Functional Programming

Alejandro Schuler, based on R for Data Science by Hadley Wickham
July 2021

- write and test your own functions
- write code that evaluates conditionally
- create, manipulate, and inspect lists
- iterate functions over lists of arguments
- iterate in parallel

Writing functions

Motivation

- It's handy to be able to reuse your code and automate repetitive tasks
- Writing your own functions allows you to do that
- When you write your code as functions, you can
  - name the function something evocative and readable
  - update the code in a single place instead of many
  - reduce the chance of making mistakes while copy-pasting
  - make your code shorter overall

Example

What does this code do? (recall that `df$col` is the same as `df %>% pull(col)`)

```
df <- tibble(
  a = rnorm(10), # 10 random numbers from a normal distribution
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

```
df$a = (df$a - min(df$a, na.rm = TRUE)) /
   (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b = (df$b - min(df$b, na.rm = TRUE)) /
   (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c = (df$c - min(df$c, na.rm = TRUE)) /
   (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d = (df$d - min(df$d, na.rm = TRUE)) /
   (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Example

(recall that `df$col` is the same as `df %>% pull(col)`)

```r
df$a = (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b = (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c = (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d = (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

- It looks like we're standardizing all the variables by their range so that they fall between 0 and 1
- But did you spot the mistake? The code runs with no errors…

Example

We already know a better way to do this:

```
df2 = df %>%
  mutate(across(
    a:d,
    ~ (.- min(.,na.rm=T)) / (max(.,na.rm=T) - min(.,na.rm=T))
  ))
```

- but it's still ugly and hard to read, especially with all the periods and parentheses

Example

```r
rescale_0_1 = function(vec) {
  (vec - min(vec))/(max(vec) - min(vec))
}

df2 = df %>%
  mutate(across(a:d, rescale_0_1))
```

- Much improved!
- The last two lines clearly say: replace all the columns with their rescaled versions
  - This is because the function name `rescale_0_1()` is informative and communicates what it does
  - If a user (or you a few weeks later) is curious about the specifics, they can check the function body
- … now we notice that `min()` is being computed twice in the function body, which is inefficient
- We are also not accounting for NAs

Example

```r
rescale_0_1 = function(vec) {
  vec_rng = range(vec, na.rm=T) # same as c(min(vec,na.rm=T), max(vec,na.rm=T))
  (vec - vec_rng[1])/(vec_rng[2] - vec_rng[1])
}

df2 = df %>%
  mutate_all(rescale_0_1)
```

- Since we have a function, we can make the change in a single place and improve the efficiency of multiple parts of our code
- Bonus question: why use `range()` instead of getting and saving the results of `min()` and `max()` separately?

Example

We can also test our function in cases where we know what the output should be to make sure it works as intended before we let it loose on the real data

```
rescale_0_1(c(0,0,0,0,0,1))
[1] 0 0 0 0 0 1
rescale_0_1(0:10)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
rescale_0_1(-10:0)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
x = c(0,1,runif(100))
all(x == rescale_0_1(x))
[1] TRUE
```

- These tests are a critical part of writing good code! It is helpful to save your tests in a separate file and organize them as you go

Function syntax

To write a function, just wrap your code in some special syntax that tells it what variables will be passed in and what will be returned

```
rescale_0_1 = function(x) {
  x_rng = range(x, na.rm=T)
  (x - x_rng[1])/(x_rng[2] - x_rng[1])
}

rescale_0_1(c(0,0,0,0,0,1))
[1] 0 0 0 0 0 1
rescale_0_1(0:10)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

- The syntax is `FUNCTION_NAME <- function(ARGUMENTS...) { CODE }`
- Just like assigning a variable, except what you put into `FUNCTION_NAME` now isn't a data frame, vector, etc, it's a function object that gets created by the `function(..) {...}` syntax
- At any point in the body you can `return()` the value, or R will automatically return the result of the last line of code in the body that gets run

Function syntax

To add a named argument, add an = after you declare it as a variable and write in the default value that you would like that variable to take

```
rescale_0_1 = function(x, na.rm=TRUE) {
  x_rng = range(x, na.rm=na.rm)
  (x - x_rng[1])/(x_rng[2] - x_rng[1])
}

rescale_0_1(c(0,0,0,0,0,1))
[1] 0 0 0 0 0 1
rescale_0_1(0:10)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

- All named arguments must go after positional arguments in the function declaration

Exercise: NAs in two vectors

- Write a function called both_na() that takes two vectors of the same length and returns the total number of positions that have an NA in both vectors
- Make a few vectors and test out your code

Note: functions are objects just like variables are

```
rescale_0_1
function(x, na.rm=TRUE) {
  x_rng = range(x, na.rm=na.rm)
  (x - x_rng[1])/(x_rng[2] - x_rng[1])
}
<bytecode: 0x7fce9327db08>
```

- As we've seen, they themselves can be passed as arguments to other functions

```
df2 = df %>% mutate(across(a:d, rescale_0_1))
```

- This is what **functional programming** means. The functions themselves are can be treated as regular objects like variables
- The name of the function is just what you call the “box” that the function (the code) lives in, just like variables names are names for “boxes” that contain data

Exercise: function factory

Without running this code, predict what the output will be:

```
f = function(x) {
  y = x
  function(y) {
    y + x
  }
}
f(1)(2)
```

Exercise: exponent function factory

Write a function called `power()` that takes an argument `n` and returns a function that takes an argument x and computes `x^n`

- Example use:

```
square = power(2)
cube = power(3)
square(2)
[1] 4
cube(2)
[1] 8
```

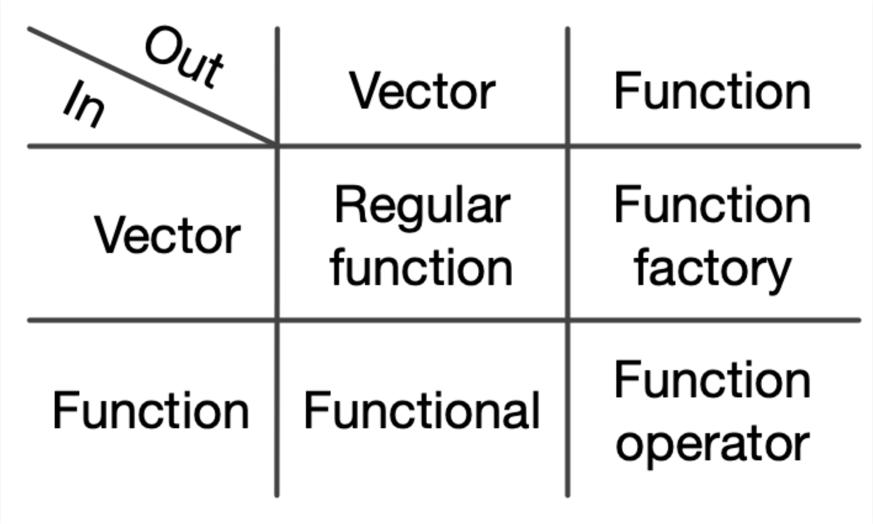
Function operators

- As we've seen, functions can also take other functions as arguments
- We can use this to write functions that take functions, modify them, and return the modified function

