

Functional programming and iteration with purrr

2021-01-24

purrr: A functional programming toolkit for R



Complete and consistent set of tools for working with functions and vectors

Problems we want to solve:

- 1 Making code clear
- 2 Making code safe
- 3 Working with lists and data frames

Lists, vectors, and data.frames (or tibbles)

```
c(char = "hello", num = 1)
```

```
##      char      num  
## "hello"      "1"
```

lists can contain any object

```
list(char = "hello", num = 1, fun = mean)
```

```
## $char
## [1] "hello"
##
## $num
## [1] 1
##
## $fun
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x7fc191985ee8>
## <environment: namespace:base>
```

Your Turn 1

```
measurements <- list(  
  blood_glucose = rnorm(10, mean = 140, sd = 10),  
  age = rnorm(5, mean = 40, sd = 5),  
  heartrate = rnorm(20, mean = 80, sd = 15)  
)
```

There are two ways to subset lists: dollar signs and brackets. Try to subset blood_glucose from measurements using these approaches. Are they different? What about measurements[["blood_glucose"]]?

Your Turn 1

```
measurements["blood_glucose"]
```

```
## $blood_glucose
## [1] 132.6953 140.3573 141.1298 154.2855 149.8340 133.7754 132.6846
## [8] 134.8333 122.4927 148.8010
```

```
measurements$blood_glucose
```

```
## [1] 132.6953 140.3573 141.1298 154.2855 149.8340 133.7754 132.6846
## [8] 134.8333 122.4927 148.8010
```

```
measurements[["blood_glucose"]]
```

```
## [1] 132.6953 140.3573 141.1298 154.2855 149.8340 133.7754 132.6846
## [8] 134.8333 122.4927 148.8010
```

data frames are lists

```
x <- list(char = "hello", num = 1)  
as.data.frame(x)
```

```
##      char num  
## 1 hello   1
```


data frames are lists

```
library(gapminder)  
head(gapminder$pop)
```

```
## [1] 8425333 9240934 10267083 11537966 13079460 14880372
```

data frames are lists

```
gapminder[1:6, "pop"]
```

data frames are lists

```
gapminder[1:6, "pop"]
```

```
## # A tibble: 6 x 1
##       pop
##   <int>
## 1  8425333
## 2  9240934
## 3 10267083
## 4 11537966
## 5 13079460
## 6 14880372
```

data frames are lists

```
head(gapminder[["pop"]])
```

```
## [1] 8425333 9240934 10267083 11537966 13079460 14880372
```

vectorized functions don't work on lists

```
sum(rnorm(10))
```

vectorized functions don't work on lists

```
sum(rnorm(10))
```

```
## [1] -3.831574
```

vectorized functions don't work on lists

```
sum(rnorm(10))
```

```
## [1] -3.831574
```

```
sum(list(x = rnorm(10), y = rnorm(10), z = rnorm(10)))
```

vectorized functions don't work on lists

```
sum(rnorm(10))
```

```
## [1] -3.831574
```

```
sum(list(x = rnorm(10), y = rnorm(10), z = rnorm(10)))
```

```
## Error in sum(list(x = rnorm(10), y = rnorm(10), z = rnorm(10))): invalid 'type' (list) of argument
```


map(.x, .f)

.x: a vector, list, or data frame

.f: a function

Returns a list



Using map()

```
library(purrr)
x_list <- list(x = rnorm(10), y = rnorm(10), z = rnorm(10))
map(x_list, mean)
```

Using map()

```
library(purrr)
x_list <- list(x = rnorm(10), y = rnorm(10), z = rnorm(10))
map(x_list, mean)
```

Using map()

