purr beyond map()

(function)ctional programming in R

result <- purrr::modify(.x = ...,
                         .f = ...)

Hendrik van Broekhuizen
Predictive Insights
2020-03-07
@hendrikvanb
hendrik@predictiveinsights.net
The obligatory preamble
Making sure we are all on the same page

Disclaimer
- purrrfantatic → purrr expert
- 15min ≠ enough time

Admissions
- I’m an extreme centrist w.r.t. tidyverse and data.table
- I love to %>%

Setup
- Working in RStudio in an .Rproj context
- Using same set of packages throughout
- Using the mpg dataset (ggplot2) throughout

```r
library(data.table)
library(tidyverse)
```
The `purrr` package
what is it and why should I care?

A complete and consistent functional programming toolkit for R
- `help(purrr)`

... to give you similar expressiveness to a classical FP language, while allowing you to write code that looks and feels like R
- `purrr 0.1.0`
The `purrr` package
what is it and why should I care?

A complete and consistent functional programming toolkit for R
- `help(purrr)`

... to give you similar expressiveness to a classical FP language, while allowing you to write code that looks and feels like R
- `purrr 0.1.0`

`map()` is the posterchild, but Narnia lies beyond
- Other functions get less press
  - Terse official documentation
  - Lack of package vignettes
  - Few "deep dive" tutorials and resources online

Things get good when you dive in
- `purrr` offers one of the highest rates of return on investment for any R package
**purrr**: what is it good for?

Absolutely nothing everything lots of stuff

Iterative tasks
- `lapply++`
  - more consistent, more general, more powerful

Working with lists
- Yes, even complex, nested lists
- It’s lists, all the way down

Creating consistent, robust functions/routines
- Consistent syntax
- Fail loudly
- Nice error handling
Some tips
useful things to keep in mind when using purrr

When not to use purrr
- `lapply()` is the base equivalent to `map()` (sans purrr helpers support)
  - if you’re only using `map()` from purrr, you can skip the additional dependency and use `lapply()` directly
  - there is no need to map if the operation is already appropriately vectorised

Avoiding nasty surprises
- `map*()` and `modify()` functions always return output of the same length as the input

Never forget
- a data frame is simply a list of [consistently typed] vectors of equal length
A map() primer

map( , f) → f()
  f()
  f()
  f()
map()

Apply to all

\[
\text{map(.x, .f)}
\]

- call function \( .f \) once for each element of vector \( .x \);
- return the result as a list
**map()**

Apply to all

map(.x, .f)

- call function `.f` once for each element of vector `.x`
- return the result as a list

Get the square of each number from 1 to 5

```r
# function to get square of number
my_square <- function(x) x^2

# get square of each number 1:5 and output as list
res1 <- 1:5 %>% map(my_square)  # direct call
res2 <- 1:5 %>% map(~my_square(.))  # for backward compatibility
res3 <- 1:5 %>% map(~my_square(.x))  # formula
res4 <- 1:5 %>% map(function(x) my_square(x))  # inline anonymous function

# test equivalence
identical(res1, res2) & identical(res2, res3) & identical(res3, res4)
```

[1] TRUE
Passing arguments with . . .

Many ways to do the same thing

map(.x, .f, ...)
passes arguments specified in ... along
Passing arguments with . . .

Many ways to do to the same thing

\[
\text{map}(\text {.x, .f, . . .})
\]

passes arguments specified in ... along

Use \text{paste() } to add 'min' as suffix to each number from 1 to 5

```r
# pass arguments along
spec1 <- 1:5 %>% map(paste, 'min')

# formula specification (two variants)
spec2 <- 1:5 %>% map(~paste(.,'min'))
spec3 <- 1:5 %>% map(~paste(.x,'min'))

# inline anonymous function specification
spec4 <- 1:5 %>% map(function(x) paste(x, 'min'))

# test equivalence
list(spec2, spec3, spec4) %>% map_lgl(identical, y = spec1)
```

[1] TRUE TRUE TRUE

\[purrr \text{ beyond } \text{map() }\]
Passing arguments: via . . . vs in function

A seemingly subtle, yet important difference

Not all that seems vectorised is...
- `map()` is only vectorised over its first argument so arguments passed to `map()` after `.f` will be
  - passed along as is and
  - evaluated once

