

Functional Programming

John Tipton

Readings

- R for data science
 - Introduction
 - Chapters 14 (Pipes with `magrittr`), 15 (Functions), 16 (Vectors), and 17 (Iteration with `purrr`)

Pipes

```
library(tidyverse)
```

- Pipes `%>%` are from the `magrittr` package (part of the `tidyverse`)
- Pipes focus on making functions act like verbs

Pipes

The itsy bitsy spider went up the water spout
Down came the rain and washed the spider out
Out came the sun and dried up all the rain
And the itsy bitsy spider went up the spout again

- multiple variable names, likely to make a typo

```
itsy_bitsy_spider_1 <- went(itsy_bitsy_spider, up = "waterspout")  
itsy_bitsy_spider_2 <- rain_down(itsy_bitsy_spider_1, washed = "out")  
itsy_bitsy_spider_3 <- sun(itsy_bitsy_spider_2, dried = "rain")  
itsy_bitsy_spider_4 <- went(itsy_bitsy_spider_3, up = "spout again")
```

- overwriting the data, likely to make errors

```
itsy_bitsy_spider <- went(itsy_bitsy_spider, up = "waterspout")  
itsy_bitsy_spider <- rain_down(itsy_bitsy_spider, washed = "out")  
itsy_bitsy_spider <- sun(itsy_bitsy_spider, dried = "rain")  
itsy_bitsy_spider <- went(itsy_bitsy_spider, up = "spout again")
```

Pipes

The itsy bitsy spider went up the water spout
Down came the rain and washed the spider out
Out came the sun and dried up all the rain
And the itsy bitsy spider went up the spout again

- hard to read

```
went(  
  sun(  
    rain_down(  
      went(itsy_bitsy_spider, up = "waterspout"),  
      washed = "out"),  
    dried = "rain"),  
  up = "spout again"  
)
```

- Much better

```
itsy_bitsy_spider %>%  
  went(up = "waterspout") %>%  
  rain_down(washed = "out") %>%  
  sun(dried = "rain") %>%  
  went(up = "spout again")
```

Pipes

- How does the pipe work?

```
x <- 1:10  
x %>%  
  mean()
```

```
## [1] 5.5
```

is equivalent to

```
x %>%  
  mean(.)
```

```
## [1] 5.5
```

where the `.` is a placeholder for the variable `x` which results in

```
mean(x)
```

```
## [1] 5.5
```

Pipes

- Pipes are not always the optimal choice
- Try to avoid super long chains of pipes
 - Instead: break up the pipe into intermediate objects
- Pipes can't handle multiple inputs/outputs
- How to build a chain of pipes
 - Start by outlining a plan of attack
 - Write one small subset of the pipe chain at a time and verify the results
- Challenge: Ask me questions about the `diamonds` data

Functions

- Remember, writing code is for humans, not computers
- Make code readable and descriptive
- Reduce errors and typos
- Easily update/fix all examples

Keep it DRY

- Don't Repeat Yourself
- If you copy and paste code 3 or more times, write a function
- Generate some data

```
df <- data.frame(  
  x = rnorm(10),  
  y = rnorm(10),  
  z = rnorm(10)  
)
```

- The above isn't DRY, but there is no chance of errors and the code is easy to understand

DRY

- Transform each variable to mean 0, standard deviation 1

```
x_scaled <- (df$x - mean(df$x, na.rm = TRUE)) / sd(df$x, na.rm = TRUE)
y_scaled <- (df$y - mean(df$y, na.rm = TRUE)) / sd(df$y, na.rm = TRUE)
z_scaled <- (df$z - mean(df$z, na.rm = TRUE)) / sd(df$y, na.rm = TRUE)
```

- Can you spot the error?

```
z_scaled <- (df$z - mean(df$z, na.rm = TRUE)) / sd(df$y, na.rm = TRUE)
```

- write a function to do this

```
rescale_variable <- function(x) {
  mean_x <- mean(x, na.rm = TRUE)
  sd_x    <- sd(x, na.rm = TRUE)
  ## save the variable for the output
  out <- (x - mean_x) / sd_x
  return(out)
}
```

- Much less likely to make an error

```
x_scaled <- rescale_variable(df$x)
y_scaled <- rescale_variable(df$y)
z_scaled <- rescale_variable(df$z)
```

Naming Functions

- Name functions for human readers!

```
# bad name -- what does this do?  
f()  
  
# what does this do?  
func()  
  
# longer names -- use autocomplete  
fill_missing_values()  
pre_process_data()
```

- Don't name your functions over functions or variables which already exist!

```
mean <- function(x) {}  
TRUE <- function(x, y) {}
```

Naming Functions

- Pick a naming convention and stick with it (I prefer snake case)

```
# snake case  
my_function_name <- function(x, y) {}  
# camel case  
myFunctionName <- function(x, y) {}
```

- Try to reuse leading names -- allows for autocomplete

```
# better  
check_input()  
check_parameters()  
check_inits()  
  
# not as good  
input_check()  
parameters_check()  
inits_check()
```

Conditional statements

- `if()` and `if() {} else {}` statements

```
if (condition) {  
  # do something if condition is TRUE  
} else {  
  # do something if condition is FALSE  
}
```

- Conditions must be either `TRUE` or `FALSE`

```
if (NA) {}
```

```
## Error in if (NA) {: missing value where TRUE/FALSE needed
```

Conditional statements

- Example: write a function to compare a number

```
compare_number <- function(x, y) {  
  ## compare the input number x to the number y  
  if (x < y) {  
    # default behavior is return  
    str_c(x, "is less than", y, sep = " ")  
  } else if (x > y) {  
    # default behavior is return  
    str_c(x, "is less greater", y, sep = " ")  
  } else {  
    # default behavior is return  
    str_c(x, "and", y, "are equal", sep = " ")  
  }  
}
```

```
compare_number(3, 4)
```

```
## [1] "3 is less than 4"
```

```
compare_number(5, 5)
```

```
## [1] "5 and 5 are equal"
```