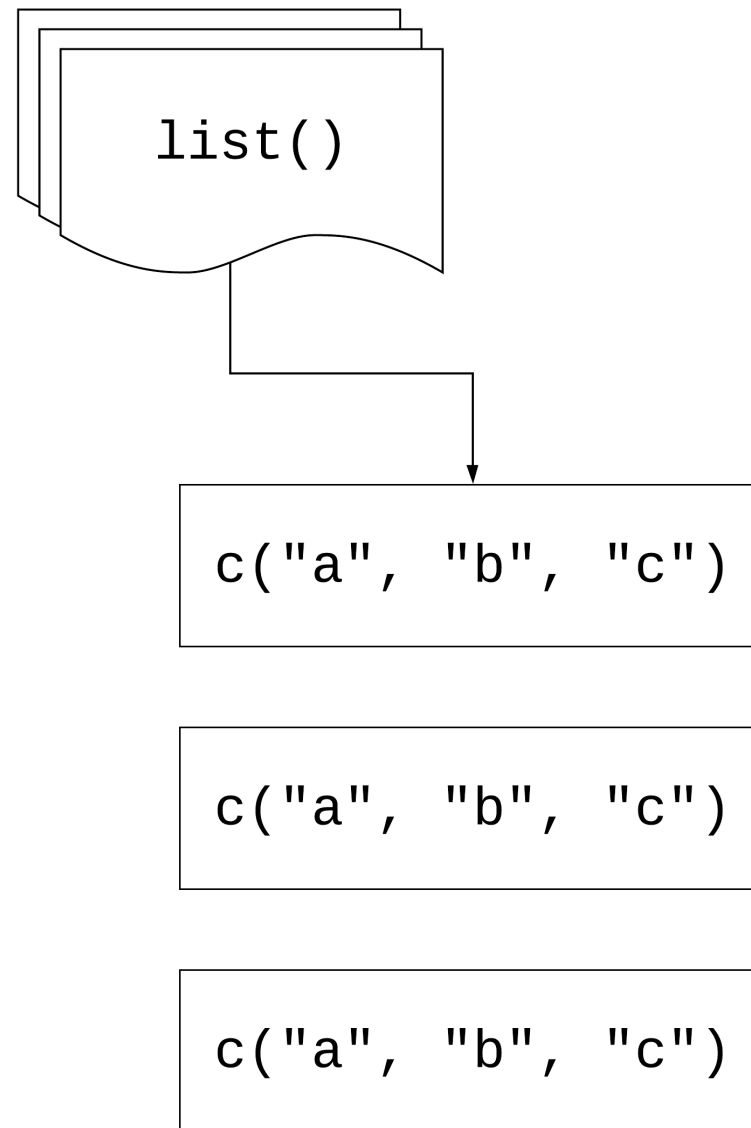
```
library(purrr)  
x_list <- list(x = rnorm(10), y = rnorm(10), z = rnorm(10))  
map(x_list, mean)
```


Using map()

```
library(purrr)
x_list <- list(x = rnorm(10), y = rnorm(10), z = rnorm(10))

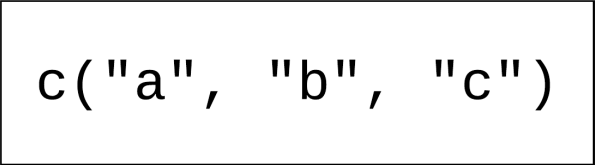
map(x_list, mean)
```

```
## $x
## [1] -0.6097971
##
## $y
## [1] -0.2788647
##
## $z
## [1] 0.6165922
```

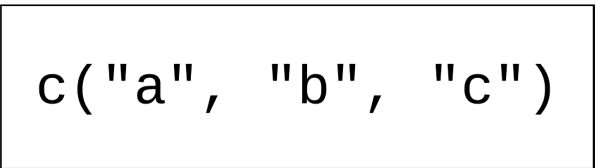


map( , .f)

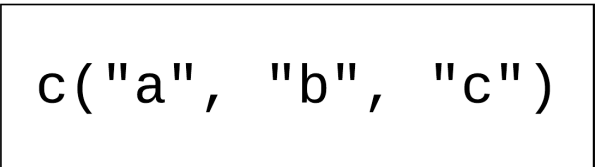
The diagram shows a stack of three overlapping rectangular boxes. The top box is labeled 'list()'. A line extends from the bottom of the stack, turns right, and then down to an arrowhead pointing towards the first function call below.