What is that supposed to mean?
- Has implications if you pass arguments to function via . . .
  - errors if you pass vectors as arguments to functions that do not accept vectors as arguments
  - potentially wrong results even if arguments specified correctly
Passing arguments: via ... vs in function
A seemingly subtle, yet important difference

Not all that seems vectorised is...
- `map()` is only vectorised over its first argument so arguments passed to `map()` after `.f` will be
  - passed along as is and
  - evaluated once

What is that supposed to mean?
- Has implications if you pass arguments to function via ...
  - errors if you pass vectors as arguments to functions that do not accept vectors as arguments
  - potentially wrong results even if arguments specified correctly

```r
# function that multiplies input (arg1) by specified constant (arg2)
temp_func <- function(x, constant = 2) {
  glue::glue('{x} x {constant} = {x*constant}')
}

# method 1: pass parameterised arg2 directly to map_chr
1:5 %>% map_chr(temp_func, constant = sample(1:10, 1))
# method 2: pass parameterised arg2 into inline anonymous function
1:5 %>% map_chr(function(x) temp_func(x, constant = sample(1:10, 1)))
```

[1] "1 x 2 = 2"  "2 x 2 = 4"  "3 x 2 = 6"  "4 x 2 = 8"  "5 x 2 = 10"
[1] "1 x 8 = 8"  "2 x 1 = 2"  "3 x 7 = 21"  "4 x 7 = 28"  "5 x 7 = 35"
map_*( )

Specifying the output format

- map_chr(.x, .f): character
- map_lgl(.x, .f): logical
- map_dbl(.x, .f): real
- map_int(.x, .f): integer
- map_dfr(.x, .f): data frame (bind_rows)
- map_dfc(.x, .f): data frame (bind_cols)
map_*( )

Specifying the output format

map_*(.x, .f, ...)
call function .f once for each element of vector .x; return the result as an atomic vector of type *; error if impossible

- map_chr(.x, .f): character
- map_lgl(.x, .f): logical
- map_dbl(.x, .f): real
- map_int(.x, .f): integer
- map_dfr(.x, .f): data frame (bind_rows)
- map_dfc(.x, .f): data frame (bind_cols)

1:5 %>% map_chr(paste, 'min') %>% class()
[1] "character"

1:5 %>% map_lgl(function(x) x < 3) %>% class()
[1] "logical"

1:5 %>% map_int(function(x) x * 2L) %>% class()
[1] "integer"

1:5 %>% map_dfr(function(x) tibble(value = x)) %>% class()
[1] "tbl_df" "tbl" "data.frame"

1:5 %>% map_dfc(function(x) data.table(value = x)) %>% class()
[1] "data.table" "data.frame"
Map variants
walk() and modify()

map() has siblings...

- walk(.x, .f, ...)  
  call function .f once for each element of .x; return nothing

- modify(.x, .f, ...)  
  call function .f once for each element of .x; return the result as an object of the same type as .x
walk() and modify()
map() has siblings...

walk(.x, .f, ...)
call function .f once for each element of .x; return nothing

modify(.x, .f, ...)
call function .f once for each element of .x; return the result as an object of the same type as .x

# no output
1:5 %>% walk(paste, 'min')

# output, but not what you might have expected
1:5 %>% walk(function(x) x ^ 2) %>% print()
[1] 1 2 3 4 5

# proof that walk is actually doing stuff
1:5 %>% walk(function(x) print(x ^ 2)) %>% print()
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 1 2 3 4 5

# obviously a character vector
x <- c('1', '2', '3', '4', '5')

# try to convert each element to integer using map_dbl
x %>% map_dbl(as.integer)
[1] 1 2 3 4 5

# try to convert each element to integer using modify
x %>% modify(as.integer)
[1] "1" "2" "3" "4" "5"
Why `walk()`? Why `modify()`?

What's the point?