Logical comparisons

- Use `||` for or

```
TRUE || FALSE
```

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

- Use `&&` for and

```
TRUE && FALSE
```

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

- Note: `|` and `&` are vectorized, `||` and `&&` are not

```
c(TRUE, TRUE, FALSE) || c(TRUE, FALSE, FA
```

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

```
c(TRUE, TRUE, FALSE) | c(TRUE, FALSE, FAL
```

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

```
c(TRUE, TRUE, FALSE) && c(TRUE, FALSE, FA
```

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

```
c(TRUE, TRUE, FALSE) & c(TRUE, FALSE, FAL
```

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

Logical comparisons

- `any()` and `all()`

```
any(c(TRUE, FALSE, FALSE))
```

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

```
any(c(FALSE, FALSE, FALSE))
```

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

```
all(c(TRUE, FALSE))
```

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

```
all(c(TRUE, TRUE))
```

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


Logical comparisons

- `identical()`, `all.equal()`, and `dplyr::near()`
- rounding (floating point math) can cause issues

```
identical(2L, 2)
```

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

```
all.equal(2L, 2)
```

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

```
near(2L, 2)
```

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

```
identical(sqrt(2)^2, 2)
```

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

```
all.equal(sqrt(2)^2, 2)
```

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

```
near(sqrt(2)^2, 2)
```

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

```
sqrt(2)^2 - 2
```

```
## [1] 4.440892098500626161695e-16
```

Multiple conditions

- `if () {} else if () {} else {}`
- `switch()`

```
# evaluate based on position  
switch(1, "red", "green", "blue")
```

```
## [1] "red"
```

```
switch(3, "red", "green", "blue")
```

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

Multiple conditions

- switch

```
# evaluate based on match
switch("christmas",
  "christmas" = c("red", "green"),
  "st. patricks day" = "green",
  "kwanzaa" = c("black", "red", "gre
  "valentines" = "red",
  "easter" = "pastels",
  stop("Unknown")
)
```

```
## [1] "red" "green"
```

```
switch("st. patricks",
  "christmas" = c("red", "green"),
  "st. patricks" = "green",
  "kwanzaa" = c("black", "red", "gre
  "valentines" = "red",
  "easter" = "pastels",
  stop("Unknown")
)
```

```
## [1] "green"
```

```
# evaluate based on match
switch("kwanzaa",
  "christmas" = c("red", "green"),
  "st. patricks day" = "green",
  "kwanzaa" = c("black", "red", "gre
  "valentines" = "red",
  "easter" = "pastels",
  stop("Unknown")
)
```

```
## [1] "black" "red" "green"
```

```
switch("fourth of july",
  "christmas" = c("red", "green"),
  "st. patricks" = "green",
  "kwanzaa" = c("black", "red", "gre
  "valentines" = "red",
  "easter" = "pastels",
  stop("Unknown Holiday")
)
```

```
## Error in eval(expr, envir, enclos): Unknown Holiday
```

code style

- See [Google's R coding style guide](#) and the [Tidyverse R coding style](#)
- There is also a [Google python code style guide](#)
- Develop a consistent style and stick with it
- Coding style will make your code more readable

Function arguments

- Typically, functions will have two components
 - the data (first arguments)
 - the options for the function (later arguments, often given default values)
- `mean()`
 - `x` is the data
 - `trim` is the number of smallest/largest values to drop
 - `na.rm` is what to do with missing values

```
?mean
```

- Make the default arguments the most commonly used
 - Exception: force the user to omit missing values

Function arguments

- Calculate a 95% confidence interval for the mean

```
ci_mean <- function(x, conf_level = 0.95, na.rm = FALSE) {  
  ## calculate CI for vector of data x  
  n <- length(x)  
  if (na.rm == TRUE) {  
    n <- length(na.omit(x))  
  }  
  xbar <- mean(x, na.rm = na.rm)  
  s <- sd(x, na.rm = na.rm)  
  alpha <- 1 - conf_level  
  t_crit <- qt(c(alpha/2, 1 - alpha/2), n-1)  
  return(xbar + t_crit * s / sqrt(n))  
}  
x <- rnorm(100, 5, 2)
```

```
ci_mean(x)
```

```
## [1] 4.665667076301064142285 5.4140586356546283041234.695185279708751657779 5.4423397814483136514
```

```
x[2] <- NA  
ci_mean(x)
```

```
## [1] NA NA
```

```
ci_mean(x, na.rm = TRUE)
```

```
ci_mean(x, conf_level = 0.8, na.rm = TRUE)
```

```
## [1] 4.825872171789698406030 5.311652889367366903
```

Calling functions

- Call the arguments in order

```
# good or bad?  
mean(trim = 2, x = 1:10)
```

```
## [1] 5.5
```

```
# better  
mean(x = 1:10, trim = 2)
```

```
## [1] 5.5
```

```
mean(1:10, trim = 2)
```

```
## [1] 5.5
```

- Use the full name of any default arguments that you change
- Any non-standard calls should use the full name of the arguments

```
# confusing argument -- use the full names  
mean(1:10, tr = 2)
```

Naming arguments

There are many times where a name is commonly used by convention

- `x`, `y`, `z`: vectors.
- `w`: a vector of weights.
- `df` or `dat`: a data frame.
- `i`, `j`: numeric indices (typically rows and columns).
- `n`: length, or number of rows.
- `p`: number of columns.