```
set_na.rm = function(f, na.rm) {
  function(x) {
    f(x, na.rm=na.rm)
  }
}
mean_na.rm = set_na.rm(mean, na.rm=T)

x = c(rnorm(100), NA)
mean_na.rm(x)
[1] -0.1878243
```

Functional programming

We've discussed a number of higher-order functions that either take functions as inputs, return functions as outputs, or both.



Out In	Vector	Function
Vector	Regular function	Function factory
Function	Functional	Function operator

- `mean()` is a "regular function". It takes data and returns data
- the `power()` function we made as an exercise is a "function factory". It takes data and returns a function (e.g. `square()` or `cube()`)
- `mutate(across(...))` are "functional". They take a function (and data) and return data
- the `set_na.rm` function we made as an exercise is a "function operator": it takes a function and returns a function

These categories are not explicit constructs that exist in R, they are just a way to think about the power of higher order functions.

Tidyverse + Functions

Passing column names to tidyverse within functions

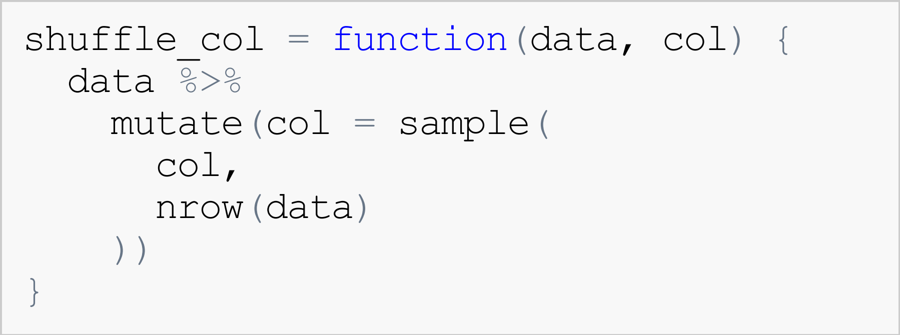
- Consider this function, which shuffles the `number` column in the passed-in dataframe

```
data = tibble(
  number = c(1,2,3),
  label = c('a','b','c')
)

shuffle_col_named_number = function(df) {
  df %>%
    mutate(number = sample(number, nrow(df)))
}

data %>% shuffle_col_named_number()
# A tibble: 3 x 2
  number label
   <dbl> <chr>
1       3 a
2       1 b
3       2 c
```

- What if we wanted to modify it so that `mpg` weren't hard-coded? Intuitively, we should write something like this:

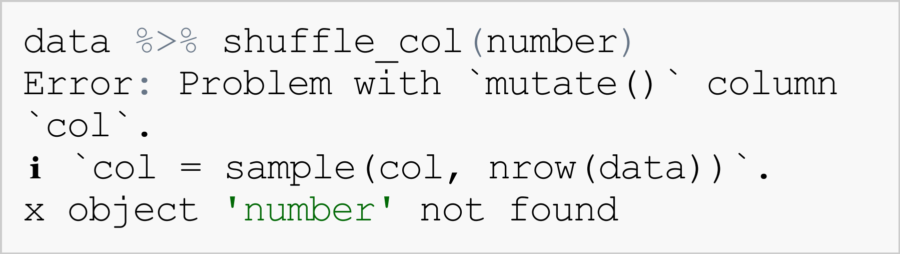
```
shuffle_col = function(data, col) {
  data %>%
    mutate(col = sample(col, nrow(data)))
}
```

Passing column names to tidyverse within functions



```
shuffle_col = function(data, col) {
  data %>%
    mutate(col = sample(
      col,
      nrow(data)
    ))
}
```

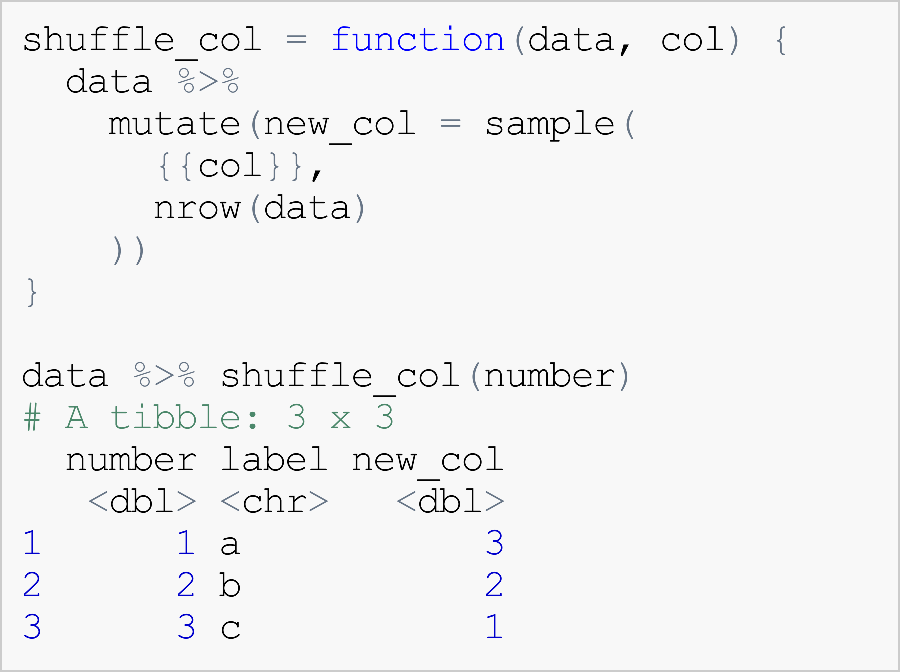
- Unfortunately, this doesn't work:



```
data %>% shuffle_col(number)
Error: Problem with `mutate()` column
`col`.
i `col = sample(col, nrow(data))`.
x object 'number' not found
```

- The problem is that R doesn't know that `number` should refer to a column in the data, not to an object in the global environment.

- The fix is to use the `{{...}}` syntax to surround the passed in variable name where it gets used in the function.



```
shuffle_col = function(data, col) {
  data %>%
    mutate(new_col = sample(
      {{col}},
      nrow(data)
    ))
}

data %>% shuffle_col(number)
# A tibble: 3 x 3
  number label new_col
   <dbl> <chr>   <dbl>
1      1 a           3
2      2 b           2
3      3 c           1
```

- This tells R not to look for that object in the global environment.
- Called “embracing”

Passing column names to tidyverse within functions

- The last thing we need to resolve is how to specify the naming of the new column.
- You may think this would work:

```
shuffle_col = function(data, col) {
  data %>%
    mutate({{col}} = sample(
      {{col}},
      nrow(data)
    ))
}
Error: <text>:3:20: unexpected '='
2:   data %>%
3:     mutate({{col}} =
                      ^
```

- But unfortunately this doesn't even parse as valid R code

- For arcane technical reasons, the right way to do this is to use the Walrus operator :=

```
shuffle_col = function(data, col) {
  data %>%
    mutate({{col}} := sample(
      {{col}},
      nrow(data)
    ))
}

data %>% shuffle_col(number)
# A tibble: 3 x 2
  number label
   <dbl> <chr>
1      2 a
2      1 b
3      3 c
```

Passing column names to tidyverse within functions as strings

- What if we wanted our function to work with character vectors so it could ingest the output of other code? For instance:

```
col_to_shuffle = names(data)[1] # whatever the name of the first column is
col_to_shuffle
[1] "number"
```

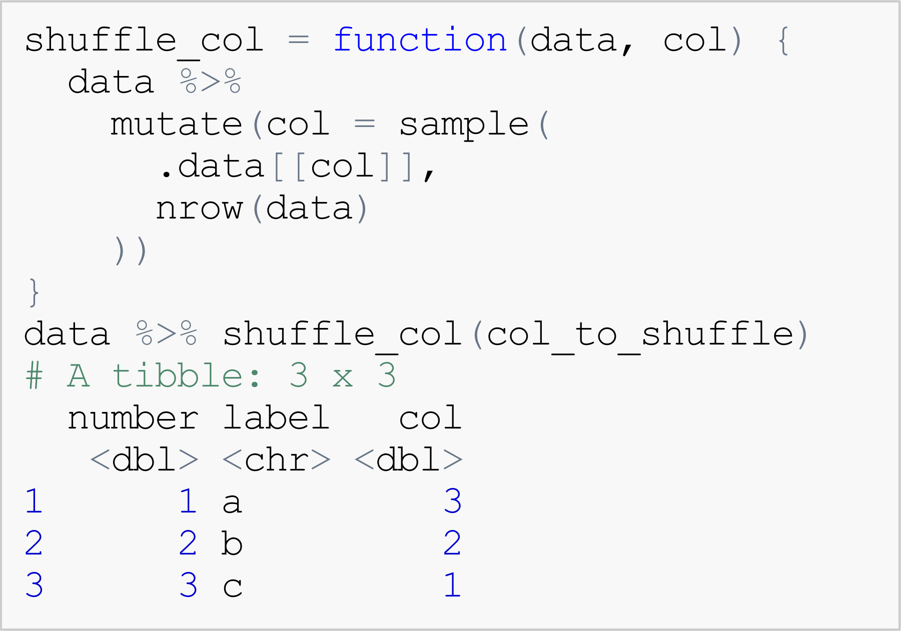
```
data %>% shuffle_col(col_to_shuffle)
Error: Problem with `mutate()` column `col_to_shuffle`.
ℹ `col_to_shuffle = sample(col_to_shuffle, nrow(data))`.
x cannot take a sample larger than the population when 'replace = FALSE'
```

- This fails because now that we've embraced the argument R expects it to be a bare column name that it will parse in the context of the data, not a string.
- The un-embraced version also fails (why?):

```
shuffle_col = function(data, col) {
  data %>%
    mutate(col = sample(
      col,
      nrow(data)
    ))
}
data %>% shuffle_col(col_to_shuffle)
Error: Problem with `mutate()` column `col`.
```

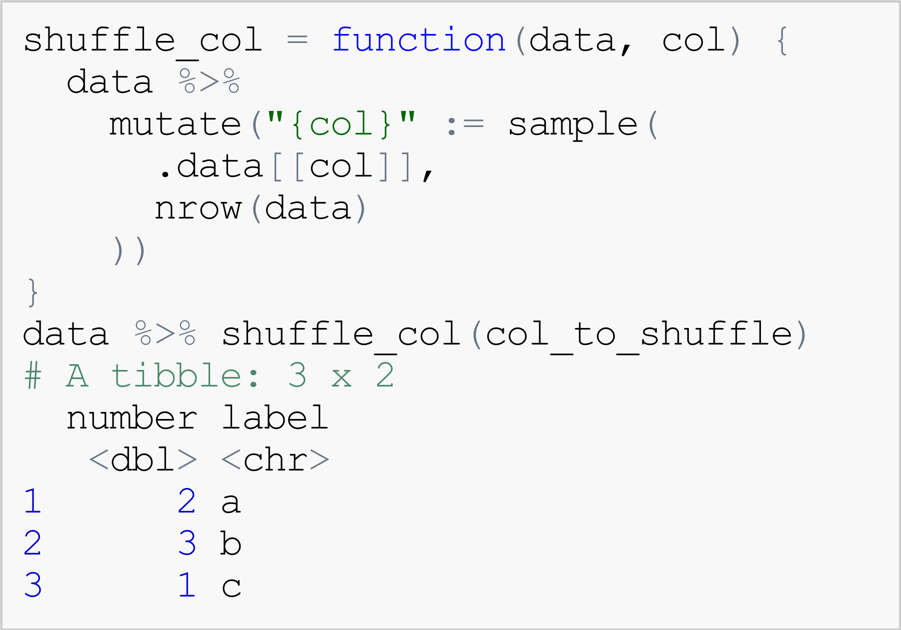
Passing column names to tidyverse within functions as strings

- To get this to work you have to use the special `.data` pronoun within the tidyverse function and index into that dataframe using the string column name:



```
shuffle_col = function(data, col) {
  data %>%
    mutate(col = sample(
      .data[[col]],
      nrow(data)
    ))
}
data %>% shuffle_col(col_to_shuffle)
# A tibble: 3 x 3
  number label   col
   <dbl> <chr> <dbl>
1      1 a         3
2      2 b         2
3      3 c         1
```

- If you want to use the arguments to assign new column names inside tidyverse functions you need to use single braces inside a string along with the Walrus operator:



```
shuffle_col = function(data, col) {
  data %>%
    mutate("{col}" := sample(
      .data[[col]],
      nrow(data)
    ))
}
data %>% shuffle_col(col_to_shuffle)
# A tibble: 3 x 2
  number label
   <dbl> <chr>
1      2 a
2      3 b
3      1 c
```

- Is this hideous? Yes. Confusing? Yes.
- This is the price you pay for being able to write beautiful, concise tidyverse code outside of functions!

Exercise: mean-by

1. Write a function that takes a dataframe and two bare (unquoted) column names as arguments. Your function should return the result of grouping the dataframe by the first column and then taking the mean of the second column by the groups defined by the first. Name the resulting column `"mean_by_grp"`.
2. Modify your function so that the user must pass in a string as a fourth argument. This string should be the name of the resulting summary column in the resulting dataframe
3. Modify your function so that the summarizing function (e.g. `mean`) can be passed in by the user as an argument

Conditional Evaluation

Conditional Evaluation

- An if statement allows you to conditionally execute code. It looks like this:

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

- The condition is code that evaluates to TRUE or FALSE

```
absolute_value = function(x) {
    if (x>0) {
        x
    } else {
        -x
    }
}
absolute_value(2)
[1] 2
absolute_value(-2)
[1] 2
```

Conditional Evaluation

- if_else is a function that condenses an if-else statement and is good when the condition and bodies of code are very short
- the first argument is the conditon, the second is what it returns when the condition is true, the third is what it returns when the condition is false

```
absolute_value = function(x) {
    if_else(x>0, x, -x)
}
absolute_value(2)
[1] 2
absolute_value(-2)
[1] 2
```

Multiple conditions

- You can evaluate multiple conditions with if… else if… else

```
valence = function(x) {
    if (x>0) {
        "positive"
    } else if (x<0) {
        "negative"
    } else {
        "zero"
    }
}
valence(-12)
[1] "negative"
valence(99)
[1] "positive"
valence(0)
[1] "zero"
```

Exercise: greeting

Write a greeting function that says “good morning”, “good afternoon”, or “good evening”, depending on the time of day of a `dttm` that the user passes in. (*Hint*: use an argument that defaults to `lubridate::now()`. That will make it easier to test your function. `lubridate::hour()` and `print()` may also be useful.)

Conditions

- Conditions can be anything that evaluates to a single `TRUE` or `FALSE`

```
library(lubridate)

sun_or_moon = function(time = now()) {
  this_hour = hour(time)
  if (this_hour < 6 | this_hour > 18) {
      "moon in the sky"
  } else {
      "sun in the sky"
  }
}
sun_or_moon(mdy_hm("Jan 2, 2000, 10:55pm"))
[1] "moon in the sky"
sun_or_moon(mdy_hm("Sept 10, 1990, 09:12am"))
[1] "sun in the sky"
```

- but not a logical vector or an NA

```
sun_or_moon(c(
  mdy_hm("Jan 2, 2000, 10:55pm"),
  mdy_hm("Sept 10, 1990, 09:12am")))
Warning in if (this_hour < 6 | this_hour > 18) {: the condition has length > 1
and only the first element will be used
[1] "moon in the sky"
```

Motivation for lists and iteration

```
sun_or_moon(c(
  mdy_hm("Jan 2, 2000, 10:55pm"),
  mdy_hm("Sept 10, 1990, 09:12am")))
Warning in if (this_hour < 6 | this_hour > 18) {: the condition has length > 1
and only the first element will be used
[1] "moon in the sky"
```

- We see this doesn't work. How can we tell the function to iterate over multiple sets of arguments?