.f()

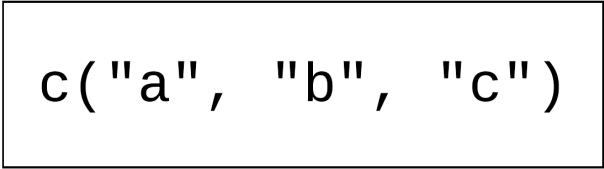
The diagram shows a rectangular box containing the text 'c("a", "b", "c")'.


.f()

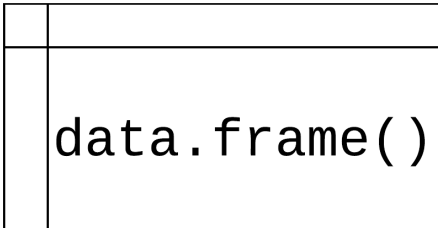
The diagram shows a rectangular box containing the text 'c("a", "b", "c")'.

.f()

The diagram shows a rectangular box containing the text 'c("a", "b", "c")'.

`map(`  `,` `.f)`

`map(`  `,` `.f)`

`map(`  `,` `.f)`

Your Turn 2

Read the code in the first chunk and predict what will happen

Run the code in the first chunk. What does it return?

```
list(  
  blood_glucose = sum(measurements$blood_glucose),  
  age = sum(measurements$age),  
  heartrate = sum(measurements$heartrate)  
)
```

Now, use `map()` to create the same output.

Your Turn 2

```
map(measurements, sum)
```

```
## $blood_glucose
```

```
## [1] 1390.889
```

```
##
```

```
## $age
```

```
## [1] 198.894
```

```
##
```

```
## $heartrate
```

```
## [1] 1625.451
```

using map() with data frames

```
library(dplyr)
gapminder %>%
  select(where(is.numeric)) %>%
  map(sd)
```

using map() with data frames

```
library(dplyr)
gapminder %>%
  select(where(is.numeric)) %>%
  map(sd)
```

using map() with data frames

```
library(dplyr)
gapminder %>%
  select(where(is.numeric)) %>%
  map(sd)
```

using map() with data frames

```
library(dplyr)
gapminder %>%
  select(where(is.numeric)) %>%
  map(sd)
```

```
## $year
## [1] 17.26533
##
## $lifeExp
## [1] 12.91711
##
## $pop
## [1] 106157897
##
## $gdpPercap
## [1] 9857.455
```

Your Turn 3

Pass diabetes to `map()` and map using `class()`. What are these results telling you?

Your Turn 3

```
head(  
  map(diabetes, class),  
  3  
)
```

```
## $id  
## [1] "numeric"  
##  
## $chol  
## [1] "numeric"  
##  
## $stab.glu  
## [1] "numeric"
```


Review: writing functions

```
x <- x^2  
x <- scale(x)  
x <- max(x)
```

Review: writing functions

```
x <- x^2
x <- scale(x)
x <- max(x)

y <- x^2
y <- scale(y)
y <- max(y)

z <- z^2
z <- scale(x)
z <- max(z)
```

Review: writing functions

```
x <- x^2  
x <- scale(x)  
x <- max(x)
```

```
y <- x^2  
y <- scale(y)  
y <- max(y)
```

```
z <- z^2  
z <- scale(x)  
z <- max(z)
```

Review: writing functions

```
x <- x^3  
x <- scale(x)  
x <- max(x)
```

```
y <- x^2  
y <- scale(y)  
y <- max(y)
```

```
z <- z^2  
z <- scale(x)  
z <- max(z)
```

Review: writing functions

```
.f <- function(x) {  
  x <- x^3  
  x <- scale(x)  
  
  max(x)  
}  
  
.f(x)  
.f(y)  
.f(z)
```

**If you copy and paste your code
three times, it's time to write a
function**

Your Turn 4

Write a function that returns the mean and standard deviation of a numeric vector.

