CIS 8392
Topics in Big Data Analytics

#Data Programming

Yu-Kai Lin

Agenda

- Conditions: if-else, ifelse, and switch
- Iterations: while-loop and for-loop
- Functions
- Functional programming
- Parallel computing

[**Acknowledgements**] The materials in the following slides are based on the source(s) below:

- R for Data Science by Garrett Grolemund and Hadley Wickham
- Advanced R by Hadley Wickham
- Efficient R Programming by Colin Gillespie and Robin Lovelace
- The R Inferno by Patrick Burns
- Quick Intro to Parallel Computing in R by Matt Jones
- How-to go parallel in R – basics + tips by Max Gordon

Prerequisites

- repurrrsive: examples of recursive lists and nested or split data frames
- purrr: functional programming toolkit for R (included in `tidyverse`)
- doParallel and foreach: toolkits for parallel computing

```
install.packages(c("purrr","repurrrsive", "doParallel", "foreach"))
library(repurrrsive)
library(tidyverse)

# See the data included in the repurrrsive package
data(package = "repurrrsive")$result[, c("Item","Title")]
```

```
##       Item           Title
##  [1,] "discog"       "Sharla Gelfand's music collection"
##  [2,] "gap_nested"   "Gapminder data frame in various forms"
##  [3,] "gap_simple"   "Gapminder data frame in various forms"
##  [4,] "gap_split"    "Gapminder data frame in various forms"
##  [5,] "gh_repos"     "GitHub repos"
##  [6,] "gh_users"     "GitHub users"
##  [7,] "got_chars"    "Game of Thrones POV characters"
##  [8,] "sw_films"     "Entities from the Star Wars Universe"
##  [9,] "sw_people"    "Entities from the Star Wars Universe"
## [10,] "sw_planets"   "Entities from the Star Wars Universe"
## [11,] "sw_species"   "Entities from the Star Wars Universe"
## [12,] "sw_starships" "Entities from the Star Wars Universe"
```

if()...else...

```
x = 10
if(x == 10) {
  print("x equals 10")
}
```

```
## [1] "x equals 10"
```

```
if(x > 20) {
  print("x is greater than 20")
} else {
  print("x is equal to or less than 20")
}
```

```
## [1] "x is equal to or less than 20"
```

```
if(x > 20) {
  print("x is greater than 20")
} else if(x >= 10) {
  print("x is equal to or greater than 10")
} else {
  print("x is less than 10")
}
```

ifelse()

`ifelse` is a handy function that works like if()...else...

Template:

```
ifelse(<CONDITION>, <do_this_if_true>, <do_this_if_false>)
```

Example:

```
a <- 2; b <- 1
ifelse(a > b, "a is greater than b", "a is less than b")
```

```
## [1] "a is greater than b"
```

```
color <- ifelse(a==b, "red", "blue")
print(color)
```

```
## [1] "blue"
```

while-loop

The `while` loop continually executes a block of statements while a particular condition is true.

Template:

```
while(<CONDITION>) {
  #DO SOMETHING HERE
}
```

Example:

```
x = 1
while(x <= 3) {
  print(x)
  x = x + 1 # what would heppen if you skip this line?
}
```

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

for-loop

A `for` loop iterates through every element in a vector.

Template:

```
for (<ELEMENT> in <VECTOR>){
  #Use the element to do something
}
```

Example:

```
for (year in 2001:2004){
  print( paste("The year was", year) )
}
```

```
## [1] "The year was 2001"
## [1] "The year was 2002"
## [1] "The year was 2003"
## [1] "The year was 2004"
```

In words, for each `year` that is in the sequence `2001:2004`, you execute the code chunk `print( paste("The year was", year) )`

Tips when working with for-loops

1.If you want to skip some items in a for-loop, use the keyword `next`.

```
# Example
for (year in 2001:2004){
  if(year==2002) next
  print( paste("The year was", year) )
}
```

```
## [1] "The year was 2001"
## [1] "The year was 2003"
## [1] "The year was 2004"
```

Tips when working with for-loops

2.Never use a for-loop to grow a vector. If necessary, initiate the vector with an appropriate size outside the loop.