- **walk(.x, .f, ...)**
  - call function `.f` once for each element of `.x`; return nothing

- **modify(.x, .f, ...)**
  - call function `.f` once for each element of `.x`; return the result as an object of the same type as `.x`

### Just do stuff
- Some functions just need to do stuff, not necessarily return stuff
  - E.g.: `cat()`, `message()`, `saveRDS()`, etc
  - Particularly useful for disk I/O operations
  - Allows input "passthrough"

### Change the content; keep the wrapper
- Some functions just need to change stuff, not necessarily create stuff
  - Not everything needs to be coerced
    - What if input is already of the type we want as output?
    - Type preservation can be essential
  - Particularly useful are the `modify_if()` and `modify_at()` variants
map( ) variants cheatsheet
Basic rules & the matrix of understanding

map variant rules:
1. `map()` returns list; `map_*()` returns vector of type specified
2. `modify()` returns same type as input
3. `walk()` returns nothing
4. Iterate over two inputs with `map2(), walk2(), modify2()`
5. Iterate over input and index with `imap(), imodify(), iwalk()`
6. Iterate over any number of inputs with `pmap()` and `pwalk()`

map variant matrix:
- map family of functions has orthogonal input and outputs
- can organise all the family into a matrix, with inputs in the rows and outputs in the columns

<table>
<thead>
<tr>
<th>arguments</th>
<th>list</th>
<th>atomic</th>
<th>preserve type</th>
<th>nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>one argument</td>
<td><code>map()</code></td>
<td><code>map_lgl()</code>, ...</td>
<td><code>modify()</code></td>
<td><code>walk()</code></td>
</tr>
<tr>
<td>two arguments</td>
<td><code>map2()</code></td>
<td><code>map2_lgl()</code>, ...</td>
<td><code>modify2()</code></td>
<td><code>walk2()</code></td>
</tr>
<tr>
<td>one argument + index</td>
<td><code>imap()</code></td>
<td><code>imap_lgl()</code>, ...</td>
<td><code>imodify()</code></td>
<td><code>iwalk()</code></td>
</tr>
<tr>
<td>n arguments</td>
<td><code>pmap()</code></td>
<td><code>pmap_lgl()</code>, ...</td>
<td>NA</td>
<td><code>pwalk()</code></td>
</tr>
</tbody>
</table>
Using `walk()`

Iteratively write data to disk using `purrr::pwalk()`

For each manufacturer in the `mpg` dataset, write a `.csv` file to disk containing only the data for that manufacturer.
Using `walk()`

Iteratively write data to disk using `purrr::pwalk()`

For each manufacturer in the `mpg` dataset, write a `.csv` file to disk containing only the data for that manufacturer.

```r
# check for files (show that there are none)
list.files('data/mpg')
character(0)
```
Using `walk()`

Iteratively write data to disk using `purrr::pwalk()`

For each manufacturer in the `mpg` dataset, write a `.csv` file to disk containing only the data for that manufacturer.

```r
# check for files (show that there are none)
list.files('data/mpg')
character(0)

# create files by taking the mpg df %>% collapsing the data for each manufacturer into a list column %>% walking over the two columns in the df and for each pair (i.e. row of manufacturer and data values) doing: {create path variable to point to the path where the data should be written %>% write the data to disk in .csv format}
mpg %>%
group_nest(manufacturer, keep = T) %>%
pwalk(function(manufacturer, data) {
  path <- file.path('data/mpg', glue::glue('df_{manufacturer}.csv'))
  write_csv(data, path)
})
```
Using `walk()`
Iteratively write data to disk using `purrr::pwalk()`

For each manufacturer in the `mpg` dataset, write a `.csv` file to disk containing only the data for that manufacturer

```r
# check for files (show that there are none)
list.files('data/mpg')
character(0)

# create files by taking the mpg df %>% collapsing the data for each manufacturer into a list column %>% walking over the two columns in the df and for each pair (i.e. row of manufacturer and data values) doing: [create path variable to point to the path where the data should be written %>% write the data to disk in .csv format]
mpg %>%
  group_nest(manufacturer, keep = T) %>%
pwalk(function(manufacturer, data) {
  path <- file.path('data/mpg', glue::glue('df_{manufacturer}.csv'))
  write_csv(data, path)
})

# check for files again (show that there are now files)
list.files('data/mpg') %>% {c(head(.[, 2]), tail(.[, 2]))}
[1] "df_audi.csv"   "df_chevrolet.csv" "df_toyota.csv" "df_volkswagen.csv"
```
Using `iwalk()`

Iteratively read data into `purrr::iwalk()`

Read each of the `.csv` files just written to disk into R's global environment as a data frames. Use each file's name (without the `.csv` extension) as the name for its data frame.
Using `iwalk()`