Give the function a name

Find the mean and SD of x

Map your function to measurements

Your Turn 4

```
mean_sd <- function(x) {  
  x_mean <- mean(x)  
  x_sd <- sd(x)  
  tibble(mean = x_mean, sd = x_sd)  
}  
  
map(measurements, mean_sd)
```


Your Turn 4

```
## $blood_glucose
## # A tibble: 1 x 2
##   mean    sd
##   <dbl> <dbl>
## 1  139.   9.72
##
## $age
## # A tibble: 1 x 2
##   mean    sd
##   <dbl> <dbl>
## 1  39.8   5.92
##
## $heartrate
## # A tibble: 1 x 2
##   mean    sd
##   <dbl> <dbl>
## 1  81.3  12.8
```

Three ways to pass functions to `map()`

- 1 **pass directly to `map()`**
- 2 **use an anonymous function**
- 3 **use `~`**

```
map(  
  .X,  
  mean,  
  na.rm = TRUE  
)
```

```
map(  
  .x,  
  function(.x) {  
    mean(.x,  
      na.rm = TRUE)  
  }  
)
```

```
map(  
  .x,  
  ~mean(.x,  
    na.rm = TRUE)  
)
```

```
map(gapminder, ~length(unique(.x)))
```

```
map(gapminder, ~length(unique(.x)))
```

```
## $country  
## [1] 142  
##  
## $continent  
## [1] 5  
##  
## $year  
## [1] 12  
##  
## $lifeExp  
## [1] 1626  
##  
## $pop  
## [1] 1704  
##  
## $gdpPercap  
## [1] 1704
```

Returning types

map	returns
<code>map()</code>	list
<code>map_chr()</code>	character vector
<code>map_dbl()</code>	double vector (numeric)
<code>map_int()</code>	integer vector
<code>map_lgl()</code>	logical vector
<code>map_dfc()</code>	data frame (by column)
<code>map_dfr()</code>	data frame (by row)

Returning types

```
map_int(gapminder, ~length(unique(.x)))
```

Returning types

```
map_int(gapminder, ~length(unique(.x)))
```

##	country	continent	year	lifeExp	pop	gdpPercap
##	142	5	12	1626	1704	1704

Your Turn 5

Do the same as #3 above but return a vector instead of a list.

Your Turn 5

```
map_chr(diabetes, class)
```

```
##           id           chol   stab.glu           hdl           ratio           glyhb
##  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##    location           age      gender      height      weight      frame
## "character" "numeric" "character" "numeric" "numeric" "character"
##         bp.1s         bp.1d         bp.2s         bp.2d         waist         hip
##  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##    time.ppn
##  "numeric"
```

Your Turn 6

Check diabetes for any missing data.

Using the `~f(.x)` shorthand, check each column for any missing values using `is.na()` and `any()`

Return a logical vector. Are any columns missing data? What happens if you don't include `any()`? Why?

Try counting the number of missing, returning an integer vector

Your Turn 6

```
map_lgl(diabetes, ~any(is.na(.x)))
```

##	id	chol	stab.glu	hdl	ratio	glyhb	location	age
##	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE
##	gender	height	weight	frame	bp.1s	bp.1d	bp.2s	bp.2d
##	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
##	waist	hip	time.ppn					
##	TRUE	TRUE	TRUE					

Your Turn 6

```
map_int(diabetes, ~sum(is.na(.x)))
```

##	id	chol	stab.glu	hdl	ratio	glyhb	location	age
##	0	1	0	1	1	13	0	0
##	gender	height	weight	frame	bp.1s	bp.1d	bp.2s	bp.2d
##	0	5	1	12	5	5	262	262
##	waist	hip	time.ppn					
##	2	2	3					

