(提示出现了 automatic coercion，便于 debug)

```
##  nums chars bools
##   "6"  "12"   "6"
```

```r
# Base R is a bit more complicated
sapply(my.list, length)
```

```
##  nums chars bools
##     6    12     6
```

drop 掉 list structure

```r
unlist(lapply(my.list, length))
```

```
##  nums chars bools
##     6    12     6
```

类似于 sapply，但要指明 returned value type

```r
vapply(my.list, FUN=length, FUN.VALUE=numeric(1))
```

```
##  nums chars bools
##     6    12     6
```

Applying a custom function

As before (with the apply family), we can of course apply a custom function, and define it “on-the-fly”

Also, you need to install the package before using it

```r
install.packages("repurrrsive")
```

```r
library(repurrrsive) # Load Game of Thrones data set
class(got_chars)
```

一个内置的 dataset

```
## [1] "list"
```

```r
class(got_chars[[1]])
```

```
## [1] "list"
```

list 内嵌套了 list

```
names(got_chars[[1]])
```

```
##  [1] "url"         "id"          "name"        "gender"
##  [6] "born"        "died"        "alive"       "titles"
## [11] "father"      "mother"      "spouse"      "allegianc
## [16] "povBooks"    "tvSeries"    "playedBy"
```

```
map_chr(got_chars, function(x) { return(x$name) })
```

```
##  [1] "Theon Greyjoy"      "Tyrion Lannister"   "Victarion
##  [4] "Will"               "Areo Hotah"         "Chett"
##  [7] "Cressen"            "Arianne Martell"    "Daenerys
## [10] "Davos Seaworth"     "Arya Stark"         "Arys Oakh
## [13] "Asha Greyjoy"       "Barristan Selmy"    "Varamyr"
## [16] "Brandon Stark"      "Brienne of Tarth"   "Catelyn S
## [19] "Cersei Lannister"   "Eddard Stark"       "Jaime Lan
## [22] "Jon Connington"     "Jon Snow"           "Aeron Gre
## [25] "Kevan Lannister"    "Melisandre"         "Merrett F
## [28] "Quentyn Martell"    "Samwell Tarly"      "Sansa Sta
```

Example: Produce an integer vector that represents how many allegiances each character holds.

```
got_chars[[1]]$allegiances
```

```
## [1] "House Greyjoy of Pyke"
```

```
map_int(got_chars, function(x) length(x$allegiances))
```

```
##  [1] 1 1 1 0 1 0 0 1 1 2 1 1 2 2 0 1 3 2 1 1 1 2 1 1 1 0
```

Extracting elements

Handily, the map functions all allow the second argument to be an integer or string, and treat this internally as an appropriate extractor function

```
map_chr(got_chars, "name")
```

```
##  [1] "Theon Greyjoy"      "Tyrion Lannister"   "Victarion
##  [4] "Will"               "Areo Hotah"         "Chett"
##  [7] "Cressen"            "Arianne Martell"    "Daenerys
## [10] "Davos Seaworth"     "Arya Stark"         "Arys Oakh
## [13] "Asha Greyjoy"       "Barristan Selmy"    "Varamyr"
## [16] "Brandon Stark"      "Brienne of Tarth"   "Catelyn S
## [19] "Cersei Lannister"   "Eddard Stark"       "Jaime Lan
## [22] "Jon Connington"     "Jon Snow"           "Aeron Gre
## [25] "Kevan Lannister"    "Melisandre"         "Merrett F
## [28] "Quentyn Martell"    "Samwell Tarly"      "Sansa Sta
```

```
map_lgl(got_chars, "alive")
```

```
##  [1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRU
## [13]  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRU
## [25] FALSE  TRUE FALSE FALSE  TRUE  TRUE
```

Interestingly, we can actually do the following in base R: `` `[`() `` and `` `[[`() `` are functions that act in the following way for an integer `x` and index `i`

- `` `[`(x, i) `` is equivalent to `x[i]` (keep list structure)
- `` `[[`(x, i) `` is equivalent to `x[[i]]` (drop list structure)

(This works whether `i` is an integer or a string) 对每个 got_chars[[i]]选取"name"

got_chars[[i]][["name"]] for ∀ i

```
sapply(got_chars, `[[`, "name")
```

此处会返回一个 vector；若 [[改成 [，则返回 list（无法简化为 vector）

```
##  [1] "Theon Greyjoy"      "Tyrion Lannister"   "Victarion
##  [4] "Will"               "Areo Hotah"         "Chett"
##  [7] "Cressen"            "Arianne Martell"    "Daenerys
## [10] "Davos Seaworth"     "Arya Stark"         "Arys Oakh
## [13] "Asha Greyjoy"       "Barristan Selmy"    "Varamyr"
## [16] "Brandon Stark"      "Brienne of Tarth"   "Catelyn S
## [19] "Cersei Lannister"   "Eddard Stark"       "Jaime Lan
## [22] "Jon Connington"     "Jon Snow"           "Aeron Gre
```

```
## [25] "Kevan Lannister"    "Melisandre"         "Merrett F
## [28] "Quentyn Martell"    "Samwell Tarly"      "Sansa Sta
```

```
sapply(got_chars, `[[`, "alive")
```

```
##  [1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRU
## [13]  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRU
## [25] FALSE  TRUE FALSE FALSE  TRUE  TRUE
```