```
list(mdy_hm("Jan 2, 2000, 10:55pm"),
     mdy_hm("Sept 10, 1990, 09:12am")) %>%
map(sun_or_moon)
[[1]]
[1] "moon in the sky"

[[2]]
[1] "sun in the sky"
```

- To do this, we need to understand:
  - how to build and manipulate `list`s
  - how to iterate with `purrr::map()` and its friends

Lists

Lists

- A `list` is like an atomic vector, except the elements don't have to be the same type of thing

```
a_vector = c(1,2,4,5)
maybe_a_vector = c(1,2,"hello",5,TRUE)
maybe_a_vector # R converted all of these things to strings!
[1] "1"     "2"     "hello" "5"     "TRUE"
```

- You make them with list() and you can index them like vectors

```
a_list = list(1,2,"hello",5,TRUE)
a_list[3:5]
[[1]]
[1] "hello"

[[2]]
[1] 5

[[3]]
[1] TRUE
```

- Anything can go in lists, including vectors, other lists, data frames, etc.
- In fact, a data frame (or tibble) is actually just a list of named column vectors with an enforced constraint that all of the vectors have to be of the same length. That's why the `df$col` syntax works for data frames.

Seeing into lists

- Use `str()` to dig into nested lists and other complicated objects

```
nested_list = list(a_list, 4, mtcars)
str(nested_list)
List of 3
 $ :List of 5
  ..$ : num 1
  ..$ : num 2
  ..$ : chr "hello"
  ..$ : num 5
  ..$ : logi TRUE
 $ : num 4
 $ :'data.frame':   32 obs. of  11 variables:
  ..$ mpg : num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
  ..$ cyl : num [1:32] 6 6 4 6 8 6 8 4 4 6 ...
  ..$ disp: num [1:32] 160 160 108 258 360 ...
  ..$ hp  : num [1:32] 110 110 93 110 175 105 245 62 95 123 ...
  ..$ drat: num [1:32] 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
  ..$ wt  : num [1:32] 2.62 2.88 2.32 3.21 3.44 ...
  ..$ qsec: num [1:32] 16.5 17 18.6 19.4 17 ...
  ..$ vs  : num [1:32] 0 0 1 1 0 1 0 1 1 1 ...
  ..$ am  : num [1:32] 1 1 1 0 0 0 0 0 0 0 ...
  ..$ gear: num [1:32] 4 4 4 3 3 3 3 4 4 4 ...
  ..$ carb: num [1:32] 4 4 1 1 2 1 4 2 2 4 ...
```

Getting elements from a list

- You can also name the elements in a list

```
a_list = list(
    first_number = 1,
    second_number = 2,
    a_string = "hello",
    third_number = 5,
    some_logical = TRUE)
```

- and then retrieve elements by name or position or using the tidyverse-friendly `pluck()`

```
a_list$a_string  # returns the element named "thrid_number"
[1] "hello"
a_list[[3]] # returns the 3rd element
[1] "hello"
a_list[3] # subsets the list, so returns a list of length 1 that contains a single
element (the third)
$a_string
[1] "hello"
a_list %>%
  pluck("a_string")
[1] "hello"
```

Using elements in list

- If you use the `magrittr` package, you can operate on items in lists with the `%$%` operator (fun fact: the `%>%` operator originally came from `magrittr`)

```
library(magrittr)
list(a=5, b=6) %$%
  rnorm(10, mean=a, sd=b)
 [1] 16.379239 15.661179  8.399627  5.094317  7.298344  4.729177  5.206111
 [8]  6.014161 11.990161  4.734776
```

- `%$%` makes the elements of the list on the left-hand side accessible in bare-name form to the expression on the right-hand side so you don't have to type extra dollar signs:

```
x =  list(a=5, b=6)
rnorm(10, mean=x$a, sd=x$b)
 [1]  4.397789  3.299333 14.244890  5.991014 12.845734 12.729541  8.557382
 [8]  3.302338 12.535304 10.459035
```

Exercise: getting things out of a list

Create this list in your workspace and write code to extract the element `"b"`.

```
x = list(
  list("a", "b", "c"),
  "d",
  "e",
  6
)
```

Functions returning multiple values

- A function can only return a single object
- Often, however, it makes sense to group the calculation of two or more things you want to return within a single function
- You can put all of that into a list and then retrun a single list

```
min_max = function(x) {
  x_sorted = sort(x)
  list(
    min = x_sorted[1],
    max = x_sorted[length(x)]
  )
}
```

- Why might this code be preferable to running `min()` and then `max()`?

Functions returning multiple values

- If you use the `zeallot` package, you can assign multiple values out of a list at once using the `c(...) %<-% ...` syntax

```
min_max = function(x) {
  x_sorted = sort(x)
  list(
    min = x_sorted[1],
    max = x_sorted[length(x)]
  )
}

library(zeallot)
c(min_x, max_x) %<-% min_max(rnorm(10))
min_x
[1] -1.482189
max_x
[1] 1.240181
```

Dots

- Besides positional and named arguments, functions in R can also be written to take a variable number of arguments!

```
sum(1)
[1] 1
sum(1,2,3,6) # pass in as many as you want and it still works
[1] 12
```

- Obviously there aren't an infinite number of `sum()` functions to work with all the possible different numbers of arguments
- To write functions like this, use the dots `...` syntax

```
space_text = function(...) {
  dots = list(...)
  str_c(dots, collapse=" ")
}

space_text("hello", "there,", "how", "are", "you")
[1] "hello there, how are you"
space_text("fine,", "thanks")
[1] "fine, thanks"
```

- You can get whatever is passed to the function and save it as a (possibly named) list using `list(...)`

Dots

- This is useful when you want to pass on a bunch of arbitrary named arguments to another function, but hardcode one or more of them

```
mean_no_na = function(...) {
  mean(..., na.rm=T)
}
x = c(rnorm(100), NA)
mean(x)
[1] NA
mean_no_na(x)
[1] -0.06457281
mean_no_na(x, trim=0.5)
[1] -0.1241746
mean(x, trim=0.5, na.rm=T)
[1] -0.1241746
```

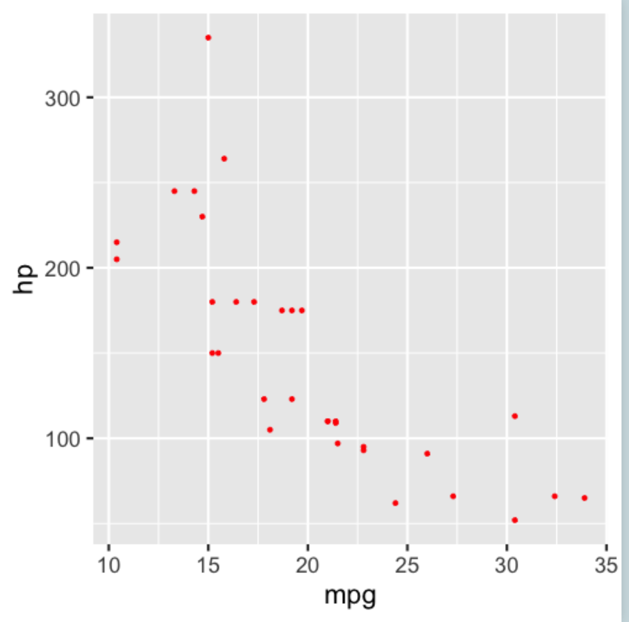
- Note how `trim` gets passed through as an argument to `mean()` even though it is not specified as an argument in the function declaration of `mean_no_na()`

Exercise: ggplot with default red points

- Use the dots to create a fully-featured function (call it `ggplot_redpoint()`), that works exactly like `ggplot()` except that it automatically adds a geom with red points.
- Test it out using data of your choice

Example:

```
mtcars %>%
ggplot_redpoint(aes(mpg, hp))
```



Iteration

Map

- Map is a function that takes a list (or vector) as its first argument and a function as its second argument
- Recall that functions are objects just like anything else so you can pass them around to other functions
- Map then runs that function on each element of the first argument, slaps the results together into a list, and returns that