Your Turn 7

Turn diabetes **into a list split by** location **using the** `split()` **function. Check its length.**

Fill in the `model_lm` **function to model** `chol` **(the outcome) with** `ratio` **and pass the** `.data` **argument to** `lm()`

map `model_lm` **to** `diabetes_list` **so that it returns a data frame (by row).**

Your Turn 7

```
diabetes_list <- split(diabetes, diabetes$location)
length(diabetes_list)
model_lm <- function(.data) {
  mdl <- lm(chol ~ ratio, data = .data)
  # get model statistics
  broom::glance(mdl)
}
map(diabetes_list, model_lm)
```

Your Turn 7

```
## [1] 2

## $Buckingham
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic p.value    df
##   <dbl>      <dbl> <dbl>    <dbl>    <dbl> <dbl>
## 1     0.252      0.248  38.8     66.4 4.11e-14     1
## # ... with 6 more variables: logLik <dbl>, AIC <dbl>,
## #   BIC <dbl>, deviance <dbl>, df.residual <int>,
## #   nobs <int>
##
## $Louisa
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic p.value    df
##   <dbl>      <dbl> <dbl>    <dbl>    <dbl> <dbl>
## 1     0.204      0.201  39.4     51.7 1.26e-11     1
## # ... with 6 more variables: logLik <dbl>, AIC <dbl>,
## #   BIC <dbl>, deviance <dbl>, df.residual <int>,
## #   nobs <int>
```

`map2(.x, .y, .f)`

.x, .y: a vector, list, or data frame

.f: a function

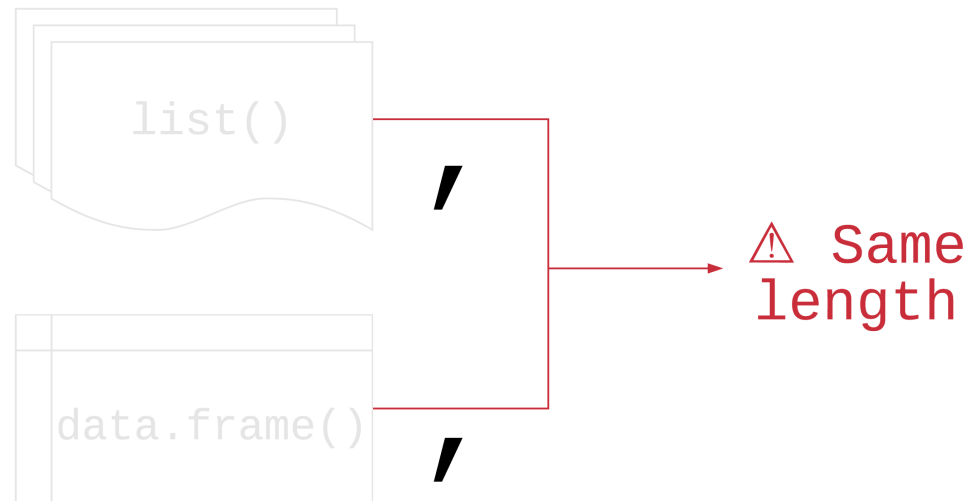
Returns a list

map2(



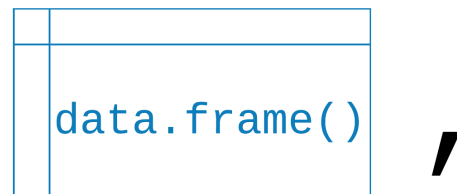
`.f`
`)`

map2(



.f
)

map2(



`~ .f(.x, .y)`
`)`

map2()

```
means <- c(-3, 4, 2, 2.3)
sds <- c(.3, 4, 2, 1)

map2_dbl(means, sds, rnorm, n = 1)
```

map2()

```
means <- c(-3, 4, 2, 2.3)
sds <- c(.3, 4, 2, 1)

map2_dbl(means, sds, rnorm, n = 1)
```


map2()

```
means <- c(-3, 4, 2, 2.3)
sds <- c(.3, 4, 2, 1)

map2_dbl(means, sds, rnorm, n = 1)
```

```
## [1] -2.997932  2.178125  1.266952  2.948287
```