Iteratively read data into `purrr::iwalk()`

⚠️ Read each of the `.csv` files just written to disk into R's global environment as a data frames. Use each file's name (without the `.csv` extension) as the name for its data frame.

```r
# check for objects (show that there are none)
ls()
character(0)
```
Using `iwalk()`

Iteratively read data into `purrr::iwalk()`

Read each of the `.csv` files just written to disk into R’s global environment as a data frames. Use each file’s name (without the `.csv` extension) as the name for its data frame.

```r
# check for objects (show that there are none)
ls()
character(0)
```

```r
# get a list of all of the .csv files located in data/mpg %>% using set_names, name each element in this list with its filename sans the .csv extension %>% using iwalk to apply the assign function to each element in the list. Specifically, use fread to read the csv file from disk into a data frame and then assign that data frame as a named object to R’s global environment
list.files('data/mpg', pattern = '.csv', full.names = T) %>%
  set_names(str_remove(basename(.), '.csv$')) %>%
iwalk(function(x, i) assign(i, fread(x), .GlobalEnv))
```
Using `iwalk()`

Iteratively read data into `purrr::iwalk()`

Read each of the `.csv` files just written to disk into R's global environment as a data frames. Use each file's name (without the `.csv` extension) as the name for its data frame.

```r
# check for objects (show that there are none)
ls()
character(0)

# get a list of all of the .csv files located in data/mpg %>% using set_names, name each element in this list with its filename sans the .csv extension %>% using iwalk to apply the assign function to each element in the list. Specifically, use fread to read the csv file from disk into a data frame and then assign that data frame as a named object to R's global environment
list.files('data/mpg', pattern = '.csv', full.names = T) %>%
  set_names(str_remove(basename(.), '.csv$')) %>%
iwalk(function(x, i) assign(i, fread(x), .GlobalEnv))

# check for objects again (show that there are now files)
ls() %>% {c(head(.[, 2]), tail(.[, 2]))}
[1] "df_audi"       "df_chevrolet"  "df_toyota"     "df_volkswagen"
```
Using `modify_*()`

Conditionally change contents using `purrr::modify_if()`

For each manufacturer in the `mpg` dataset, express all of the numeric columns as the percentage deviation from the mean.
Using `modify_*()`
Conditionally change contents using `purrr::modify_if()`

For each manufacturer in the `mpg` dataset, express all of the numeric columns as the percentage deviation from the mean.

```r
# define function to express each element in vector as % deviation from mean
myfunc <- function(x) x / mean(x, na.rm = T) - 1
```
Using `modify_*()`

Conditionally change contents using `purrr::modify_if()`

For each manufacturer in the `mpg` dataset, express all of the numeric columns as the percentage deviation from the mean.

```r
# define function to express each element in vector as % deviation from mean
myfunc <- function(x) x / mean(x, na.rm = T) - 1

# data.table approach
# take mpg %>% convert to data.table %>% group by manufacturer, then use modify_if to target all numeric columns and modify each using the deviation function
a <- mpg %>%
  setDT() %>%
  .[by = .(manufacturer),
    j = modify_if(.SD, is.numeric, myfunc)]

# dplyr approach
# take mpg %>% group by manufacturer %>% use mutate_if to mutate all numeric columns using the deviation function %>% ungroup the data
b <- mpg %>%
  group_by(manufacturer) %>%
  mutate_if(is.numeric, myfunc) %>%
  ungroup()
```
Using `modify_*()`
Conditionally change contents using `purrr::modify_if()`

For each manufacturer in the `mpg` dataset, express all of the numeric columns as the percentage deviation from the mean.

```r
# define function to express each element in vector as % deviation from mean
myfunc <- function(x) x / mean(x, na.rm = T) - 1

# data.table approach
# take mpg %>% convert to data.table %>% group by manufacturer, then use modify_if to target all numeric columns and modify each using the deviation function
a <- mpg %>%
    setDT() %>%
    .[by = .(manufacturer),
        j = modify_if(.SD, is.numeric, myfunc)]

# dplyr approach
# take mpg %>% group by manufacturer %>% use mutate_if to mutate all numeric columns using the deviation function %>% ungroup the data
b <- mpg %>%
    group_by(manufacturer) %>%
    mutate_if(is.numeric, myfunc) %>%
    ungroup()

# show that methods produce equivalent outputs
all_equal(a, b)

[1] TRUE
```
Predicate functionals

A predicate function is a function that either returns **TRUE** or **FALSE**. Predicate functionals take vector \( \mathbf{x} \) and predicate function \( f \) and do something useful.
Using `every()` and `some()`

All or nothing some!