```
times = list(mdy_hm("Jan 2, 2000, 10:55pm"),
             mdy_hm("Sept 10, 1990, 09:12am"))
map(times, sun_or_moon)
[[1]]
[1] "moon in the sky"

[[2]]
[1] "sun in the sky"
```

- Equivalently:

```
list(mdy_hm("Jan 2, 2000, 10:55pm"),
     mdy_hm("Sept 10, 1990, 09:12am")) %>%
map(sun_or_moon)
[[1]]
[1] "moon in the sky"

[[2]]
[1] "sun in the sky"
```

Names are preserved through map

- If the input list (or vector) has names, the output will have elements with the same names

```
list(
  random_day = mdy_hm("Jan 2, 2000, 10:55pm"),
  my_birthday = mdy_hm("Sept 10, 1990, 09:12am")) %>%
map(sun_or_moon)
$random_day
[1] "moon in the sky"

$my_birthday
[1] "sun in the sky"
```

- You can also dynamically set names in a list using `set_names()`

```
list(
  mdy_hm("Jan 2, 2000, 10:55pm"),
  mdy_hm("Sept 10, 1990, 09:12am")) %>%
set_names(c("random_day", "my_birthday")) %>%
map(sun_or_moon)
$random_day
[1] "moon in the sky"

$my_birthday
[1] "sun in the sky"
```

Returning vectors

- `map()` typically returns a list (why?)
- But there are variants that return vectors of different types

```
list(mdy_hm("Jan 2, 2000, 10:55pm"),
     mdy_hm("Sept 10, 1990, 09:12am")) %>%
map_chr(sun_or_moon)
[1] "moon in the sky" "sun in the sky"
```

```
harmonic_sum = function(n) {
  sum(1/(1:n)) # 1/1 + 1/2 + ... 1/n
}

1:10 %>%
  map_dbl(harmonic_sum)
 [1] 1.000000 1.500000 1.833333 2.083333 2.283333 2.450000 2.592857 2.717857
 [9] 2.828968 2.928968
```

Returning a data frame

```
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
) %>%
  mutate(d = if_else(row_number()==6, NA_real_, d)) # makes an element an NA
```

```
df %>%
  map_df(mean)
# A tibble: 1 x 4
        a     b      c     d
    <dbl> <dbl>  <dbl> <dbl>
1 -0.0944 0.187 -0.385    NA
```

- Why can we feed a data frame into `map()`?
- Notice the `c` column's mean is NA because we don't have a way to specify `na.rm=TRUE` to `mean()`

Composing functions inside map()

- How can we pass na.rm=TRUE to mean() inside of map?

- Define a new function with `na.rm=T` hardcoded in

```
mean_no_rm = function(x) mean(x,
na.rm=TRUE)
df %>% map_df(mean_no_rm)
# A tibble: 1 x 4
        a     b      c     d
    <dbl> <dbl>  <dbl> <dbl>
1 -0.0944 0.187 -0.385 0.240
```

- **Better**: define an anonymous function inside of map

```
df %>% map_df(function(x) mean(x,
na.rm=TRUE))
# A tibble: 1 x 4
        a     b      c     d
    <dbl> <dbl>  <dbl> <dbl>
1 -0.0944 0.187 -0.385 0.240
```

- **Better**: use the ~ . syntax to create an "anonymous" function inside of map. The ~ tells map that what follows is the body of a function (with no name) and a single argument called .

```
df %>% map_df(~ mean(., na.rm=TRUE))
# A tibble: 1 x 4
        a     b      c     d
    <dbl> <dbl>  <dbl> <dbl>
1 -0.0944 0.187 -0.385 0.240
```

- **Best**: any additional named arguments to map get passed on as named arguments to the function you want to call!

```
df %>% map_df(mean, na.rm=TRUE)
# A tibble: 1 x 4
        a     b      c     d
    <dbl> <dbl>  <dbl> <dbl>
1 -0.0944 0.187 -0.385 0.240
```

- In this case, the last option is the clearest, but in others it maybe a good idea to define a function outside of map or explicitly inside of map with the `function` syntax. It all depends on what makes the most sense for the situation.

Exercise: map practice

- Determine the type of each column in `nycflights13::flights` (`?typeof`). Return the result as a vector of strings
- Generate 10 normally-distributed random numbers for each of `mean`=-10, 0, 10, and 100 (?rnorm). Use the default value of `sd`.

Exercise: partial harmonic sum

```
harmonic_sum = function(n) {
  sum(1/(1:n)) # 1/1 + 1/2 + ... 1/n
}
```

- Combine the harmonic sum function with `map_dbl()` to create a new function that takes the same arguments but returns a vector of partial sums up to the full sum. i.e. instead of returning `1 + 1/2 + 1/3 ... + 1/n`, it returns `c(1, 1+1/2, 1+1/2,...1+1/2+...1/n)`.
- Accomplish the same thing by modifying the harmonic sum function. Do not use `map()`. The function `cumsum()` will be useful.
- Both of these give the same answer. Make an argument for which is better.

Exercise: partial harmonic sum

- Now use map() with `harmonic_partial_sum()` (your function from the previous exercise) to make a list where each element is a vector of partial sums from 1 to 100, but for values of `r=1`, `r=2` and `r=3` (each element of the list corresponds to the partial sums for a different value of r when `n=100`).

Mapping over multiple inputs

- So far we've mapped along a single input. But often you have multiple related inputs that you need iterate along in parallel. That's the job of `pmap()`. For example, imagine you want to simulate some random normals with different means. You know how to do that with `map()`:

```
means <- list(5, 10, -3)
means %>%
  map(rnorm, n = 5) %>%
  glimpse()
List of 3
 $ : num [1:5] 4.2 3.87 3.97 5.07 5.38
 $ : num [1:5] 8.38 11.9 9.28 10.38 10.44
 $ : num [1:5] -2.74 -3.18 -3.69 -3 -2.43
```

Mapping over multiple inputs

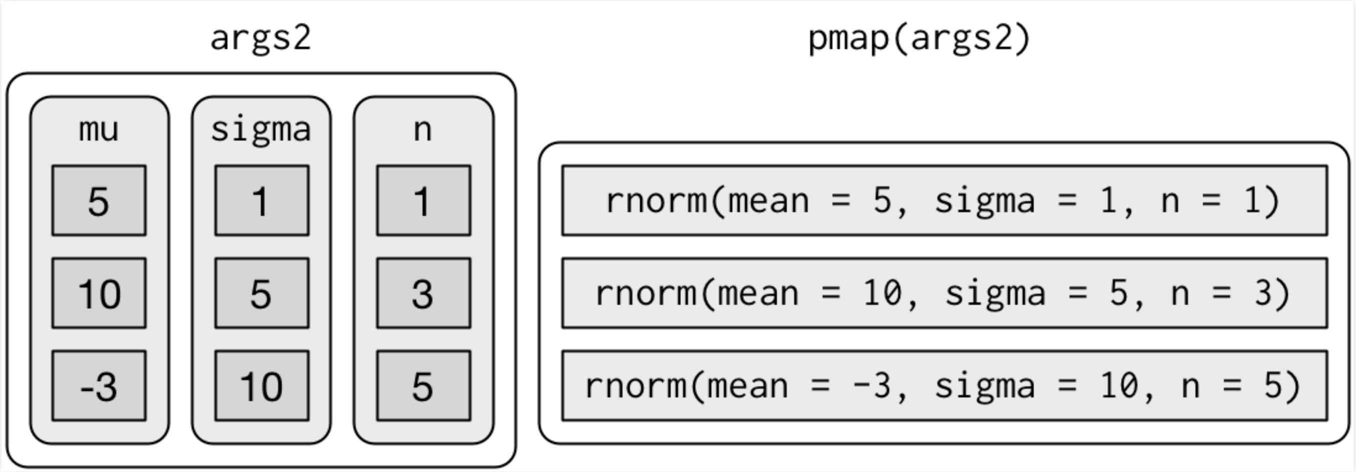
- What if you also want to vary the standard deviation and `n`? One way to do that would be to iterate over the indices and index into vectors of means and sds:

```
mean = list(5, 10, -3)
sd = list(1, 5, 10)
n = list(1, 3, 5)
1:3 %>%
  map(~rnorm(n[[.]], mean[[.]], sd[[.]])) %>%
  glimpse()
List of 3
 $ : num 3.79
 $ : num [1:3] 8.27 6.75 5.55
 $ : num [1:5] 11.77 -14.95 14.5 9.15 -18.48
```

- This looks like `C++`, not `R`. It's better to use `pmap()`

Mapping over multiple inputs

```
list(
  n = list(1, 3, 5),
  mean = list(5, 10, -3),
  sd = list(1, 5, 10)
) %>%
pmap(rnorm) %>%
glimpse()
List of 3
 $ : num 4.7
 $ : num [1:3] 15.2 6.16 17.62
 $ : num [1:5] -27.22 2.56 8.06 -1.34 -5.25
```



args2

mu: 5, 10, -3

sigma: 1, 5, 10

n: 1, 3, 5

pmap(args2)

```
rnorm(mean = 5, sigma = 1, n = 1)
rnorm(mean = 10, sigma = 5, n = 3)
rnorm(mean = -3, sigma = 10, n = 5)
```

Mapping over multiple inputs

- If you name the elements of the list, `pmap()` will match them to named arguments of the function you are calling

```r
list(
    to = c(1,2,3),
    from = c(10,11,12)
  ) %>%
  pmap(seq) %>%
  glimpse()
```

- Otherwise arguments will be passed positionally

```r
list(
    c(1,2,3),
    c(10,11,12)
  ) %>%
  pmap(seq) %>%
  glimpse()
```

Mapping over multiple inputs

- You can and should sometimes write functions inside of `pmap()`
- The same variants (e.g. `pmap_dbl()`) exist for `pmap()` as for `map()`

```
list(
    c(1,2,3),
    c(10,11,12)
  ) %>%
  pmap_dbl(function(x,y) {
    seq(x,y) %>%
      median()
  })
[1] 5.5 6.5 7.5
```

Cross

- `cross()` gives you every combintion of the items in the list you pass it

```
list(
    a = c(1,2,3),
    b = c(10,11)
  ) %>%
  cross() %>%
  glimpse()
List of 6
 $ :List of 2
  ..$ a: num 1
  ..$ b: num 10
 $ :List of 2
  ..$ a: num 2
  ..$ b: num 10
 $ :List of 2
  ..$ a: num 3
  ..$ b: num 10
 $ :List of 2
  ..$ a: num 1
  ..$ b: num 11
 $ :List of 2
  ..$ a: num 2
  ..$ b: num 11
 $ :List of 2
  ..$ a: num 3
  ..$ b: num 11
```

Cross

- `cross_df()` returns the result as a data frame

```
list(
    a = c(1,2,3),
    b = c(10,11)
  ) %>%
  cross_df()
# A tibble: 6 x 2
      a     b
  <dbl> <dbl>
1     1    10
2     2    10
3     3    10
4     1    11
5     2    11
6     3    11
```

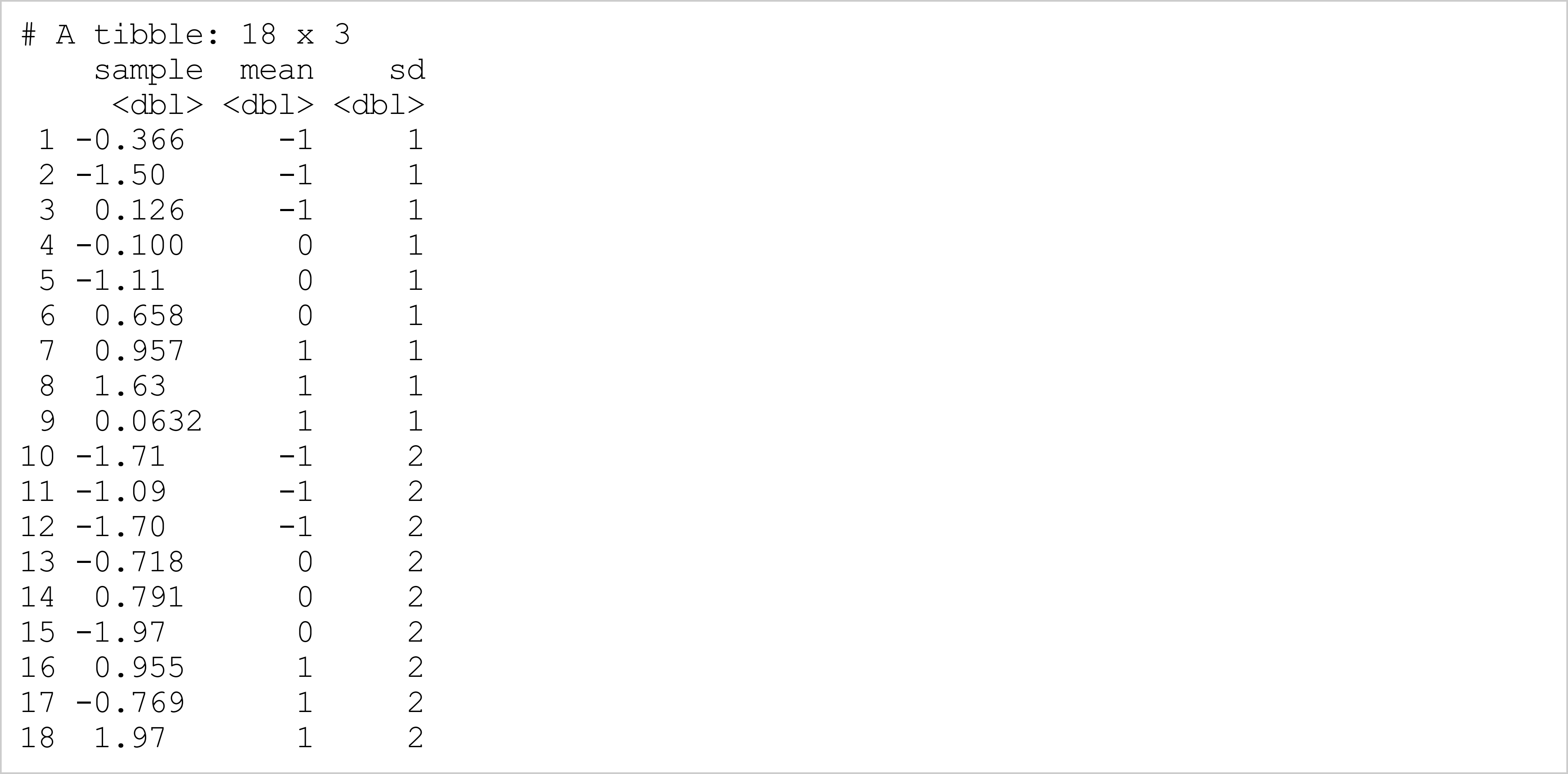
Binding rows

- `map()` and `pmap()` sometimes give you back a list of data frames. To glue them all together you will need `bind_rows()`:

```
1:3 %>%
  map(~tibble(
    rep = .,
    sample = rnorm(5)
  )) %>%
  bind_rows()
# A tibble: 15 x 2
      rep   sample
    <int>    <dbl>
 1      1 -0.228
 2      1 -0.253
 3      1  2.07
 4      1  1.58
 5      1 -1.04
 6      2 -0.00839
 7      2 -1.33
 8      2  0.147
 9      2 -0.788
10      2 -0.289
11      3  0.667
12      3 -0.137
13      3  0.224
14      3  1.12
15      3 -1.44
```

Exercise:

Generate a table like the one shown here by drawing 3 normally-distributed random numbers from every combination of the following parameters: `mean = c(-1, 0, 1),sd = c(1,2)`.