Your Turn 8

Split the gapminder dataset into a list by country

Create a list of models using `map()`. For the first argument, pass `gapminder_countries`. For the second, use the `~.f()` notation to write a model with `lm()`. Use `lifeExp` on the left hand side of the formula and `year` on the second. Pass `.x` to the data argument.

Use `map2()` to take the models list and the data set list and map them to `predict()`. Since we're not adding new arguments, you don't need to use `~.f()`.

Your Turn 8

```
gapminder_countries <- split(gapminder, gapminder$country)
models <- map(gapminder_countries, ~ lm(lifeExp ~ year, data = .x))
preds <- map2(models, gapminder_countries, predict)
head(preds, 3)
```

Your Turn 8

```
gapminder_countries <- split(gapminder, gapminder$country)
models <- map(gapminder_countries, ~ lm(lifeExp ~ year, data = .x))
preds <- map2(models, gapminder_countries, predict)
head(preds, 3)
```

Your Turn 8

```
gapminder_countries <- split(gapminder, gapminder$country)
models <- map(gapminder_countries, ~ lm(lifeExp ~ year, data = .x))
preds <- map2(models, gapminder_countries, predict)
head(preds, 3)
```

Your Turn 8

\$Afghanistan

###	1	2	3	4	5	6
###	29.90729	31.28394	32.66058	34.03722	35.41387	36.79051

###

\$Albania

###	1	2	3	4	5	6
###	59.22913	60.90254	62.57596	64.24938	65.92279	67.59621

###

\$Algeria

###	1	2	3	4	5	6
###	43.37497	46.22137	49.06777	51.91417	54.76057	57.60697

input 1	input 2	returns
map()	map2()	list
map_chr()	map2_chr()	character vector
map_dbl()	map2_dbl()	double vector (numeric)
map_int()	map2_int()	integer vector
map_lgl()	map2_lgl()	logical vector
map_dfc()	map2_dfc()	data frame (by column)
map_dfr()	map2_dfr()	data frame (by row)

Other mapping functions

pmap() and friends: take n lists or data frame with argument names

Other mapping functions

`pmap()` and friends: take n lists or data frame with argument names

`walk()` and friends: for side effects like plotting; returns input invisibly

Other mapping functions

`pmap()` and friends: take n lists or data frame with argument names

`walk()` and friends: for side effects like plotting; returns input invisibly

`imap()` and friends: includes counter i

Other mapping functions

`pmap()` and friends: take n lists or data frame with argument names

`walk()` and friends: for side effects like plotting; returns input invisibly

`imap()` and friends: includes counter i

`map_if()`, `map_at()`: Apply only to certain elements

input 1	input 2	input n	returns
map()	map2()	pmap()	list
map_chr()	map2_chr()	pmap_chr()	character vector
map_dbl()	map2_dbl()	pmap_dbl()	double vector (numeric)
map_int()	map2_int()	pmap_int()	integer vector
map_lgl()	map2_lgl()	pmap_lgl()	logical vector
map_dfc()	map2_dfc()	pmap_dfc()	data frame (by column)
map_dfr()	map2_dfr()	pmap_dfr()	data frame (by row)
walk()	walk2()	pwalk()	input (side effects!)

Your turn 9

Create a new directory using the fs package. Call it "figures".

Write a function to plot a line plot of a given variable in gapminder over time, faceted by continent. Then, save the plot (how do you save a ggplot?). For the file name, paste together the folder, name of the variable, and extension so it follows the pattern "folder/variable_name.png"

Create a character vector that has the three variables we'll plot: "lifeExp", "pop", and "gdpPercap".