DON'T DO THIS:

```
n = 1e8 #see difference with a large n
vec1 = c() #empty vector
for (i in 1:n) vec1[i] <- i^2 #growing
```

BETTER CHOICE:

```
vec2 = vector("numeric", n)
for (i in 1:n) vec2[i] <- i^2
```

BEST CHOICE:

```
vec3 = (1:n)^2 # vectorized operation
```

```
all.equal(vec1, vec2, vec3)
```

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

```
system.time({
  vec1 = c()
  for (i in 1:n) vec1[i] <- i^2
})
```

```
##    user  system elapsed
##   22.66    3.18   26.07
```

```
system.time({
  vec2 = vector("numeric", n)
  for (i in 1:n) vec2[i] <- i^2
})
```

```
##    user  system elapsed
##    4.77    0.11    4.92
```

```
system.time({ vec3 = (1:n)^2 })
```

```
##    user  system elapsed
##    0.27    0.06    0.33
```

Your turn

Calculate the average of each column in `iris`, and save them to a vector. Intuitively, non-numeric column(s) should have `NA` as the average.

```
head(iris, 5)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
```

Functions

A function is a procedure or routine which takes optional inputs and produces an optional output.

Why functions?

- Data structures tie related values into one object
- Functions tie related commands into one object
- In both cases: easier to understand, easier to work with, easier to build into larger things

We have been using many built-in functions:

- Vector
  - `seq()`, `rep()`, `mean()`, `length()`, ...
- Data frame
  - `colnames()`, `rownames()`
- Character string
  - `paste()`

Creating new functions

Template:

```
<FUNCTION_NAME> <- function(<INPUT>, ...) {
  #DO SOMETHING
  return(<SOME_VALUE>)
}
```

Example:

```
hello_world <- function() { # this particular function takes no input
  print("Hello world!")
  # This function has no return statement; return nothing.
}

hello_world()  # call the function; don't forget the parentheses
```

```
## [1] "Hello world!"
```

Another example:

```
add_one <- function(num) {
  num <- num + 1  # be sure to match the input variable name
  return(num)  # return() says what the output is
}
a = add_one(10)
print(a)
```

```
## [1] 11
```

Q: What would happen if you skip `return(num)` in the `add_one` function.

> Ans: The value of the last line will be returned automatically

Q: What would happen if you run `add_one("big data")`?

> Ans: You get the error meesage:
> **Error in num + 1 : non-numeric argument to binary operator**

Q: How to make the function more robust to address issues such as `add_one("big data")`?

try and tryCatch

What happens when something goes wrong with your R code? What do you do? What tools do you have to address the problem?

- Ignore errors with `try()`

```
result = NULL
try({result = 1 + "gsu"})
```

```
## Error in 1 + "gsu" : non-numeric argument to binary operator
```

```
try({result = 1 + "gsu"}, silent = T)
result
```

```
## NULL
```

- Handle conditions with `tryCatch()`

```
out <- tryCatch({
      # code
  }, error=function(cond) {
      # What to do or return if there is an error
  }, warning=function(cond) {  # This is optional...
      # What to do or return if there is a warning
  }, finally={  # This is optional...
      # Here goes everything that should be executed at the end,
      # regardless of success or error.
  }
)
```

Your turn

Try to address the issues in `add_one("big data")` by using `tryCatch()` inside the `add_one` function

More than one input to a function

```
# Notice that the input for university_name has a default value!
greeting <- function(your_name, university_name="GSU") {
  print(paste0("Hello, ", your_name, ". Welcome to ", university_name, "!"))
}

greeting(your_name="Alice")
```

```
## [1] "Hello, Alice. Welcome to GSU!"
```

```
greeting(your_name="Bob", university_name="UGA")
```

```
## [1] "Hello, Bob. Welcome to UGA!"
```

```
greeting("Claire")
```

```
## [1] "Hello, Claire. Welcome to GSU!"
```

```
greeting("David", "Emory")
```

```
## [1] "Hello, David. Welcome to Emory!"
```

Your turn

Develop a function `add_two_values()` which allows 0, 1, or 2 input values.

