# INTRODUCTION TO PURRR

## WELCOME R-LADIES!

WANT TO PLAY ALONG?

CODE AVAILABLE AT GITHUB.COM/JENNIFERTHOMPSON/RLADIESINTROTOPURRR

install.packages("tidyverse")
install.packages("viridis")

WILL GET YOU SET UP

**OPTIONAL**: TO RUN **ALL** THE CODE, YOU'LL ALSO NEED A DATA.WORLD ACCOUNT + API TOKEN, AND

install.packages("data.world")

JENNIFER THOMPSON, MPH @JENT103

R-LADIES LOUISVILLE **INAUGURAL** MEETUP 🎉

APRIL 2018

“

”

“

ITERATION:

”

“

# ITERATION:

”

“

ITERATION:

DOING THE SAME* THING TO A BUNCH OF THINGS

”

“

ITERATION:

DOING THE SAME* THING TO A BUNCH OF THINGS

”

“

ITERATION:

DOING THE SAME* THING TO A BUNCH OF THINGS

**ISH*

”

“

ITERATION:

DOING THE SAME* THING TO A BUNCH OF THINGS

*ISH

— *Jennifer Thompson*

”

“

# ITERATION:

# DOING THE SAME* THING TO A BUNCH OF THINGS

*ISH

— *Jennifer Thompson*

”

AUDIENCE PARTICIPATION!

WHAT ARE SOME EXAMPLES?

# WAYS TO ITERATE IN R

doctordisruption.com

# WAYS TO ITERATE IN R

AUDIENCE PARTICIPATION!

HOW DO **YOU** ITERATE?

doctordisruption.com

# WAYS TO ITERATE IN R

- **Copy & paste**
  - PROS: easy (in the short run)
  - CONS: hard to maintain, edit

AUDIENCE PARTICIPATION!

HOW DO **YOU** ITERATE?

doctordisruption.com

# WAYS TO ITERATE IN R

- Copy & paste
  - PROS: easy (in the short run)
  - CONS: hard to maintain, edit
- for loops
  - PROS: easy to conceptualize, write
  - CONS: inefficient

AUDIENCE PARTICIPATION!

HOW DO **YOU** ITERATE?

doctordisruption.com

# WAYS TO ITERATE IN R

AUDIENCE PARTICIPATION!

HOW DO **YOU** ITERATE?

- **Copy & paste**
  - PROS: easy (in the short run)
  - CONS: hard to maintain, edit
- **for loops**
  - PROS: easy to conceptualize, write
  - CONS: inefficient
- **lapply**
  - PROS: faster than for loop; base R -> more stable, fewer dependencies
  - CONS: often need to do something else after you iterate (eg, `do.call(rbind, lapply(…)))`

doctordisruption.com

# WAYS TO ITERATE IN R

AUDIENCE PARTICIPATION!

HOW DO **YOU** ITERATE?

- **Copy & paste**
  - PROS: easy (in the short run)
  - CONS: hard to maintain, edit
- **for loops**
  - PROS: easy to conceptualize, write
  - CONS: inefficient
- **lapply**
  - PROS: faster than for loop; base R -> more stable, fewer dependencies
  - CONS: often need to do something else after you iterate (eg, `do.call(rbind, lapply(…)))`
- **apply**
  - PROS: working with rows/cols in a data.frame
  - CONS: syntax & defining functions can be tricky

doctordisruption.com

# WAYS TO ITERATE IN R

AUDIENCE PARTICIPATION!

HOW DO **YOU** ITERATE?

- **Copy & paste**
  - PROS: easy (in the short run)
  - CONS: hard to maintain, edit
- **for loops**
  - PROS: easy to conceptualize, write
  - CONS: inefficient
- **lapply**
  - PROS: faster than for loop; base R -> more stable, fewer dependencies
  - CONS: often need to do something else after you iterate (eg, `do.call(rbind, lapply(…)))`
- **apply**
  - PROS: working with rows/cols in a data.frame
  - CONS: syntax & defining functions can be tricky
- **mapply, sapply, tapply, vapply**
  - PROS: efficient at their specific purposes
  - CONS: mystifying, inconsistent syntax

doctordisruption.com

# REASONS TO USE PURRR VS BASE R

- Consistent, readable syntax (compare to `apply` vs `lapply` vs `mapply` vs…)
- More efficient than for loops
- Plays nicely with pipes `%>%`
- Returns the output you expect (type-stable)
- Ease of making changes
- Flexibility
- Particularly excellent if you work with list-columns, JSON, other non-strictly-rectangular data