Use walk() to save a plot for each of the variables

Your turn 9

```
fs::dir_create("figures")

ggsave_gapminder <- function(variable) {
  # we're using `aes_string()` so we don't need the curly-curly syn
  p <- ggplot(
    gapminder,
    aes_string(x = "year", y = variable, color = "country")
  ) +
    geom_line() +
    scale_color_manual(values = country_colors) +
    facet_wrap(vars(continent.)) +
    theme(legend.position = "none")

  ggsave(
    filename = paste0("figures/", variable, ".png"),
    plot = p,
    dpi = 320
  )
}
```

Your turn 9

```
vars <- c("lifeExp", "pop", "gdpPercap")  
walk(vars, ggsave_gapminder)
```

Base R

base R	purrr
lapply()	map()
vapply()	map_*()
sapply()	?
x[] <- lapply()	map_dfc()
mapply()	map2(), pmap()

Benefits of purrr

- 1 Consistent
- 2 Type-safe
- 3 $\sim f(.x)$

Loops vs functional programming

```
x <- rnorm(10)
y <- map(x, mean)
```

```
x <- rnorm(10)
y <- vector("list", length(x))
for (i in seq_along(x)) {
  y[[i]] <- mean(x[[i]])
}
```

Loops vs functional programming

```
x <- rnorm(10)
y <- map(x, mean)
```

```
x <- rnorm(10)
y <- vector("list", length(x))
for (i in seq_along(x)) {
  y[[i]] <- mean(x[[i]])
}
```

Loops vs functional programming

```
x <- rnorm(10)
y <- map(x, mean)
```

```
x <- rnorm(10)
y <- vector("list", length(x))
for (i in seq_along(x)) {
  y[[i]] <- mean(x[[i]])
}
```

Loops vs functional programming

```
x <- rnorm(10)
y <- map(x, mean)
```

```
x <- rnorm(10)
y <- vector("list", length(x))
for (i in seq_along(x)) {
  y[[i]] <- mean(x[[i]])
}
```

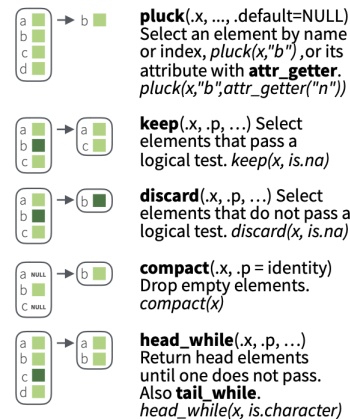
Of course someone has to write loops. It doesn't have to be you.