- When there is not a input value, return 0
- When there is only one input value, return that value
- When there are two input values, return the sum of these two values
- Ensure that the function is able to handle incorrect data types (non-numeric) for input values

Functional Programming

R, at its heart, is a functional programming language.

This means that it provides many tools for the creation and manipulation of functions.

You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

`purrr` is a package included in `tidyverse`, designed to enhance R's functional programming capability. (Cheatsheet for how to use purrr)

map()

`map()` is a function for applying a function to each element of a list.

Template:

```
map(YOUR_LIST, YOUR_FUNCTION)
```

Example:

```
map(c(3,4,5), sqrt)
```

```
## [[1]]
## [1] 1.732051
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 2.236068
```

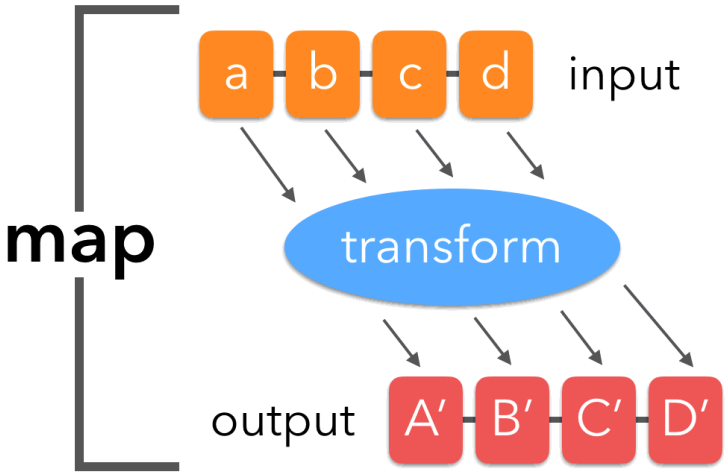


Image source: https://swiftunboxed.com/open-source/map/

Type-specific map

`map()` always returns a list, even if all the elements have the same flavor and are of length one. But in that case, you might prefer a simpler object: **an atomic vector**.

`map_*()` allows you to simplify and specify the type of output.

- `map_lgl()` makes a logical vector.
- `map_int()` makes an integer vector.
- `map_dbl()` makes a double vector.
- `map_chr()` makes a character vector.

Each function takes a vector as input, applies a function to each piece, and then returns a new vector that's the same length (and has the same names) as the input. The type of the vector is determined by the suffix to the map function.

Examples:

```
map_dbl(c(3,4,5), sqrt)
```

```
## [1] 1.732051 2.000000 2.236068
```

Use an anonymous function with map

An anonymous function, as you expect, does not have a name.

The function is created at where it will be used, and disappears after that:

```
map_dbl(1:3, function(x) {x^2})
```

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

Your turn

1. Use `map` and the `repurrrsive::sw_people` data to calculate the BMI for all the characters in the the Star Wars Universe.
    - The BMI formula is `BMI = kg/m2`, where `kg` is a person's weight in kilograms and `m2` is their height in metres squared.
2. Based on the above result, create a data frame with the following columns:
    - name
    - gender
    - height
    - mass
    - bmi
    - bmi_category: see this reference

Use a formula within `map`

You can use a **formula**, e.g. `~ .x + 2`, to express a function. purrr will convert the formula to a function. This syntax allows you to create very compact anonymous functions.

There are three ways to refer to the arguments:

- For a single argument function, use `.`
- For a two argument function, use `.x` and `.y`
- For more arguments, use `..1`, `..2`, `..3` etc

If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function.

```
map_dbl(1:3, ~ .^2) # it genertes an anonymous function with one argument
```

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

```
map2_dbl(c(1,3,5), c(2,4,6), ~ .x + .y) # two arguments. More on this later
```

```
## [1]  3  7 11
```

A second look at the previous exercises

Get the number of followers for each user in `gh_users`:

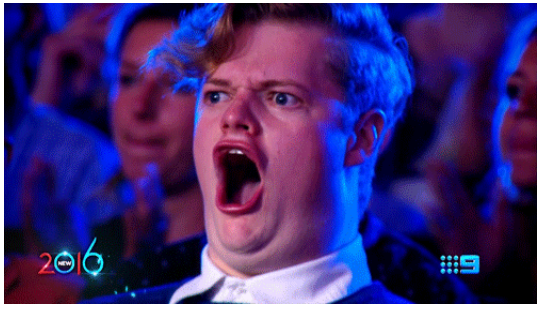
```
followers = map_dbl(gh_users, ~.$followers)
followers
```

```
## [1]  303  780 3958  115  213   34
```

Calculate the average of each column in `iris`:

```
map_dbl(iris, mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##     5.843333     3.057333     3.758000     1.199333           NA
```



Use pipes with `map`

We have learned that you can use pipes (`%>%`) to manipulate a data frame.

You can actually use pipes to streamline any functions.

The following three are equivalent:

```
followers = map_int(gh_users, ~.$followers)
```

```
followers = gh_users %>%
  map_int(., ~.$followers) # . (dot) is whatever from before the pipe
```

```
followers = gh_users %>%
  map_int(~.$followers) # safe to skip . (dot) if it is the first argument
```

Your turn

Use `map` to find out the time difference in days between `created_at` and `updated_at` for each of the 6 individual in the `gh_users` data.

Hints:

- How to get the two pieces of information (`created_at` and `updated_at`)
- What are their data types?
- How to get the time difference in two days?
- How to put all these togeter into `map()`?

The output would look like this:

```
## [[1]]
## Time difference of 2043 days
##
## [[2]]
## Time difference of 2090 days
##
## [[3]]
## Time difference of 1656 days
##
## [[4]]
## Time difference of 520 days
##
## [[5]]
## Time difference of 1351 days
##
## [[6]]
## Time difference of 811 days
```

Mapping over multiple arguments

Template:

```
map(YOUR_LIST, YOUR_FUNCTION, OTHER_INPUTS_FOR_YOUR_FUNCTION)
```

Example:

```
mu <- c(5, 10, -3)
map(mu, rnorm, n = 5) # rnorm() generates numbers from a normal distribution
                      # n indicates how many numbers to generate
                      # type ?rnorm to learn more about this function
```

```
## [[1]]
## [1] 4.479277 5.167937 7.256878 4.765528 3.650872
##
## [[2]]
## [1]  9.366155 10.316405 11.707057  8.721389 11.954135
##
## [[3]]
## [1] -3.624545 -1.129436 -2.743722 -1.892758 -3.009386
```

Another example:

```
map_df(
  gh_users, # for each element in the list
  magrittr::extract, # run the extract function in the magrittr package
  c("name", "followers", "following", "public_repos") # values to extract()
)
```

```
## # A tibble: 6 x 4
##   name                  followers following public_repos
##   <chr>                     <int>     <int>        <int>
## 1 Gábor Csárdi                303        22           52
## 2 Jennifer (Jenny) Bryan      780        34          168
## 3 Jeff L.                    3958         6           67
## 4 Julia Silge                 115        10           26
## 5 Thomas J. Leeper            213       230           99
## 6 Maëlle Salmon                34        38           31
```

Map over multiple inputs in parallel

These functions below are variants of `map()` that iterate over multiple arguments simultaneously. They are parallel in the sense that each input is processed in parallel with the others, not in the sense of multicore computing (which we will see later in just a moment).

- `map2(.x, .y, .f, ...)`
- `map2_lgl(.x, .y, .f, ...)`
- `map2_int(.x, .y, .f, ...)`
- `map2_dbl(.x, .y, .f, ...)`
- `map2_chr(.x, .y, .f, ...)`
- `pmap(.l, .f, ...)`