Part III: *A bit of `dplyr`: `map_dfr()` and `map_dfc()`*

`map_dfr()` and `map_dfc()`: list in, data frame out

The `map_dfr()` and `map_dfc()` functions iterate a function call over a list or vector, but automatically combine the results into a data frame. They differ in whether that data frame is formed by **r**ow-binding or **c**olumn-binding

```
map_dfr(got_chars, `[`, c("name", "alive"))
```

重要

```
## # A tibble: 30 × 2
##    name               alive
##    <chr>              <lgl>
##  1 Theon Greyjoy      TRUE
##  2 Tyrion Lannister   TRUE
##  3 Victarion Greyjoy  TRUE
##  4 Will               FALSE
##  5 Areo Hotah         TRUE
##  6 Chett              FALSE
##  7 Cressen            FALSE
##  8 Arianne Martell    TRUE
##  9 Daenerys Targaryen TRUE
## 10 Davos Seaworth     TRUE
## # ℹ 20 more rows
```

```r
# Base R is much less convenient
data.frame(name = sapply(got_chars, `[[`, "name"),
           alive = sapply(got_chars, `[[`, "alive"))
```

需要 sapply 返回 vector 而不是 list

```
##                  name alive
## 1       Theon Greyjoy  TRUE
## 2    Tyrion Lannister  TRUE
## 3   Victarion Greyjoy  TRUE

## 4                Will FALSE
## 5          Areo Hotah  TRUE
## 6               Chett FALSE
## 7             Cressen FALSE
## 8     Arianne Martell  TRUE
## 9  Daenerys Targaryen  TRUE
## 10     Davos Seaworth  TRUE
## 11         Arya Stark  TRUE
## 12      Arys Oakheart FALSE
## 13       Asha Greyjoy  TRUE
## 14    Barristan Selmy  TRUE
## 15            Varamyr FALSE
## 16      Brandon Stark  TRUE
## 17   Brienne of Tarth  TRUE
## 18      Catelyn Stark FALSE
## 19   Cersei Lannister  TRUE
## 20       Eddard Stark FALSE
## 21    Jaime Lannister  TRUE
## 22     Jon Connington  TRUE
## 23           Jon Snow  TRUE
## 24      Aeron Greyjoy  TRUE
## 25    Kevan Lannister FALSE
## 26         Melisandre  TRUE
## 27       Merrett Frey FALSE
## 28    Quentyn Martell FALSE
## 29      Samwell Tarly  TRUE
## 30        Sansa Stark  TRUE
```

Note: the first example uses **extra arguments**; the map functions work just like the apply functions in this regard

`map_dfr()` requires matched names

Produce a matrix that has dimension 30 x 6, with each column representing a TV season, and each row a character. The matrix should have a value of `TRUE` in position (i,j) if character i was in season j, and `FALSE` otherwise.

```
got_chars[[1]]$tvSeries
```

```
## [1] "Season 1" "Season 2" "Season 3" "Season 4" "Season 5
```

```
six_season <- c("Season 1", "Season 2", "Season 3", "Season
```

Here, we introduce a useful operator `%in%` for matching, in particular, whether the elements in a vector `vec1` exist in another vector `vec2`. `vec1 %in% vec2` returns a logical vector with the same length as `vec1`. If there is a match in `vec2`, then return `TRUE`, otherwise return `FALSE`. A toy example:

```
1:5 %in% 3:6
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE
```

```
map_dfr(got_chars, function(x) six_season %in% x$tvSeries)
```

(%in%不会给 results 命名)

```
## Error in `dplyr::bind_rows()`:
## ! Argument 1 must be a data frame or a named atomic vecto
```

```
map_dfr(got_chars, function(x){
  res <- six_season %in% x$tvSeries
  names(res) <- six_season
  return(res)
})
```

(需要手动命名)

```
## # A tibble: 30 × 6
##    `Season 1` `Season 2` `Season 3` `Season 4` `Season 5`
##    <lgl>      <lgl>      <lgl>      <lgl>      <lgl>
```

```
##  1 TRUE       TRUE       TRUE       TRUE       TRUE
##  2 TRUE       TRUE       TRUE       TRUE       TRUE
##  3 FALSE      FALSE      FALSE      FALSE      FALSE
##  4 FALSE      FALSE      FALSE      FALSE      FALSE
##  5 FALSE      FALSE      FALSE      FALSE      TRUE
##  6 FALSE      FALSE      FALSE      FALSE      FALSE
##  7 FALSE      TRUE       FALSE      FALSE      FALSE
##  8 FALSE      FALSE      FALSE      FALSE      FALSE
##  9 TRUE       TRUE       TRUE       TRUE       TRUE
## 10 FALSE      TRUE       TRUE       TRUE       TRUE
## # ℹ 20 more rows
```

dplyr package

The `map_dfr()` and `map_dfc()` functions actually belong to the `dplyr` package instead of `purrr`. `dplyr` is a member of the `tidyverse` package that is very useful for data frame computations. You'll learn more soon, but for now, you can think of it as providing the `tidyverse` alternative to the base R functions `subset()`, `split()`, `tapply()`