Checking arguments

- `stop()` interrupts the function and returns an **error**
 - red light
- `warning()` allows the function to continue and returns a **warning**
 - yellow light
- `message()` allows for output to be printed
 - street sign
 - some use alternatives like `cat()` and `print()`. **Don't use these** as they are harder to disable when not wanted
 - Rmarkdown code chunks have the option `message = FALSE` to suppress `messages()` but this doesn't work for `cat()` and `print()`

Checking arguments

```
add_matrices <- function(A, B) {  
  # can only add matrices if they are the same size  
  if (dim(A) != dim(B) || !is.matrix(A) || !is.matrix(B)) {  
    stop("A and B must be matrices of the same dimension")  
  }  
  return(A + B)  
}  
  
A <- matrix(1:6, 3, 2)  
b <- 7:12
```

```
add_matrices(A, b)
```

```
## Error in add_matrices(A, b): A and B must be matrices of the same dimension
```

```
B <- matrix(7:12, 2, 3)  
add_matrices(A, B)
```

```
## Error in add_matrices(A, B): A and B must be matrices of the same dimension
```

```
B <- matrix(7:12, 3, 2)  
add_matrices(A, B)
```

```
##      [,1] [,2]  
## [1,]    8  14  
## [2,]   10  16  
## [3,]   12  18
```

Error checking

- The amount of error checking depends on the use
 1. Is this a function for personal use only?
 2. Is your function going to be used by others?
 3. Is your function going to be used in production?
- For 1), you don't need much checking. For 2) and 3), you need robust error checks
- What might you want to check?
 - Data/variable dimensions and sizes
 - Variable types (numeric, character, logical)
 - Missing values
 - Correct arguments

```
B <- matrix(NA, 3, 2)
add_matrices(A, B)
```

```
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
## [3,]   NA   NA
```

```
B <- matrix("a", 3, 2)
add_matrices(A, B)
```

```
## Error in A + B: non-numeric argument to binary operator
```

Error checking

- Formal error checking is called "unit testing" in software engineering
- Unit testing involves writing automated tests for functions to verify their behavior and their respective error messages
- Break functions into small "units" that can be easily tested
- Test driven development
 - Write the tests for how your function will fail before you write the function
 - Make the function pass the test using error checking (`stop()`)
 - Then, write the actual function and verify the expected output for given input

The ... function argument

- ... is a placeholder for other arguments!



The ... function argument

- ... is a placeholder for other arguments!
 - python uses `*args` and `*kwargs`

```
#?sum  
sum(1, 2, 3)
```

```
## [1] 6
```

```
sum(6, 5, 4, 3, 2)
```

```
## [1] 20
```

- Super flexible, but can be complicated
- ... also makes functions prone to spelling errors

```
sum(2, 3, 4, NA, na.rm = TRUE)
```

```
## [1] NA
```

Function returns

- Default behavior is to return the last line executed in the function
- I prefer to always explicitly use `return()` as it is easier to read

```
add_two_numbers <- function(a, b) {  
  a + b  
}  
add_two_numbers(5, 7)
```

```
## [1] 12
```

```
add_two_numbers <- function(a, b) {  
  return(a + b)  
}  
add_two_numbers(5, 7)
```

```
## [1] 12
```

Function returns

- Return within a conditional

```
mathify_two_numbers <- function(a, b, operator = "+") {  
  if (operator == "+") {  
    return(a + b)  
  } else if (operator == "-") {  
    return(a - b)  
  } else if (operator == "*") {  
    return(a * b)  
  }  
  stop('Operator must be either "+", "-", or "*"')  
}
```

```
mathify_two_numbers(6, 8, operator = "+")
```

```
## [1] 14
```

```
mathify_two_numbers(6, 8, operator = "/")
```

```
## Error in mathify_two_numbers(6, 8, operator = "/"): Operator must be either "+", "-", or "*"
```


Function returns

- Make the function better -- add error checking at the beginning
- More readable

```
mathify_two_numbers <- function(a, b, operator = "+") {  
  if (!(operator %in% c("+", "-", "*"))) {  
    stop('Operator must be either "+", "-", or "*"')  
  }  
  
  if (operator == "+") {  
    return(a + b)  
  } else if (operator == "-") {  
    return(a - b)  
  } else if (operator == "*") {  
    return(a * b)  
  }  
}
```

Making functions pipeable

- Pipes are a chain of `data.frames`
- Use `invisible()` to return a `data.frame` without printing

```
show_missing <- function(df) {  
  n <- sum(is.na(df))  
  # use message not cat/print  
  message("Missing values: ", n)  
  # return the df without printing  
  invisible(df)  
}
```

```
msleep %>%  
  select(brainwt) %>%  
  show_missing() %>%  
  mutate(brainwt_lbs = 2.2 * brainwt)
```

```
## Missing values: 27
```

```
## # A tibble: 83 × 2  
##   brainwt brainwt_lbs  
##   <dbl>    <dbl>  
## 1 NA      NA  
## 2 0.0155  0.0341  
## 3 NA      NA  
## 4 0.00029 0.000638  
## 5 0.423   0.931
```

Environments

- Global and local environments

```
# y is assigned the value 4 in the global environment  
y <- 4  
  
print(y)
```

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

```
f <- function() {  
  # y is assigned the value 6 in the local environment  
  y <- 6  
  print(y)  
}  
  
# in the local environment in the function f, y is 6  
f()
```

```
## [1] 6
```

```
# the variable y is still 4 in global environment  
# the local environment doesn't overwrite the global environment  
print(y)
```

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

Environments

- Don't use global environment variables inside functions

```
# y is assigned the value 4 in the global environment  
y <- 4  
  
print(y)
```

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

```
f <- function() {  
  # y is a variable in the global environment  
  print(y)  
}  
  
f()
```

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

```
print(y)
```

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

```
y <- 6  
f()
```

```
## [1] 6
```

Environments

- The flexibility of environments is very powerful
- With great power comes great responsibility

Vectors

- Atomic vectors
 - logical, integer, double, complex, character, and raw
 - integer and double types are known as numeric
- List are recursive vectors
- Vectors have two properties

- type: determine using `typeof()`

```
typeof(c("a", "b", "c"))
```

```
## [1] "character"
```

```
typeof(1:10)
```

```
## [1] "integer"
```