For which manufacturers in the `mpg` dataset do the city miles per gallon (`cty`) (a) exceed 15mpg on all models and/or (b) 25 mpg on at least some models?
Using `every()` and `some()`

All or nothing some!

For which manufacturers in the `mpg` dataset do the city miles per gallon (`cty`) (a) exceed 15mpg on all models and/or (b) 25 mpg on at least some models?

```r
# take mpg %>% group by manufacturer %>% use summarise to create 2 summary columns: all_above_15 captures whether every value of cty > 15, while some_above_25 captures whether some values of cty > 25 %>% ungroup the data
mpg %>%
  group_by(manufacturer) %>%
  summarise(
    all_above_15 = every(cty, function(x) x > 15),
    some_above_25 = some(cty, function(x) x > 25))
%>%
  ungroup()
```
Using `every()` and `some()`

All or nothing some!

For which manufacturers in the `mpg` dataset do the city miles per gallon (`cty`) (a) exceed 15mpg on all models and/or (b) 25 mpg on at least some models?

```r
# A tibble: 15 x 3
#  manufacturer  all_above_15  some_above_25
#  <chr>         <lgl>        <lgl>
#1  audi         FALSE        FALSE
#2  chevrolet    FALSE        FALSE
#3  dodge        FALSE        FALSE
#4  ford         FALSE        FALSE
#5  honda        TRUE         TRUE
#6  hyundai      TRUE         FALSE
#7  jeep         FALSE        FALSE
#8  land rover   FALSE        FALSE
#9  lincoln      FALSE        FALSE
#10 mercury      FALSE        FALSE
#11 nissan       FALSE        FALSE
#12 pontiac      TRUE         FALSE
#13 subaru       TRUE         FALSE
#14 toyota       TRUE         FALSE
#15 volkswagen   TRUE         TRUE
```

```r
# take mpg %>% group by manufacturer %>% use summarise to create 2 summary columns: all_above_15 captures whether every value of cty > 15, while some_above_25 captures whether some values of cty > 25 %>% ungroup the data
mpg %>%
group_by(manufacturer) %>%
summarise(
    all_above_15 = every(cty, function(x) x > 15),
    some_above_25 = some(cty, function(x) x > 25))
%>%
ungroup()
```
Using `every()` and `some()`
All or nothing some!

For which manufacturers in the `mpg` dataset do the city miles per gallon (`cty`) (a) exceed 15mpg on all models and/or (b) 25 mpg on at least some models?

```r
# take mpg %>% group by manufacturer %>% use summarise to create 2 summary columns: all_above_15 captures whether every value of cty > 15, while some_above_25 captures whether some values of cty > 25 %>% ungroup the data
mpg %>%
group_by(manufacturer) %>%
summarise(
  all_above_15 = every(cty, function(x) x > 15),
  some_above_25 = some(cty, function(x) x > 25))
%
%>%
ungroup()
```

# A tibble: 15 x 3
#  manufacturer all_above_15 some_above_25
#  <chr>        <lgl>        <lgl>
#1 audi         FALSE        FALSE
#2 chevrolet    FALSE        FALSE
#3 dodge        FALSE        FALSE
#4 ford         FALSE        TRUE
#5 honda        TRUE         TRUE
#6 hyundai      TRUE         FALSE
#7 jeep         FALSE        FALSE
#8 land rover   FALSE        FALSE
#9 lincoln      FALSE        FALSE
#10 mercury     FALSE        FALSE
#11 nissan      FALSE        FALSE
#12 pontiac     TRUE         TRUE
#13 subaru      TRUE         FALSE
#14 toyota      FALSE        TRUE
#15 volkswagen  TRUE         TRUE

Bonus:

```r
# take mpg %>% group by manufacturer %>% filter to keep only data for manufacturers whose models all have cty > 15mpg
mpg %>%
group_by(manufacturer) %>%
filter(every(cty, function(x) x > 15))
```
other vector transformations
reduce() and accumulate()

Collapsing it all or building it up

reduce(.x, .f, ... , .init)
use function .f to combine elements of .x by passing the result of each iteration as an initial value to the next iteration; return single result from final iteration

accumulate(.x, .f, ... , .init)
use function .f to combine elements of .x by passing the result of each iteration as an initial value to the next iteration; return list of results from each iteration
reduce() and accumulate()