—Jenny Bryan

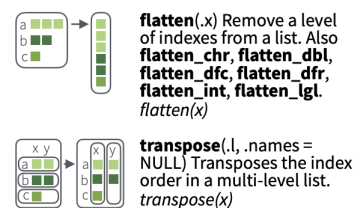
Working with lists and nested data

Work with Lists

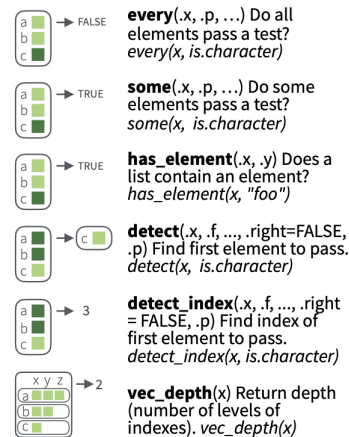
FILTER LISTS



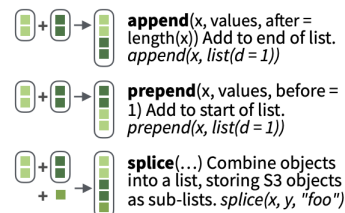
RESHAPE LISTS



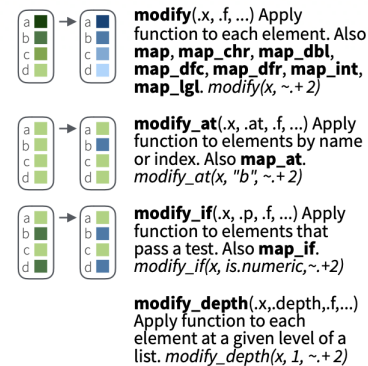
SUMMARISE LISTS



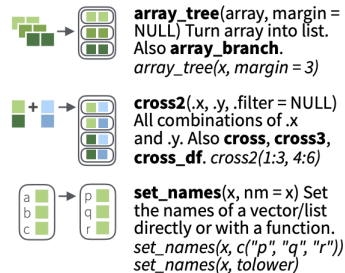
JOIN (TO) LISTS



TRANSFORM LISTS



WORK WITH LISTS



Working with lists and nested data

Nested Data

A **nested data frame** stores individual tables within the cells of a larger, organizing table.

nested data frame	
Species	data
setosa	<tibble [50 x 4]>
versicolor	<tibble [50 x 4]>
virginica	<tibble [50 x 4]>

n_iris

Use a nested data frame to:

- preserve relationships between observations and subsets of data

- manipulate many sub-tables at once with the **purrr** functions **map()**, **map2()**, or **pmap()**.

"cell" contents

Sepal.L	Sepal.W	Petal.L	Petal.W
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5.0	3.6	1.4	0.2

n_iris\$data[[1]]

Sepal.L	Sepal.W	Petal.L	Petal.W
7.0	3.2	4.7	1.4
6.4	3.2	4.5	1.5
6.9	3.1	4.9	1.5
5.5	2.3	4.0	1.3
6.5	2.8	4.6	1.5

n_iris\$data[[2]]

Sepal.L	Sepal.W	Petal.L	Petal.W
6.3	3.3	6.0	2.5
5.8	2.7	5.1	1.9
7.1	3.0	5.9	2.1
6.3	2.9	5.6	1.8
6.5	3.0	5.8	2.2

n_iris\$data[[3]]

List Column Workflow

Nested data frames use a **list column**, a list that is stored as a column vector of a data frame. A typical **workflow** for list columns:



1 Make a list column

Species	S.L	S.W	P.L	P.W
setosa	5.1	3.5	1.4	0.2
setosa	4.9	3.0	1.4	0.2
setosa	4.7	3.2	1.3	0.2
setosa	4.6	3.1	1.5	0.2
setosa	5.0	3.6	1.4	0.2

```
n_iris <- iris %>%
  group_by(Species) %>%
  nest()
```

2 Work with list columns

Species	data	model
setosa	<tibble [50 x 4]>	<S3: lm>
versicolor	<tibble [50 x 4]>	<S3: lm>
virginica	<tibble [50 x 4]>	<S3: lm>

```
mod_fun <- function(df)
  lm(Sepal.Length ~ ., data = df)

m_iris <- n_iris %>%
  mutate(model = map(data, mod_fun))
```

3 Simplify the list column

Species	beta
setosa	2.35
versicolor	1.89
virginica	0.69

```
b_fun <- function(mod)
  coefficients(mod)[1]

m_iris %>% transmute(Species,
  beta = map_dbl(model, b_fun))
```

1. MAKE A LIST COLUMN - You can create list columns with functions in the **tibble** and **dplyr** packages, as well as **tidyr**'s **nest()**

tibble::tibble(...)

tibble::tibble(...)

dplyr::mutate(data, ...) Also **transmute()**

Adverbs: Modify function behavior

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).

Learn more!

Jenny Bryan's purrr tutorial: A detailed introduction to purrr. Free online.

R for Data Science: A comprehensive but friendly introduction to the tidyverse. Free online.

RStudio Primers: Free interactive courses in the Tidyverse