`filter()`: subset rows based on a condition

```
head(mtcars) # Built in data frame of cars data, 32 cars x 1
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs a
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1
```

```
filter(mtcars, (mpg >= 20 & disp >= 200) | (drat <= 3))
```

```
##                     mpg cyl disp  hp drat    wt  qsec vs
## Hornet 4 Drive      21.4   6  258 110 3.08 3.215 19.44  1
## Valiant             18.1   6  225 105 2.76 3.460 20.22  1
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0
```

```
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0
## Dodge Challenger    15.5   8  318 150 2.76 3.520 16.87  0
```

```
# Base R is just as easy with subset(), more complicated wit
subset(mtcars, (mpg >= 20 & disp >= 200) | (drat <= 3))
```

```
##                      mpg cyl disp  hp drat    wt  qsec vs
## Hornet 4 Drive      21.4   6  258 110 3.08 3.215 19.44  1
## Valiant             18.1   6  225 105 2.76 3.460 20.22  1
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0
## Dodge Challenger    15.5   8  318 150 2.76 3.520 16.87  0
```

```
mtcars[(mtcars$mpg >= 20 & mtcars$disp >= 200) | (mtcars$dra
```

```
##                      mpg cyl disp  hp drat    wt  qsec vs
## Hornet 4 Drive      21.4   6  258 110 3.08 3.215 19.44  1
## Valiant             18.1   6  225 105 2.76 3.460 20.22  1
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0
## Dodge Challenger    15.5   8  318 150 2.76 3.520 16.87  0
```

`group_by()` : define groups of rows based on columns or conditions

```
group_by(mtcars, cyl)
```

```
## # A tibble: 32 × 11
## # Groups:   cyl [3]
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160    110  3.9   2.62  16.5     0     1
##  2  21       6  160    110  3.9   2.88  17.0     0     1
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0
```

```
##  8  24.4     4  147.    62  3.69  3.19  20       1     0
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0
## # ℹ 22 more rows
```

- This doesn't actually change anything about the way the data frame looks
- Only difference is that when it prints, we're told about the groups
- But it will play a big role in how `dplyr` functions act on the data frame

`summarize()`: apply computations to (groups of) rows of a data frame

```
# Ungrouped
summarize(mtcars, mpg = mean(mpg), hp = mean(hp))
```

```
##        mpg       hp
## 1 20.09062 146.6875
```

```
# Grouped by number of cylinders
summarize(group_by(mtcars, cyl), mpg = mean(mpg), hp = mean(
```

```
## # A tibble: 3 × 3
##     cyl   mpg    hp
##   <dbl> <dbl> <dbl>
## 1     4  26.7  82.6
## 2     6  19.7 122.
## 3     8  15.1 209.
```

Note: the use of `group_by()` makes the difference here

```
# Base R, ungrouped calculation is not so bad
c("mpg" = mean(mtcars$mpg), "hp" = mean(mtcars$hp))
```

```
##       mpg        hp
##  20.09062 146.68750
```

```r
# Base R, grouped calculation is getting a bit ugly
cbind(tapply(mtcars$mpg, INDEX=mtcars$cyl, FUN=mean),
      tapply(mtcars$hp, INDEX=mtcars$cyl, FUN=mean))
```

```
##       [,1]      [,2]
## 4 26.66364  82.63636
## 6 19.74286 122.28571
## 8 15.10000 209.21429
```

```r
sapply(split(mtcars, mtcars$cyl), FUN=function(df) {
  return(c("mpg" = mean(df$mpg), "hp" = mean(df$hp)))
})
```

```
##            4         6        8
## mpg 26.66364  19.74286  15.1000
## hp  82.63636 122.28571 209.2143
```

```r
aggregate(mtcars[, c("mpg", "hp")], by=list(mtcars$cyl), mea
```

```
##   Group.1      mpg        hp
## 1       4 26.66364  82.63636
## 2       6 19.74286 122.28571
## 3       8 15.10000 209.21429
```

Summary

- For iteration tasks, `plyr` and `tidyverse` are two alternative of base R with better consistency
- `plyr` provides apply-like functions `**ply` and ensures the consistency of input and output data types
- `tidyverse` is a collection of packages for common data science tasks
- `purrr` is one such package that provides a consistent family of iteration functions
- Compared with `plyr`, `purrr` is often faster
- `map()` : list in, list out

- `map_dbl()`, `map_lgl()`, `map_chr()`: list in, vector out (of a particular data type)
- `map_dfr()`, `map_dfc()`: list in, data frame out (row-binded or column-binded)
- `dplyr` is another such package that provides functions for data frame computations
- `filter()`: subset rows based on a condition
- `group_by()`: define groups of rows according to a condition
- `summarize()`: apply computations across groups of rows