Collapsing it all or building it up

reduce(.x, .f, ..., .init)
use function .f to combine elements of .x by passing the result of each iteration as an initial value to the next iteration; return single result from final iteration

```r
# return cumulative sum of 1:5
1:5 %>% reduce(`+`)
```

```r
[1] 15
```

```r
# which numbers appear in the vector 1:5
1:5 %>% reduce(function(x, y) paste(x, 'and', y))
```

```r
[1] "1 and 2 and 3 and 4 and 5"
```

accumulate(.x, .f, ..., .init)
use function .f to combine elements of .x by passing the result of each iteration as an initial value to the next iteration; return list of results from each iteration

```r
# return each step in cumulative sum of 1:5
1:5 %>% accumulate(`+`)
```

```r
[1]  1  3  6 10 15
```

```r
# which numbers appear in each iteration
1:5 %>% accumulate(function(x, y) paste(x, 'and', y))
```

```r
[1] "1"                         "1 and 2"
"1 and 2 and 3"       "1 and 2 and 3 and 4"   "1 and 2 and 3 and 4 and 5"
```
Why `reduce()`? Why `accumulate()`?
what's the point?

- `reduce(.x, .f, ..., .init)`
  use function `.f` to combine elements of `.x` by passing the result of each iteration as an initial value to the next iteration; return single result from final iteration

- `accumulate(.x, .f, ..., .init)`
  use function `.f` to combine elements of `.x` by passing the result of each iteration as an initial value to the next iteration; return list of results from each iteration

**E pluribus unum**
- You want just one thing
- Getting that thing requires repeating (effectively) the same additive operation
  - E.g: `bind_rows()`, `bind_cols()`, `left_join()`, `merge`, etc

**Build something bit by bit**
- Each step that builds up to the final "thing" is of interest
  - Incrementally building a plot
  - Building up a model specification
  - Complex accumulative sequences
Using `accumulate()`

Building up a model using `purrr::accumulate()`

Starting with `cty ~ manufacturer` as a base, (1) build up several linear model specifications for estimating the city miles per gallon (`cty`) in the `mpg` dataset by incrementally adding the `trans`, `drv`, and `class` terms to the model. (2) Estimate each model and report the adjusted R-squared.
Using `accumulate()`

Building up a model using `purrr::accumulate()`

Starting with `cty ~ manufacturer` as a base, (1) build up several linear model specifications for estimating the city miles per gallon (cty) in the `mpg` dataset by incrementally adding the `trans`, `drv`, and `class` terms to the model. (2) Estimate each model and report the adjusted R-squared.

```r
# create a vector of model specifications by taking the relevant column names %>% accumulating each into the base specification using paste and a ' + ' separator %>% number each model sequentially using set_names %>% print the results in a neatly formatted tibble
models <- c('trans', 'drv', 'class') %>%
    accumulate(function(x, y) paste(x, y, sep = ' + '),
               .init = 'cty ~ manufacturer') %>%
    set_names(1:length(.))
enframe(models, name = 'model', value = 'spec')
```

# A tibble: 4 x 2
#  model spec
#1 ctymanufacturer
#2 ctymanufacturer + trans
#3 ctymanufacturer + trans + drv
#4 ctymanufacturer + trans + drv + class
Using `accumulate()`

Building up a model using `purrr::accumulate()`

1. Starting with `cty ~ manufacturer` as a base, build up several linear model specifications for estimating the city miles per gallon (`cty`) in the `mpg` dataset by incrementally adding the `trans`, `drv`, and `class` terms to the model.

   ```r
   models <- c('trans', 'drv', 'class') %>%
             accumulate(function(x, y) paste(x, y, sep = ' + '),
                         .init = 'cty ~ manufacturer') %>%
             set_names(1:length(.))
   enframe(models, name = 'model', value = 'spec')
   # A tibble: 4 x 2
   #  model spec
   # 1 cty ~ manufacturer
   # 2 cty ~ manufacturer + trans
   # 3 cty ~ manufacturer + trans + drv
   # 4 cty ~ manufacturer + trans + drv + class
   ```

2. Estimate each model and report the adjusted R-squared.

   ```r
   models %>%
             map(lm, data = mpg) %>%
             map(summary) %>%
             map_dbl("adj.r.squared") %>%
             enframe(name = 'model', value = 'Adj-R2')
   # A tibble: 4 x 2
   #  model `Adj-R2`
   # 1 cty ~ manufacturer 0.528
   # 2 cty ~ manufacturer + trans 0.551
   # 3 cty ~ manufacturer + trans + drv 0.687
   # 4 cty ~ manufacturer + trans + drv + class 0.713
   ```
Adverbs modify the action of a function; taking a function as input and returning a function with modified action as output.