- `length()`

```
length(1:10)
```

```
## [1] 10
```

```
length(list("a", "b", 1:10))
```

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

Types

- logical: TRUE, FALSE, and NA
- numeric: integer and double

```
typeof(1)
```

```
## [1] "double"
```

```
typeof(1L)
```

```
## [1] "integer"
```

- integers have special values of NA
- doubles have special values of NA, -Inf, Inf, and NaN

```
c(-1, 0, 1) / 0
```

```
## [1] -Inf NaN Inf
```

- Test for the special cases with is.finite(), is.infinite(), is.na(), and is.nan()
- characters == strings

Working with vectors

- Converting between types -- coercion

```
as.logical(c("TRUE", "FALSE"))
```

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

```
as.logical(c(0, 1, 2))
```

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

```
as.integer(c("3.5", "4"))
```

```
## [1] 3 4
```

```
as.integer(c(1.3, 4.3))
```

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

```
as.factor(c("a", "b"))
```

```
## [1] a b  
## Levels: a b
```

```
as.double(c("3.4", 6.2))
```

```
## [1] 3.39999999999999911182 6.200000000000000177
```

```
as.double(c("four", "6"))
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA 6
```

```
as.character(factor(c("a", "b")))
```

```
## [1] "a" "b"
```

```
as.character(c(3, 4))
```

```
## [1] "3" "4"
```


Coercion

- Can you fix the vector type when loading the data? (`col_types` in `readr` library)
- Coercion can be explicit (using the `as.` functions) or implicit (done within another function)

```
x <- 1:20  
x > 10
```

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

```
sum(x > 10)
```

```
## [1] 10
```

```
mean(x > 10)
```

```
## [1] 0.5
```

Coercion

```
typeof(c("TRUE", 1))
```

```
## [1] "character"
```

```
typeof(c(TRUE, 1))
```

```
## [1] "double"
```

```
typeof(c(TRUE, 1L))
```

```
## [1] "integer"
```

Test functions

	lgl	int	dbl	chr	list
<code>is_logical()</code>	x				
<code>is_integer()</code>		x			
<code>is_double()</code>			x		
<code>is_numeric()</code>		x	x		
<code>is_character()</code>				x	
<code>is_atomic()</code>	x	x	x	x	
<code>is_list()</code>					x
<code>is_vector()</code>	x	x	x	x	x

Vector recycling

- scalars will be recycled

```
1:10 + 5
```

```
## [1] 6 7 8 9 10 11 12 13 14 15
```

- What about vectors that are multiples?

```
1:10 + 1:5
```

```
## [1] 2 4 6 8 10 7 9 11 13 15
```

```
matrix(1:2, 2, 3)
```

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

```
matrix(1:3, 2, 3, byrow = TRUE)
```

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

Vector recycling

- what about vectors that aren't multiples?

```
1:10 + 1:3
```

```
## Warning in 1:10 + 1:3: longer object length is not a multiple of shorter object length
```

```
## [1]  2  4  6  5  7  9  8 10 12 11
```

- How does recycling work for `data.frames`?

```
dat <- data.frame(x = 1:10, y = 1:2)
```

- This can get you into trouble if you aren't paying attention
- How does recycling work for `tibbles`?

```
dat <- tibble(x = 1:10, y = 1:2)
```

```
## Error:  
## ! Tibble columns must have compatible sizes.  
## • Size 10: Existing data.  
## • Size 2: Column y.  
## i Only values of size one are recycled.
```

- Can you fix this using `rep()`?

Vectors

- Naming vectors

```
x <- c(a = 1, b = 2, c = 3, "three" = 4)
x
```

```
##      a      b      c three
##      1      2      3      4
```

- Subsetting vectors

```
# select 2 and 4th element
x[c(2, 4)]
```

```
##      b three
##      2      4
```

```
# repeated selection
x[c(2, 4, 2, 4, 3, 2, 1)]
```

```
##      b three      b three      c      b      a
##      2      4      2      4      3      2      1
```

Vectors

- dropping elements of vectors

```
x
```

```
##      a      b      c three
##      1      2      3      4
```

```
# drop first and 3rd observation
x[c(-1, -3)]
```

```
##      b three
##      2      4
```

```
# subsetting with conditions
x[x > 2] ## values greater than 2
```

```
##      c three
##      3      4
```

```
# subsetting with names
x[names(x) %in% c("a", "c")]
```

```
## a c
## 1 3
```

```
x[c("a", "c")]
```

```
## a c
## 1 3
```

Lists

- lists are recursive vectors
- Good visuals [here](#), [here](#), and [here](#)

```
nba_player <- list(  
  name = "Lebron James",  
  height = 81,  
  weight = 250,  
  nicknames = list("King James", "LBJ", "Bron-Bron")  
)  
nba_player
```

```
## $name  
## [1] "Lebron James"  
##  
## $height  
## [1] 81  
##  
## $weight  
## [1] 250  
##  
## $nicknames  
## $nicknames[[1]]  
## [1] "King James"  
##  
## $nicknames[[2]]  
## [1] "LBJ"  
##  
## $nicknames[[3]]
```


Lists

- `str()` shows the structure

```
str(nba_player)
```

```
## List of 4
## $ name      : chr "Lebron James"
## $ height    : num 81
## $ weight     : num 250
## $ nicknames:List of 3
## ..$ : chr "King James"
## ..$ : chr "LBJ"
## ..$ : chr "Bron-Bron"
```

Lists

- subsetting lists
 - `[` selects a sub-list and always returns a list
 - `[[` selects a sub-component from a list and removes the hierarchical structure

```
nba_player["name"]
```

```
## $name  
## [1] "Lebron James"
```

```
str(nba_player["name"])
```

```
## List of 1  
## $ name: chr "Lebron James"
```

```
nba_player[1]
```

```
## $name  
## [1] "Lebron James"
```

```
nba_player[["name"]]
```

```
## [1] "Lebron James"
```

```
str(nba_player[["name"]])
```

```
## chr "Lebron James"
```