**PREAMBLE:**

STOP WORRYING AND

# LOVE LISTS

**PREAMBLE:**

STOP WORRYING AND

# LOVE LISTS

- Lists in R are collections of elements - that’s it

**PREAMBLE:**

STOP WORRYING AND

# LOVE LISTS

- Lists in R are collections of elements - that's it
- Each element can be any length and any type… even another list (it's lists all the way down…)

**PREAMBLE:**
STOP WORRYING AND

# LOVE LISTS

- Lists in R are collections of elements - that's it
- Each element can be any length and any type… even another list (it's lists all the way down…)
- Totally valid example:

```
list("a" = 1:10,          ## numeric vector of length 10
```

**PREAMBLE:**
STOP WORRYING AND

# LOVE LISTS

- Lists in R are collections of elements - that's it
- Each element can be any length and any type… even another list (it's lists all the way down…)
- Totally valid example:

```
list("a" = 1:10,           ## numeric vector of length 10
     "b" = list(1:10),     ## list of length 1; element 1 = vector of length 10
```

## **PREAMBLE:**
## STOP WORRYING AND

# LOVE LISTS

- Lists in R are collections of elements - that's it
- Each element can be any length and any type… even another list (it's lists all the way down…)
- Totally valid example:

```
list("a" = 1:10,          ## numeric vector of length 10
     "b" = list(1:10),    ## list of length 1; element 1 = vector of length 10
     "c" = LETTERS[1:10]) ## character vector of length 10
```

**PREAMBLE:**

STOP WORRYING AND

# LOVE LISTS

- Lists in R are collections of elements - that's it
- Each element can be any length and any type… even another list (it's lists all the way down…)
- Totally valid example:

```
list("a" = 1:10,          ## numeric vector of length 10
     "b" = list(1:10),    ## list of length 1; element 1 = vector of length 10
     "c" = LETTERS[1:10]) ## character vector of length 10
```

**PREAMBLE:**

STOP WORRYING AND

# LOVE LISTS

- Lists in R are collections of elements - that's it
- Each element can be any length and any type… even another list (it's lists all the way down…)
- Totally valid example:

```
list("a" = 1:10,           ## numeric vector of length 10
     "b" = list(1:10),     ## list of length 1; element 1 = vector of length 10
     "c" = LETTERS[1:10]) ## character vector of length 10
```

- With such flexibility comes both great power & great complexity

**PREAMBLE:**

STOP WORRYING AND

# LOVE LISTS

- Lists in R are collections of elements - that's it
- Each element can be any length and any type… even another list (it's lists all the way down…)
- Totally valid example:

```
list("a" = 1:10,           ## numeric vector of length 10
     "b" = list(1:10),     ## list of length 1; element 1 = vector of length 10
     "c" = LETTERS[1:10]) ## character vector of length 10
```

- With such flexibility comes both great power & great complexity
- `purrr` works really well with lists by providing ways to:
  - iterate quickly over lists comprising elements of the same type
  - quickly extract elements of complicated lists

# MAP(): WHERE IT'S AT

## (ALONG WITH ITS VARIANTS)

# MAP(): WHERE IT'S AT
## (ALONG WITH ITS VARIANTS)

- Let us do the same (or similar) things to a list of things, and know what kind of output to expect

# MAP(): WHERE IT'S AT
## (ALONG WITH ITS VARIANTS)

- Let us do the same (or similar) things to a list of things, and know what kind of output to expect
- Several variants, depending on how many combinations you're iterating over and what type of output you want

# MAP(): WHERE IT'S AT

## (ALONG WITH ITS VARIANTS)

- Let us do the same (or similar) things to a list of things, and know what kind of output to expect
- Several variants, depending on how many combinations you're iterating over and what type of output you want
- Different combinations: `map()` (one thing), `map2()` (two things), `pmap()` (infinite number of things)

# MAP(): WHERE IT'S AT

## (ALONG WITH ITS VARIANTS)

- Let us do the same (or similar) things to a list of things, and know what kind of output to expect
- Several variants, depending on how many combinations you're iterating over and what type of output you want
- Different combinations: `map()` (one thing), `map2()` (two things), `pmap()` (infinite number of things)
- Different outputs: `map()` (list), `map_chr()`, `map_dbl()`, `map_int()`, `map_lgl()`, `map_df()`