- Fitter, happier
- More productive
compose() and partial()

why work so hard?

compose(..., .dir = c('backward', 'forward))

- apply functions ... in order in the direction .dir specified

partial(.f, ...)

- modify function .f by pre-filling and fixing some of its arguments
compose() and partial()

why work so hard?

```r
compose(..., .dir = c('backward', 'forward'))
apply functions ... in order in the direction .dir specified

round(mean(log(1:20), na.rm = T), digits = 2)
[1] 2.12

round(mean(log(5:100), na.rm = T), digits = 2)
[1] 3.76

# compose steps into a function
mycomp <- compose(log,
                   ~ mean(.x, na.rm = T),
                   ~ round(.x, digits = 2),
                   .dir = 'forward')
mycomp(1:20)
[1] 2.12

mycomp(5:100)
[1] 3.76

partial(.f, ...)
modify function .f by pre-filling and fixing some of its arguments

round(0.532131245, digits = 2)
[1] 0.53

round(12394.13498134, digits = 2)
[1] 12394.13

# prefill and fix parameter (WARNING!)
myround <- partial(round, digits = 2)
myround(0.532131245)
[1] 0.53

myround(12394.13498134)
[1] 12394.13

myround(1/3, digits = 2)
Error in (function (x, digits = 0) : formal argument "digits" matched by multiple actual arguments
```
Using `compose()`

There's more than one way to skin a cat

Compose a function that can be used to estimate each model in the previously defined `models` vector and report the adjusted R-squared.
Using `compose()`
There's more than one way to skin a cat

Compose a function that can be used to estimate each model in the previously defined `models` vector and report the adjusted R-squared.

```r
# previously
def estimate_models(models) {
  summaries <- models %>%
    map(lm, data = mpg) %>%
    map(summary) %>%
    map_dbl("adj.r.squared") %>%
    enframe(name = 'model', value = 'Adj-R2')
  return(summaries)
}
```

Using `compose()`
There's more than one way to skin a cat

Compose a function that can be used to estimate each model in the previously defined `models` vector and report the adjusted R-squared.

```r
# A tibble: 4 x 2
model `Adj-R2`
<chr>    <dbl>
1 1        0.528
2 2        0.551
3 3        0.687
4 4        0.713
```
Using `compose()`

There’s more than one way to skin a cat

Compose a function that can be used to estimate each model in the previously defined `models` vector and report the adjusted R-squared.

### previously

```r
# take models %>% estimate each using map to apply the lm function %>% get the summary for each set of results using map to apply the summary function %>% extract the adjusted r-squared for each set of summary results using map_dbl to extract it by name %>% print the results in a neatly formatted tibble
models %>%
  map(lm, data = mpg) %>%
  map(summary) %>%
  map_dbl("adj.r.squared") %>%
  enframe(name = 'model', value = 'Adj-R2')

# A tibble: 4 x 2
#  model `Adj-R2`
#   <chr>  <dbl>
#    1 1     0.528
#    2 2     0.551
#    3 3     0.687
#    4 4     0.713
```

### alternative

```r
# compose a function that sends arguments to lm, then passes the results to summary, then plucks the r-squared from those results, then enframes
mycomp <- compose(lm, summary, ~pluck(.x, 'adj.r.squared'), ~enframe(.x, name = 'model', value = 'adj.r.squared'), .dir = 'forward')

# take models %>% estimate each by using map_dfr to apply the mycomp function and row bind
models %>%
  map_dfr(~mycomp(.x, data = mpg))

# A tibble: 4 x 2
#  model adj.r.squared
#   <int>         <dbl>
#    1     1         0.528
#    2     2         0.551
#    3     3         0.687
#    4     4         0.713
```

Using `compose()`

There’s more than one way to skin a cat
safely(), possibly(), and insistently()

Failure is not an option!