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

```
## [1] "Lebron James"
```

Lists

- Can extract named components from lists using `$`. Note this behaves like `[[` but without quotes.

```
nba_player$name
```

```
## [1] "Lebron James"
```

```
str(nba_player$name)
```

```
## chr "Lebron James"
```

```
nba_player[["name"]]
```

```
## [1] "Lebron James"
```

```
str(nba_player[["name"]])
```

```
## chr "Lebron James"
```

Attributes

- Get/set individual attributes with `attr()` or all attributes with `attributes()`

```
attr(nba_player, "names")
```

```
## [1] "name"      "height"    "weight"    "nicknames"
```

```
attributes(nba_player)
```

```
## $names  
## [1] "name"      "height"    "weight"    "nicknames"
```

- Most common attributes
 - names
 - dimensions (`dims`)
 - class

Generic functions and methods

- Many functions in R change behavior based on the class of data input
- These are called **methods**
- For example, the methods for the **generic function** `summary()` are

```
methods(summary)
```

```
## [1] summary,ANY-method
## [4] summary,hexbin-method
## [7] summary.aovlist*
## [10] summary.bitwhich*
## [13] summary.col_spec*
## [16] summary.corARMA*
## [19] summary.corExp*
## [22] summary.corLin*
## [25] summary.corSpher*
## [28] summary.data.frame
## [31] summary.Duration*
## [34] summary.gam*
## [37] summary.gls*
## [40] summary.infl*
## [43] summary.lm
## [46] summary.loess*
## [49] summary.mlm*
## [52] summary.nlsList*
## [55] summary.pdCompSymm*
## [58] summary.pdIdnot*
```

```
summary,DBIObject-method
summary,sparseMatrix-method
summary.aspell*
summary.booltype*
summary.connection
summary.corCAR1*
summary.corGaus*
summary.corNatural*
summary.corStruct*
summary.Date
summary.ecdf*
summary.ggplot*
summary.haven_labelled*
summary.integer64*
summary.lme*
summary.manova
summary.modelStruct*
summary.packageStatus*
summary.pdDiag*
summary.pdLogChol*
```

```
summary,diagonalMat
summary.aov
summary.bit*
summary.check_package
summary.corAR1*
summary.corCompSymm
summary.corIdent*
summary.corRatio*
summary.corSymm*
summary.default
summary.factor
summary.glm
summary.hcl_palette
summary.Interval*
summary.lmList*
summary.matrix
summary.nls*
summary.pdBlocked*
summary.pdIdent*
summary.pdMat*
```

Generic functions and methods

```
dat <- data.frame(x = 1:10, y = 1:10 + rnorm(10))
```

```
summary(dat)
```

```
##           x           y
## Min.      : 1.00   Min.   :1.167603809839999895
## 1st Qu.: 3.25   1st Qu.:2.449001681009999993
## Median : 5.50   Median :5.360305710189999957
## Mean   : 5.50   Mean   :5.27439844592000017
## 3rd Qu.: 7.75   3rd Qu.:7.33577609524999996
## Max.    :10.00   Max.    :9.949641339339999939
```

```
# fit a best fit line
```

```
mod <- lm(y ~ x, data = dat)
```

```
summary(mod)
```

```
## Call:
```

```
## lm(formula = y ~ x, data = dat)
```

```
## Residuals:
```

```
##                               Min                               1Q
## -1.11408595237354668406 -0.28623804559997167996
```

```
##
```

```
## Coefficients:
```

```
##                               Estimate
## (Intercept) -0.13903262826988152523  0.529122455
## x           0.98426019530637010035  0.085275805
```

```
## ---
```

```
## Signif. codes:  0 '***' 0.0010000000000000000002
## 0.1000000000000000000055511 ' ' 1
```

```
##
```

```
## Residual standard error: 0.774555963880933950882
```

```
## Multiple R-squared:  0.9433506474461809121, Adjusted R-squared:  0.9333506474461809121
```

```
## F-statistic: 133.2196192780788238 on 1 and 8 DF, p-value: 1.11e-11
```

Generic functions and methods

- See what the `summary()` function does

```
getS3method("summary", "data.frame")
```

```
## function (object, maxsum = 7L, digits = max(3L, getOption("digits") -
##     3L), ...)
## {
##     ncw <- function(x) {
##         z <- nchar(x, type = "w")
##         if (any(na <- is.na(z))) {
##             z[na] <- nchar(encodeString(z[na]), "b")
##         }
##         z
##     }
##     z <- lapply(X = as.list(object), FUN = summary, maxsum = maxsum,
##         digits = 12L, ...)
##     nv <- length(object)
##     nm <- names(object)
##     lw <- numeric(nv)
##     nr <- if (nv)
##         max(vapply(z, function(x) NROW(x) + !is.null(attr(x,
##             "NAs"))), integer(1)))
##     else 0
##     for (i in seq_len(nv)) {
##         sms <- z[[i]]
##         if (is.matrix(sms)) {
##             cn <- paste(nm[i], gsub("^ +", "", colnames(sms),
##                 useBytes = TRUE), sep = ".")
##             tmp <- format(sms)
##             if (nrow(sms) < nr)
```

Augmented vectors

- Vectors with additional attributes
 - factors

```
x <- factor(c("ab", "cd", "ab"), levels = c("ab", "cd", "ef"))
typeof(x)
```

```
## [1] "integer"
```

```
attributes(x)
```

```
## $levels
## [1] "ab" "cd" "ef"
##
## $class
## [1] "factor"
```