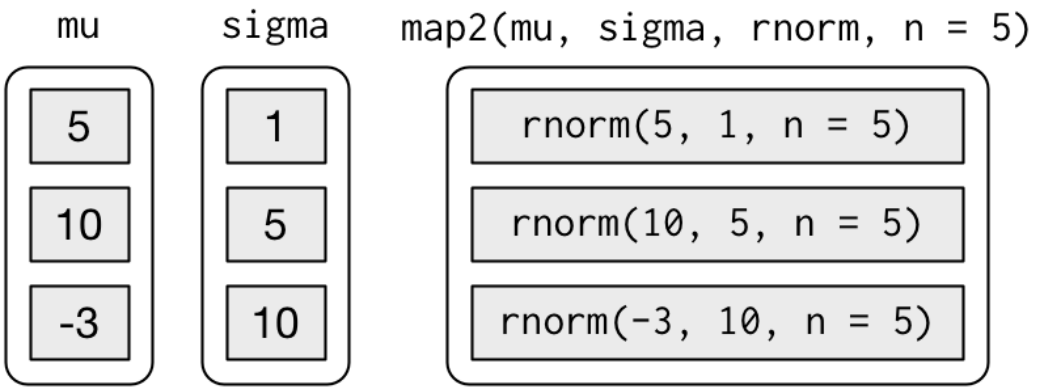
Arguments

- **.x**, **.y** : Vectors of the same length. A vector of length 1 will be recycled.
- **.f** : A function, formula, or vector (not necessarily atomic).
- **.l** : A list of vectors. Suitable for any arbitrary number of input vectors

Example

```
mu <- list(5, 10, -3)
sigma <- list(1, 5, 10)
map2(mu, sigma, rnorm, n = 5)
```

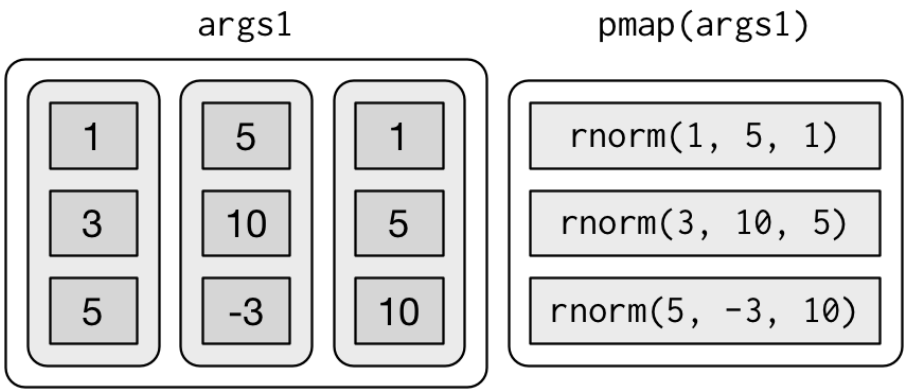
```
## [[1]]
## [1] 5.633145 4.527765 5.142791 5.198774 4.476661
##
## [[2]]
## [1]  6.143119  8.989683  8.105521 14.314140  5.286513
##
## [[3]]
## [1]  -4.956636 -15.281026   1.496675 -15.069801   5.208635
```



Example

```
mu <- list(5, 10, -3)
sigma <- list(1, 5, 10)
n <- list(1, 3, 5)
args1 <- list(n, mu, sigma) #type ?rnorm to find out why we have this order
pmap(args1, rnorm)
```

```
## [[1]]
## [1] 4.611141
##
## [[2]]
## [1] 9.242407 8.606326 8.271210
##
## [[3]]
## [1] 14.1714758  5.3059607  2.0443642 -6.9439583 -0.2986518
```



Walk

Walk is an alternative to map that you use when you want to call a function for its side effects, rather than for its return value. You typically do this because you want to render output to the screen or save files to disk. *The important thing is the action, not the returned value.*

Here's a very simple example:

```
x <- list(1, "a", 3)

x %>%
  walk(print)
```

```
## [1] 1
## [1] "a"
## [1] 3
```

`walk()` is generally not that useful compared to `walk2()` or `pwalk()`. For example, if you had a list of plots and a vector of file names, you could use pwalk() to save each file to the corresponding location on disk:

```
library(ggplot2)
plots <- mtcars %>%
  split(.$cyl) %>% # for each unique cyl value, make a plot
  map(~ggplot(., aes(mpg, wt)) + geom_point())
paths <- paste0(names(plots), ".png")

pwalk(list(paths, plots), ggsave) #save to the working directory
```

Here we use a few functions that worth mentioning:

- `split`: `split()` divides the data in the vector x into the groups
- `ggsave`: `ggsave()` is a convenient function for saving a plot. It defaults to saving the last plot that you displayed, using the size of the current graphics device. It also guesses the type of graphics device from the extension.