- `safely(.f, otherwise = NULL, quiet = TRUE)` modifies function `.f` to return a list with components `result` (result if not error, NA otherwise) and `error` (error message if error, NULL otherwise).

- `possibly(.f, otherwise, quiet = TRUE)` modifies function `.f` to return `otherwise` if error occurs.

- `insistently(f, rate = rate_backoff())` modifies function `.f` to retry specified times on error.
safely(), possibly(), and insistently()
Failure is not an option!

safely(.f, otherwise = NULL, quiet = TRUE)
modifies function .f to return a list with components result (result if not error, NA otherwise) and error (error message if error, NULL otherwise).

possibly(.f, otherwise, quiet = TRUE)
modifies function .f to return otherwise if error occurs.

insistently(f, rate = rate_backoff())
modifies function .f to retry specified times on error.

```r
# define bad function that only works on odd numbers
badfunc <- function(x) if (x %% 2 == 0) stop('Only odd numbers allowed') else (x)

# define safe version of badfunc, possible, and insistent versions of badfunc
safely_badfunc <- safely(badfunc)

possibly_badfunc <- possibly(badfunc, otherwise = NA_real_)

insistently_badfunc <- insistently(badfunc, rate = rate_backoff(pause_cap = 1, max_times = 4))
```
safely(), possibly(), and insistently()

Failure is not an option!

- **safely(.f, otherwise = NULL, quiet = TRUE)** modifies function .f to return a list with components `result` (result if not error, NA otherwise) and `error` (error message if error, NULL otherwise).

- **possibly(.f, otherwise, quiet = TRUE)** modifies function .f to return `otherwise` if error occurs.

- **insistently(f, rate = rate_backoff())** modifies function .f to retry specified times on error.

---

### "Good" value

```r
# test functions of "good" value
badfunc(1)
[1] 1

safely_badfunc(1)
$message
NULL

$error
NULL

possibly_badfunc(1)
[1] 1

insistently_badfunc(1)
[1] 1
```

### "Bad" value

```r
# test functions of "bad" value
badfunc(2)
Error in badfunc(2): Only odd numbers allowed

safely_badfunc(2)
$message
NULL

$error
<simpleError in .f(...): Only odd numbers allowed>

possibly_badfunc(2)
[1] NA

insistently_badfunc(2)
Error: Request failed after 4 attempts
```
Why `safely()`? Why `possibly()`? Why `insistently()`?

What's the point?

`safely(.f, otherwise = NULL, quiet = TRUE)` modifies function `.f` to return a list with components `result` (result if not error, NA otherwise) and `error` (error message if error, NULL otherwise).

`possibly(.f, otherwise, quiet = TRUE)` modifies function `.f` to return `otherwise` if error occurs.

`insistently(f, rate = rate_backoff())` modifies function `.f` to retry specified times on error.

---

**Give me the info and let me decide what to do**
- Ever called an API?
- Allows for robust, flexible error handling

**Let's pretend that didn't happen, okay?**
- Don't get bogged down with failures
- You care (a bit) about the fact that there was an error, but not enough to want to stop.
- You don't care at all about the reason for the error or you're fairly confident about why there is an error

**If at first you don't succeed**
- Get back on that horse!
- Really only useful if you expect the chance of success to change with repeated attempts
  - i.e. the input to the function could change over successive calls
More `purrr` fun(ctions)

But wait, there's more...

**Generalisations**
- `keep()` and `discard` as generalisations of `dplyr::select_if()`
- `pluck()` as generalisation of `[[` and `dplyr::pull()`
- etc.

**Companions**
- `prepend()` as companion to `append()`
- `negate()` as companion to any predicate function

etc
- more predicate functionals
- more vector transformations
- etc.
I’m intrigued...
where can I learn more?

Reference (R/Rstudio)
- `help(package = purrr)`
- F1 to show function help
- F2 to inspect function

Reference (online)
- purrr cheatsheet
- purrr reference

Learning and understanding
- Hadley Wickham’s Advanced R Chapter 9: Functionals
- Jenny Bryan’s purrr tutorial
- Emil Hvitfeldt’s Purrr - tips and tricks
- Emily Robinson’s Going Off the Map: Exploring purrr’s Other Functions
- Colin Fay’s 6-part A Crazy Little Thing Called {purrr}