Augmented vectors

- Vectors with additional attributes
 - dates
 - date-times

```
x <- as.Date("1971-01-01")  
unclass(x)
```

```
## [1] 365
```

```
typeof(x)
```

```
## [1] "double"
```

```
attributes(x)
```

```
## $class  
## [1] "Date"
```

```
x <- lubridate::ymd_hm("1970-01-01 01:00")  
unclass(x)
```

```
## [1] 3600  
## attr("tzone")  
## [1] "UTC"
```

```
typeof(x)
```

```
## [1] "double"
```

```
attributes(x)
```

```
## $class  
## [1] "POSIXct" "POSIXt"  
##  
## $tzone  
## [1] "UTC"
```

Augmented vectors

- Vectors with additional attributes
 - tibbles

```
tb <- tibble::tibble(x = 1:5, y = 5:1)
typeof(tb)
```

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

```
attributes(tb)
```

```
## $class
## [1] "tbl_df"      "tbl"        "data.frame"
##
## $row.names
## [1] 1 2 3 4 5
##
## $names
## [1] "x" "y"
```

```
df <- data.frame(x = 1:5, y = 5:1)
typeof(df)
```

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

```
attributes(df)
```

```
## $names
## [1] "x" "y"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3 4 5
```

Iteration

- For loops
- Can use `seq_along()` to generate the iterator

```
beatles <- c("John", "Paul", "George", "Ringo")
for (i in seq_along(beatles)) {
  # print(i)
  print(beatles[i])
}
```

```
## [1] "John"
## [1] "Paul"
## [1] "George"
## [1] "Ringo"
```

For Loops

```
dat <- data.frame(x = 1:4, y = 12:15, z = 5:8)
```

- Lots of copy paste
- Only works for a few examples

```
column_means <- c(mean(dat$x), mean(dat$y), mean(dat$z))  
column_means
```

```
## [1] 2.5 13.5 6.5
```

- A little more work, generalizes to more columns
- Easier to change the data and the functions

```
column_means <- vector("double", length(dat)) ## output  
for (i in seq_along(dat)) {                    ## sequence  
  # print(i)  
  column_means[i] <- mean(dat[[i]])             ## body  
}  
column_means
```

```
## [1] 2.5 13.5 6.5
```

For loops to modify an existing data object

- Typically better to create a new data object (not possible for very large data)

```
standardize_data <- function(x) {  
  ## convert x to mean 0, standard deviation 1  
  (x - mean(x)) / sd(x)  
}
```

```
dat <- data.frame(x = 1:4, y = 12:15, z = 5:8)
```

```
for (i in seq_along(dat)) {                ## sequence  
  # print(i)  
  dat[[i]] <- standardize_data(dat[[i]])    ## body  
}  
dat
```

```
##           x           y           z  
## 1 -1.161895003862225106417 -1.161895003862225106417 -1.161895003862225106417  
## 2 -0.387298334620741702139 -0.387298334620741702139 -0.387298334620741702139  
## 3  0.387298334620741702139  0.387298334620741702139  0.387298334620741702139  
## 4  1.161895003862225106417  1.161895003862225106417  1.161895003862225106417
```

```
apply(dat, 2, mean) # column means
```

```
## x y z  
## 0 0 0
```

```
apply(dat, 2, sd) # column sds
```

```
## x y z  
## 1 1 1
```

For loops

- Three ways to iterate
 1. Iterate over the numeric indices using `seq_along()`
 - `for (i in seq_along(dat)) {}`
 - can extract the data with `dat[[i]]`
 2. Iterate over the elements
 - `for (i in dat) {}`
 3. Iterate over the names
 - `for (nms in names(dat)) {}`
 - can extract the data with `dat[[nms]]`
 - Make sure you keep the names in the output vector
 - `names(out) <- names(dat)`

For Loops

- Iterate over elements

```
beatles <- c("John", "Paul", "George", "Ringo")  
for (i in beatles) {  
  print(i)  
}
```

```
## [1] "John"  
## [1] "Paul"  
## [1] "George"  
## [1] "Ringo"
```

For loops

- Iterate over names

```
bands <- data.frame(  
  beatles = c("John", "Paul", "George", "Ringo"),  
  stones = c("Mick", "Keith", "Charlie", "Ronnie"),  
  zeppelin = c("Robert", "Jimmy", "John Paul", "John")  
)
```

```
for (i in names(bands)) {  
  print(i)  
}
```

```
## [1] "beatles"  
## [1] "stones"  
## [1] "zeppelin"
```

```
for (i in names(bands)) {  
  print(bands[[i]])  
}
```

```
## [1] "John"    "Paul"    "George" "Ringo"  
## [1] "Mick"    "Keith"   "Charlie" "Ronnie"  
## [1] "Robert"  "Jimmy"   "John Paul" "John"
```


For loops of unknown lengths

- Bad: grow the vector as elements are created
- Scales $O(n^2)$:
 - doubling the data results in 4 times longer computation
 - quadrupling the data results in 16 times longer computation

```
out <- vector("numeric")
for (i in 1:100) {
  n <- sample(1:50, 1) # choose a sample size
  out <- c(out, rnorm(n)) # append random values to the vector
}
length(out) # this will change each time you run this unless you set.seed()
```

```
## [1] 2741
```

For loops of unknown lengths

- Better:
 - create a list
 - fill each element of the list with unequal sized data
 - "flatten" the list using `unlist()`

```
out <- list(length = 100)
for (i in 1:100) {
  n <- sample(1:50, 1) # choose a sample size
  out[[i]] <- rnorm(n)
  ## don't do this -- it will cause issues
  # out[[i]] <- c(out, rnorm(n))
}
length(out)
```

```
## [1] 100
```

```
# view first 5 elements
str(out[1:5])
```

```
## List of 5
## $ length: num [1:25] -0.0425 -0.3 1.4659 -0.0556 -0.7716 ...
## $      : num [1:22] -0.245 0.411 -0.369 -1.142 -1.357 ...
## $      : num [1:27] -0.502 0.541 -1.69 -0.842 0.539 ...
## $      : num [1:44] -1.1216 0.2856 -0.0827 0.1539 0.391 ...
## $      : num [1:4] -0.687 1.671 0.7 -1.471
```

```
# flatten the list
str(unlist(out))
```

```
## Named num [1:2600] -0.0425 -0.3 1.4659 -0.0556 -0.7716 ...
## - attr(*, "names")= chr [1:2600] "length1" "length2" "length3" "length4" ...
```