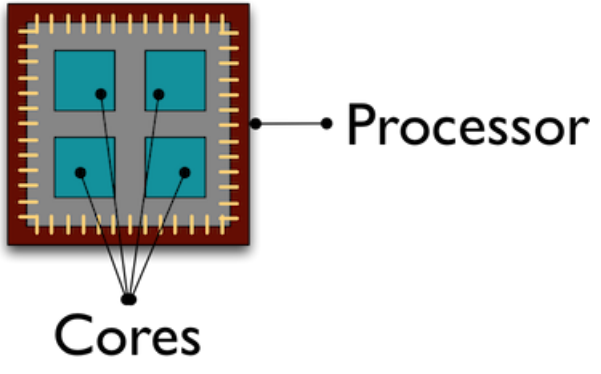
Parallelization

Why parallelization?

Much R code runs fast and fine on a single processor when the data is small. However, when dealing with big data, computations can be:

- **cpu-bound**: Take too much cpu time
- **memory-bound**: Take too much memory
- **I/O-bound**: Take too much time to read/write from disk
- **network-bound**: Take too much time to transfer

To help with **cpu-bound** computations, one can take advantage of modern processor architectures that provide multiple cores on a single processor.



How many cores in your computer?

```
library(foreach)
library(doParallel)
parallel::detectCores()
```

```
## [1] 8
```

```r
library(foreach)
library(doParallel)
base <- 2
n_core = parallel::detectCores()
registerDoParallel(n_core) #initiate a parallel cluster

# .combine specify how the results should be combined
# .combine = c says that the results should be combined into a vector
foreach(exponent = 2:4, .combine = c) %dopar% {
  base^exponent
}
```

```
## [1]  4  8 16
```

```r
# .combine = rbind says that the results should be combined into a data.frame
foreach(exponent = 2:4, .combine = rbind) %dopar% {
  base^exponent
}
```

```
##          [,1]
## result.1    4
## result.2    8
## result.3   16
```

```r
stopImplicitCluster() #remember to stop the cluster once finished
```

Variable scope in `foreach()`

```
registerDoParallel(n_core)

base <- 2
test <- function (exponent) {
  foreach(exponent = 2:4,
          .combine = c)  %dopar%
    base^exponent
}
test()

stopImplicitCluster()
```

Error in base^exponent : task 1 failed - "object 'base' not found"

```
registerDoParallel(n_core)

base <- 2
test <- function (exponent) {
  foreach(exponent = 2:4,
          .combine = c,
          .export = "base")  %dopar%
    base^exponent
}
test()
```

```
## [1]  4  8 16
```

```
stopImplicitCluster()
```

Is it indeed running in parallel?

Let's use the `mtcars` data set to do a parallel bootstrap of 10,000 iterations. In each iteration, we do the following:

1. randomly sample 100 observations with replacement from `mtcars`
2. save the sample to `mtcars_sample`
3. use `mtcars_sample` to run a linear regression `mpg ~ hp + cyl + wt`

In serial:

```
trials <- 10000
system.time({
  r <- foreach(i=1:trials) %do% {
    ind <- sample(nrow(mtcars), 100, T)
    mtcars_sample = mtcars[ind, ]
    result1 <- lm(mpg ~ hp + cyl + wt,
                  data = mtcars_sample)
    coefficients(result1)
  }
})
```

```
##    user  system elapsed
##   10.28    0.00   10.33
```

In parallel:

```
trials=10000;
registerDoParallel(cores=4)
system.time({
  r <- foreach(i=1:trials) %dopar% {
    ind <- sample(nrow(mtcars), 100, T)
    mtcars_sample = mtcars[ind, ]
    result1 <- lm(mpg ~ hp + cyl + wt,
                  data = mtcars_sample)
    coefficients(result1)
  }
})
```

```
##    user  system elapsed
##    2.29    0.56    4.82
```