```
# A tibble: 18 x 3
    sample  mean    sd
     <dbl> <dbl> <dbl>
 1 -0.366     -1     1
 2 -1.50      -1     1
 3  0.126     -1     1
 4 -0.100      0     1
 5 -1.11       0     1
 6  0.658      0     1
 7  0.957      1     1
 8  1.63       1     1
 9  0.0632     1     1
10 -1.71      -1     2
11 -1.09      -1     2
12 -1.70      -1     2
13 -0.718      0     2
14  0.791      0     2
15 -1.97       0     2
16  0.955      1     2
17 -0.769      1     2
18  1.97       1     2
```

Why not for loops?

- R also provides something called a `for` loop, which is common to many other languages as well. It looks like this:

```
## generate 5 random numbers each from
3 normal distributions with different
means
samples = list(NA, NA, NA)
means = c(1,10,100)
for (i in 1:3) {
  samples[[i]] = rnorm(5, means[i])
}
```

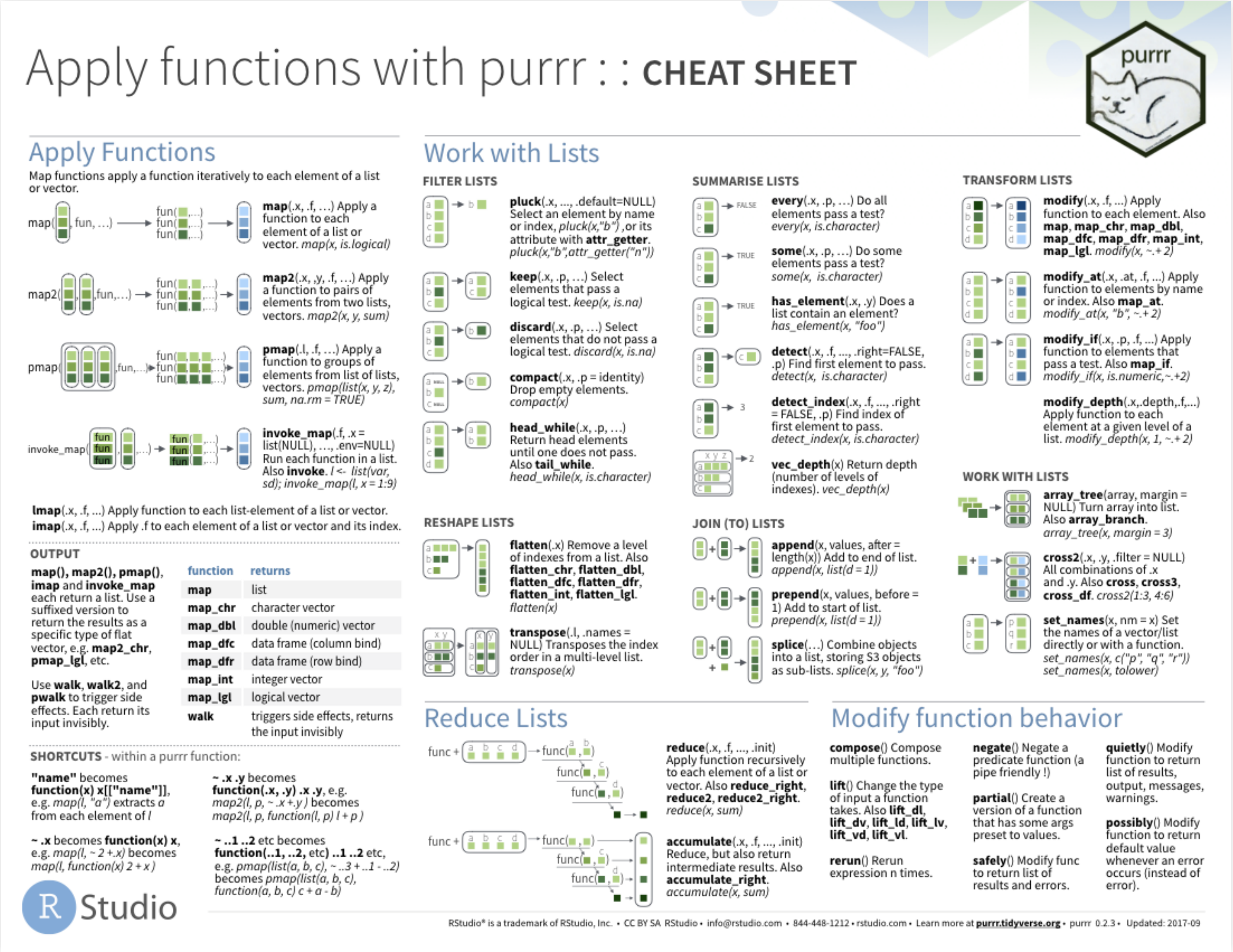
- The `for` loop is very flexible and you can do a lot with it
- `for` loops are unavoidable when the result of one iteration depends on the result of the last iteration

- Compare to the `map()` -style solution:

```
## generate 5 random numbers each from
3 normal distributions with different
means
samples = c(1,10,100) %>%
  map(~rnorm(5,.))
```

- Compared to the `for` loop, the `map()` syntax is much more concise and eliminates much of the “admin” code in the loop (setting up indices, initializing the list that will be filled in, indexing into the data structures)
- The `map()` syntax also encourages you to write a function for whatever is happening inside the loop. This means you have something that's reusable and easily testable, and your code will look cleaner
- Loops in R can be catastrophically slow due to the complexities of copy-on-modify semantics.

purrr Cheatsheet



Apply functions with purrr : : CHEAT SHEET

Apply Functions

Map functions apply a function iteratively to each element of a list or vector.

map(.x, .f, …) Apply a function to each element of a list or vector. *map(x, is.logical)*

map2(.x, ,y, .f, …) Apply a function to pairs of elements from two lists, vectors. *map2(x, y, sum)*

pmap(.l, .f, …) Apply a function to groups of elements from list of lists, vectors. *pmap(list(x, y, z), sum, na.rm = TRUE)*

invoke_map(.f, .x = list(NULL), …, .env=NULL) Run each function in a list. Also **invoke**. *l <- list(var, sd); invoke_map(l, x = 1:9)*

lmap(.x, .f, ...) Apply function to each list-element of a list or vector.
imap(.x, .f, ...) Apply .f to each element of a list or vector and its index.

OUTPUT

map(), map2(), pmap(), imap and **invoke_map** each return a list. Use a suffixed version to return the results as a specific type of flat vector, e.g. **map2_chr**, **pmap_lgl**, etc.

Use **walk**, **walk2**, and **pwalk** to trigger side effects. Each return its input invisibly.

function	returns
map	list
map_chr	character vector
map_dbl	double (numeric) vector
map_dfc	data frame (column bind)
map_dfr	data frame (row bind)
map_int	integer vector
map_lgl	logical vector
walk	triggers side effects, returns the input invisibly

SHORTCUTS - within a purrr function:

"name" becomes **function(x) x[["name"]]**, e.g. *map(l, "a")* extracts *a* from each element of *l*

~ .x becomes **function(x) x**, e.g. *map(l, ~ 2 +.x)* becomes *map(l, function(x) 2 + x)*

~ .x .y becomes **function(.x, .y) .x .y**, e.g. *map2(l, p, ~ .x +.y)* becomes *map2(l, p, function(l, p) l + p)*

~ ..1 ..2 etc becomes **function(..1, ..2,** etc**) ..1 ..2** etc, e.g. *pmap(list(a, b, c), ~ ..3 + ..1 - ..2)* becomes *pmap(list(a, b, c), function(a, b, c) c + a - b)*

RStudio

Work with Lists

FILTER LISTS

pluck(.x, ..., .default=NULL) Select an element by name or index, *pluck(x,"b")* ,or its attribute with **attr_getter**. *pluck(x,"b",attr_getter("n"))*

keep(.x, .p, …) Select elements that pass a logical test. *keep(x, is.na)*

discard(.x, .p, …) Select elements that do not pass a logical test. *discard(x, is.na)*

compact(.x, .p = identity) Drop empty elements. *compact(x)*

head_while(.x, .p, …) Return head elements until one does not pass. Also **tail_while**. *head_while(x, is.character)*

RESHAPE LISTS

flatten(.x) Remove a level of indexes from a list. Also **flatten_chr**, **flatten_dbl**, **flatten_dfc**, **flatten_dfr**, **flatten_int**, **flatten_lgl**. *flatten(x)*

transpose(.l, .names = NULL) Transposes the index order in a multi-level list. *transpose(x)*

SUMMARISE LISTS

every(.x, .p, …) Do all elements pass a test? *every(x, is.character)*

some(.x, .p, …) Do some elements pass a test? *some(x, is.character)*

has_element(.x, .y) Does a list contain an element? *has_element(x, "foo")*

detect(.x, .f, ..., .right=FALSE, .p) Find first element to pass. *detect(x, is.character)*

detect_index(.x, .f, ..., .right = FALSE, .p) Find index of first element to pass. *detect_index(x, is.character)*

vec_depth(x) Return depth (number of levels of indexes). *vec_depth(x)*

JOIN (TO) LISTS

append(x, values, after = length(x)) Add to end of list. *append(x, list(d = 1))*

prepend(x, values, before = 1) Add to start of list. *prepend(x, list(d = 1))*

splice(…) Combine objects into a list, storing S3 objects as sub-lists. *splice(x, y, "foo")*

TRANSFORM LISTS

modify(.x, .f, ...) Apply function to each element. Also **map**, **map_chr**, **map_dbl**, **map_dfc**, **map_dfr**, **map_int**, **map_lgl**. *modify(x, ~.+ 2)*

modify_at(.x, .at, .f, ...) Apply function to elements by name or index. Also **map_at**. *modify_at(x, "b", ~.+ 2)*

modify_if(.x, .p, .f, ...) Apply function to elements that pass a test. Also **map_if**. *modify_if(x, is.numeric,~.+2)*

modify_depth(.x,.depth,.f,...) Apply function to each element at a given level of a list. *modify_depth(x, 1, ~.+ 2)*

WORK WITH LISTS

array_tree(array, margin = NULL) Turn array into list. Also **array_branch**. *array_tree(x, margin = 3)*

cross2(.x, .y, .filter = NULL) All combinations of .x and .y. Also **cross**, **cross3**, **cross_df**. *cross2(1:3, 4:6)*

set_names(x, nm = x) Set the names of a vector/list directly or with a function. *set_names(x, c("p", "q", "r"))* *set_names(x, tolower)*

Reduce Lists

reduce(.x, .f, ..., .init) Apply function recursively to each element of a list or vector. Also **reduce_right**, **reduce2**, **reduce2_right**. *reduce(x, sum)*

accumulate(.x, .f, ..., .init) Reduce, but also return intermediate results. Also **accumulate_right**. *accumulate(x, sum)*

Modify function behavior

compose() Compose multiple functions.

lift() Change the type of input a function takes. Also **lift_dl**, **lift_dv**, **lift_ld**, **lift_lv**, **lift_vd**, **lift_vl**.

rerun() Rerun expression n times.

negate() Negate a predicate function (a pipe friendly !)

partial() Create a version of a function that has some args preset to values.

safely() Modify func to return list of results and errors.

quietly() Modify function to return list of results, output, messages, warnings.

possibly() Modify function to return default value whenever an error occurs (instead of error).

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **purrr.tidyverse.org** • purrr 0.2.3 • Updated: 2017-09

R in Parallel

Motivation

- Computers run programmers on processors
- Each processor can only do one thing at a time
- If there are many things to do, it will take a long time
- If the tasks are independent of each other, we can use multiple processors operating in *parallel*
- Parallel maps are implemented in the `furrr` package

Example: Permutation test of a regression coefficient

- Let's say we're interested in the association of `mpg` and `hp` in the mtcars data set (adjusted for other variables)
- We can run a linear model to estimate that association, assuming a linear trend

```
lm(mpg~., data=mtcars) %>%
  broom::tidy() %>%
  filter(term=="hp")
# A tibble: 1 x 5
  term  estimate std.error statistic p.value
  <chr>    <dbl>     <dbl>     <dbl>   <dbl>
1 hp     -0.0215    0.0218    -0.987   0.335
```

- Here we've used `lm()` to run a linear model- you can learn more about `lm()` and the formula interface it uses by looking at the help page
- We've also used `tidy()` from the `broom` package to clean up the output into a tibble.
- We could use this p-value, but since this is from the output of `lm()`, it is only valid if we believe that the noise in the model is actually normally distributed
- We can use a **permutation test** to get an estimate that does not rely on this assumption

Example: Permutation test of a regression coefficient

- The idea of the permutation test is that you shuffle the outcome differently to produce many different permutations of the data
- This destroys any associations with the outcome in the permuted datasets, but preserves the correlations within the covariates
- The analysis is then performed on the permuted data to get a set of coefficients that might have been estimated, given that there is no association
- The true estimated coefficient is thus compared to these. If it is of the same magnitude, we can say we are likely to have gotten such a value by chance even if there was no association

Example: Permutation test of a regression coefficient

- These are the functions we'll need

```
shuffle_row = function(data, row) {
  data %>% mutate({{row}} := sample({{row}}, nrow(data)))
}

analyze = function(data) {
  lm(mpg~., data=data) %>%
    broom::tidy() %>%
    filter(term=="hp") %>%
    pull(estimate)
}
```

```
mtcars %>%
  shuffle_row(mpg) %>%
  analyze()
[1] 0.05180838

mtcars %>%
  shuffle_row(mpg) %>%
  analyze()
[1] -0.009589738
```

Example: Permutation test of a regression coefficient

```
library(tictoc) # for timing things easily
```

```
tic()
perm_distribution = 1:1000 %>%
  map_dbl(~
    mtcars %>%
      shuffle_row(mpg) %>%
      analyze()
  )
toc()
14.848 sec elapsed
```

- We made 1000 coefficients, but it's slow!

Example: Permutation test of a regression coefficient

- If we parallelize across CPUs, we can go faster

```
library(furrr) # parallel maps
plan(multiprocess(workers=4)) # turns on parallel processing
```

```
tic()
perm_distribution = 1:1000 %>%
  future_map_dbl(~
    mtcars %>%
      shuffle_row(mpg) %>%
      analyze()
  )
toc()
10.147 sec elapsed
```

Example: Permutation test of a regression coefficient

```
beta = analyze(mtcars)

p_empirical = mean(abs(perm_distribution) >= abs(beta))

p_parametric = lm(mpg~., data=mtcars) %>%
  broom::tidy() %>%
  filter(term=="hp") %>%
  pull(p.value)
```

```
p_empirical
[1] 0.676
p_parametric
[1] 0.3349553
```

- Under the assumptions of the permutation test, we can be even less sure there is an association.

Using future_map()

- `furrr` implements all of the functions from `purrr` with a prefix `future_` that allows them to be used in parallel.
  - A “future” is a programming abstraction for a value that will be available in the future, which is one way of allowing for asynchronous (parallel) processes.
  - `furrr` uses futures under the hood to do what it does, therefore the naming convention
- The `future_` versions of `map()` work exactly the same as the standard `map()` versions.
- To turn on parallelism, simply put `plan(multiprocess(workers=K))` at the top of your script and all `future_` maps will be run in parallel across the number of workers that you specify
  - You can't exceed the number of available cores