Loops of unknown number of iterations

- Go bowling and bowl until a strike
- Flip a coin and don't stop until 3 heads in a row
- `while` loops -- rarely used

```
flip <- function() sample(c("T", "H"), 1)

flips <- 0
nheads <- 0

while (nheads < 3) {
  if (flip() == "H") {
    nheads <- nheads + 1
  } else {
    nheads <- 0
  }
  flips <- flips + 1
}
flips
```

```
## [1] 17
```

Functionals

- The real power of R is that it is a functional programming language
- You can write a function that takes other functions as arguments
- Example: use for loop to calculate column means

```
dat <- data.frame(x = rnorm(10), y = rnorm(10), z = rnorm(10))
colmeans <- vector(mode = "numeric", length = length(dat))
for (i in seq_along(dat)) {
  colmeans[i] <- mean(dat[[i]])
}
colmeans
```

```
## [1] 0.51383134490239301683800 0.01100435577200373389595 0.10249199142045402333157
```

- Make this into a function

```
col_means <- function(dat) {
  out <- vector(mode = "numeric", length = length(dat))
  for (i in seq_along(dat)) {
    out[i] <- mean(dat[[i]])
  }
  return(out)
}
col_means(dat)
```

```
## [1] 0.51383134490239301683800 0.01100435577200373389595 0.10249199142045402333157
```

Functionals

- Writing the function allows you to calculate column means for different datasets
- What if you want to calculate column medians?
- Write a function!

```
col_medians <- function(dat) {  
  out <- vector(mode = "numeric", length = length(dat))  
  for (i in seq_along(dat)) {  
    out[i] <- median(dat[[i]])  
  }  
  return(out)  
}  
col_medians(dat)
```

```
## [1] 0.287755012807710930378136 0.009074975753924879873225 0.272391215562530986105116
```

- Lots of copy/paste
- Chances for errors

Functionals

- What if we made `mean()` and `median()` (or any function) an argument to the function?

```
col_fun <- function(dat, fun) {  
  out <- vector(mode = "numeric", length = length(dat))  
  for (i in seq_along(dat)) {  
    ## apply the function here  
    out[i] <- fun(dat[[i]])  
  }  
  return(out)  
}  
col_fun(dat, mean)
```

```
## [1] 0.51383134490239301683800 0.01100435577200373389595 0.10249199142045402333157
```

```
col_fun(dat, median)
```

```
## [1] 0.287755012807710930378136 0.009074975753924879873225 0.272391215562530986105116
```

```
col_fun(dat, sd)
```

```
## [1] 0.7736692898359733838731 0.9292729192313109454204 1.0716754320971118019656
```

```
col_fun(dat, min)
```

```
## [1] -0.5407597082957986689422 -1.1241797328462226346346 -2.0335391439616645747890
```

Functionals

- The idea is to pass a function as an argument to another function
- Super powerful programming paradigm
- Common in the `*apply` family of functions
- Basis of the `purrr` package

The `map` family of functions

- Loop over a vector, do something to each element, and output the results
- Different map functions based on return type
 - `map()`: returns a list
 - `map_lgl()`: returns a logical vector
 - `map_int()`: returns an integer vector
 - `map_dbl()`: returns a numeric vector
 - `map_chr()`: returns a character vector

```
map_dbl(dat, mean)
```

```
##                               x                               y                               z
## 0.51383134490239301683800 0.01100435577200373389595 0.10249199142045402333157
```

```
map_dbl(dat, median)
```

```
##                               x                               y                               z
## 0.287755012807710930378136 0.009074975753924879873225 0.272391215562530986105116
```

```
map_dbl(dat, sd)
```

```
##                               x                               y                               z
## 0.7736692898359733838731 0.9292729192313109454204 1.0716754320971118019656
```


maps and pipes

- maps make the code easier to read by focusing attention on the function being called

```
dat %>%  
  map_dbl(mean)
```

```
##                x                y                z  
## 0.51383134490239301683800 0.01100435577200373389595 0.10249199142045402333157
```

```
dat %>%  
  map_dbl(median)
```

```
##                x                y                z  
## 0.287755012807710930378136 0.009074975753924879873225 0.272391215562530986105116
```

```
dat %>%  
  map_dbl(sd)
```

```
##                x                y                z  
## 0.7736692898359733838731 0.9292729192313109454204 1.0716754320971118019656
```

maps

- The `map_*()` functions allow for `...` arguments to be passed to the function
- `maps` preserve names

```
dat$x[1] <- NA
dat %>%
  map_dbl(mean)
```

```
##              x              y              z
## NA 0.01100435577200373389595 0.10249199142045402333157
```

```
dat %>%
  map_dbl(mean, na.rm = TRUE)
```

```
##              x              y              z
## 0.54840585895409355021712 0.01100435577200373389595 0.10249199142045402333157
```

```
# ?map_dbl
```

- `map_*(.x, .f, ...)`
 - Input data `.x`
 - Function (or formula or character) `.f`
 - Additional arguments `...`

Errors and failures

- repeatedly applying functions using `map_*()` can result in errors

```
dat <- data.frame(x = 1, y = 2, z = "a")  
dat %>%  
  map_dbl(log)
```

```
## Error in .Primitive("log")(x, base): non-numeric argument to mathematical function
```

- Can't take a logarithm of "a"

Errors and failures

- Can use `safely()` to capture errors
- Returns a list where each element of the list has two components: `results` and `errors`

```
out <- dat %>%  
  map(safely(log))
```

```
str(out)
```

```
## List of 3  
## $ x:List of 2  
## ..$ result: num 0  
## ..$ error : NULL  
## $ y:List of 2  
## ..$ result: num 0.693  
## ..$ error : NULL  
## $ z:List of 2  
## ..$ result: NULL  
## ..$ error :List of 2  
## .. ..$ message: chr "non-numeric argument to mathematical function"  
## .. ..$ call : language .Primitive("log")(x, base)  
## .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Errors and failures

- `purrr::transpose()` can convert a list of elements with two items each into two lists

```
out <- transpose(out)
str(out)
```

```
## List of 2
## $ result:List of 3
## ..$ x: num 0
## ..$ y: num 0.693
## ..$ z: NULL
## $ error :List of 3
## ..$ x: NULL
## ..$ y: NULL
## ..$ z:List of 2
## .. ..$ message: chr "non-numeric argument to mathematical function"
## .. ..$ call : language .Primitive("log")(x, base)
## .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Errors and failure

- Extract the errors and save as a vector

```
no_error <- out$error %>%  
  map_lgl(is_null)  
str(no_error)
```

```
## Named logi [1:3] TRUE TRUE FALSE  
## - attr(*, "names")= chr [1:3] "x" "y" "z"
```

```
out$result[no_error]
```

```
## $x  
## [1] 0  
##  
## $y  
## [1] 0.6931471805599452862268
```

- Other functions for errors: `possibly()` and `quietly()`

Mapping over multiple arguments

- Often, you want to iterate over more than one vector of data
 - `map2()` iterates over two input arguments
 - `pmap()`: positional map iterates over multiple arguments by position
 - recall positional arguments when loading data in python

map2()

- Find the maximum of $(x + y + x * y)/4 + \sin(8 * \pi * (x - 0.5)) * (y - 0.5)^2 / \cos(\pi y)$ for $0 < x < 1$ and $0 < y < 1$
- Setup a grid over the unit square

```
grid <- expand.grid(x = seq(0, 1, length = 100), y = seq(0, 1, length = 100))
str(grid)
```

```
## 'data.frame':   10000 obs. of  2 variables:
## $ x: num  0 0.0101 0.0202 0.0303 0.0404 ...
## $ y: num  0 0 0 0 0 0 0 0 0 0 ...
## - attr(*, "out.attrs")=List of 2
## ..$ dim      : Named int [1:2] 100 100
## .. ..- attr(*, "names")= chr [1:2] "x" "y"
## ..$ dimnames:List of 2
## .. ..$ x: chr [1:100] "x=0.000000000000000000000000" "x=0.0101010101010101010186870" "x=0.02020202020202020202737" ...
## .. ..$ y: chr [1:100] "y=0.000000000000000000000000" "y=0.0101010101010101010186870" "y=0.02020202020202020202737" ...
```

```
useless_function <- function(x, y) {  
  (x + y + x * y) / 4 + sin(8 * pi * (x - 0.5)) * (y - 0.5)^2 / cos(pi * y)  
}
```


Example: `map2()`

- Find the maximum of $(x + y + x * y)/4 + \sin(8 * \pi * (x - 0.5)) * (y - 0.5)^2 / \cos(\pi y)$ for $0 < x < 1$ and $0 < y < 1$

```
values <- map2(grid$x, grid$y, useless_function) %>%  
  unlist()
```

- Find which `x` and `y` give the maximum values

```
max_values <- which.max(values)  
grid[max_values, ]
```

```
##                                x y  
## 9994 0.9393939393939394477684 1
```

Example: `map2()`

- Find the maximum of $(x + y + x * y)/4 + \sin(8 * \pi * (x - 0.5)) * (y - 0.5)^2 / \cos(\pi y)$ for $0 < x < 1$ and $0 < y < 1$
- Visualize the function with a `geom_raster()` geometry data on a regular grid

```
grid$values <- values  
  
ggplot(grid, aes(x = x, y = y, fill = values)) +  
  geom_raster() +  
  scale_fill_viridis_c() +  
  geom_point(data = grid[max_values, ], aes(x = x, y = y))
```

Example: `pmap()`

- Applies the map in order of the arguments

```
worthless_function <- function(x, y, z) {  
  useless_function(x, y)^z * cos(pi * z / 8) - z^2 + z  
}
```

- Find the maximum over a grid of three input values

```
grid <- expand.grid(  
  x = seq(0, 1, length = 50),  
  y = seq(0, 1, length = 50),  
  z = seq(0, 1, length = 50)  
)  
str(grid)
```

```
## 'data.frame':    125000 obs. of  3 variables:  
## $ x: num  0 0.0204 0.0408 0.0612 0.0816 ...  
## $ y: num  0 0 0 0 0 0 0 0 0 0 ...  
## $ z: num  0 0 0 0 0 0 0 0 0 0 ...  
## - attr(*, "out.attrs")=List of 2  
## ..$ dim      : Named int [1:3] 50 50 50  
## .. ..- attr(*, "names")= chr [1:3] "x" "y" "z"  
## ..$ dimnames:List of 3  
## .. ..$ x: chr [1:50] "x=0.000000000000000000000000000000" "x=0.02040816326530612082046" "x=0.040816326530612082046" ...  
## .. ..$ y: chr [1:50] "y=0.000000000000000000000000000000" "y=0.02040816326530612082046" "y=0.040816326530612082046" ...  
## .. ..$ z: chr [1:50] "z=0.000000000000000000000000000000" "z=0.02040816326530612082046" "z=0.040816326530612082046" ...
```

Example: pmap ()

- `grid` is a `data.frame` with 3 variables
- `worthless_function()` takes 3 inputs

```
values <- pmap(grid, worthless_function) %>%  
  unlist()
```

- Find which `x` and `y` give the maximum values

```
max_values <- which.max(values)  
grid[max_values, ]
```

```
##                               x y                               z  
## 57497 0.9387755102040815646802 1 0.4489795918367346372335
```