# MAP(): WHERE IT'S AT

## (ALONG WITH ITS VARIANTS)

- Let us do the same (or similar) things to a list of things, and know what kind of output to expect
- Several variants, depending on how many combinations you're iterating over and what type of output you want
- Different combinations: `map()` (one thing), `map2()` (two things), `pmap()` (infinite number of things)
- Different outputs: `map()` (list), `map_chr()`, `map_dbl()`, `map_int()`, `map_lgl()`, `map_df()`

THEY ALL WORK THIS WAY:

# MAP(): WHERE IT'S AT

## (ALONG WITH ITS VARIANTS)

- Let us do the same (or similar) things to a list of things, and know what kind of output to expect
- Several variants, depending on how many combinations you're iterating over and what type of output you want
- Different combinations: `map()` (one thing), `map2()` (two things), `pmap()` (infinite number of things)
- Different outputs: `map()` (list), `map_chr()`, `map_dbl()`, `map_int()`, `map_lgl()`, `map_df()`

## THEY ALL WORK THIS WAY:

Two sets of arguments:

1. **What we're iterating over**
   *Specified differently depending on which map() we're using*
2. **What we're doing each time**
   *Always specified as .f*
   *Can be built-in, user-defined, or anonymous (defined within the map() call itself)*

# MAP(): WHERE IT'S AT
## (ALONG WITH ITS VARIANTS)

- Let us do the same (or similar) things to a list of things, and know what kind of output to expect
- Several variants, depending on how many combinations you're iterating over and what type of output you want
- Different combinations: `map()` (one thing), `map2()` (two things), `pmap()` (infinite number of things)
- Different outputs: `map()` (list), `map_chr()`, `map_dbl()`, `map_int()`, `map_lgl()`, `map_df()`

## THEY ALL WORK THIS WAY:

Two sets of arguments:

1. **What we're iterating over**
   *Specified differently depending on which map() we're using*
2. **What we're doing each time**
   *Always specified as .f*
   *Can be built-in, user-defined, or anonymous (defined within the map() call itself)*

```
map(.x = …, .f = …)

map2(.x = …, .y = …, .f = …)

pmap(.l = list(a1 = …, a2 = …, …),
     .f = …)
```

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10
```

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10

map(v1, ~ . * 3)
```

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10
```

```
map(v1, ~ . * 3)
```

"this is an anonymous function"

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10
```

“this is what I want to do to each element”

```
map(v1, ~ . * 3)
```

“this is an anonymous function”

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10
```

```
map(v1, ~ . * 3)
```

“this is what I want to do to each element”

“this is an anonymous function”

“this is where the thing I’m iterating over goes”

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10

map(v1, ~ . * 3)

[[1]]
[1] 3

[[2]]
[1] 6
…

[[10]]
[1] 30
```

"this is what I want to do to each element"

"this is an anonymous function"

"this is where the thing I'm iterating over goes"

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10

map(v1, ~ . * 3)
```

"this is what I want to do to each element"

"this is an anonymous function"

"this is where the thing I'm iterating over goes"

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

[[2]]
[1] 6
…

[[10]]
[1] 30
```

Great if we want to continue iterating!

If we *really* want a new vector:

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10
```

"this is what I want to do to each element"

```
map(v1, ~ . * 3)
```

"this is an anonymous function"

"this is where the thing I'm iterating over goes"

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

[[2]]
[1] 6
…

[[10]]
[1] 30
```

Great if we want to continue iterating!

If we *really* want a new vector:

```
map_dbl(v1, ~ . * 3)

[1]  3  6  9 12 15 18 21 24 27 30
```

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10
```

“this is what I want to do to each element”

```
map(v1, ~ . * 3)
```

“this is an anonymous function”

“this is where the thing I'm iterating over goes”

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

[[2]]
[1] 6
...

[[10]]
[1] 30
```

Great if we want to continue iterating!

If we *really* want a new vector:

```
map_dbl(v1, ~ . * 3)
[1]  3  6  9 12 15 18 21 24 27 30
```

Examples of other `map` variants:

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10

map(v1, ~ . * 3)
```

"this is what I want to do to each element"

"this is an anonymous function"

"this is where the thing I'm iterating over goes"

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

[[2]]
[1] 6
…

[[10]]
[1] 30
```

Great if we want to continue iterating!
If we *really* want a new vector:

```
map_dbl(v1, ~ . * 3)
[1]  3  6  9 12 15 18 21 24 27 30
```

Examples of other `map` variants:

```
map_chr(v1, ~ LETTERS[.])
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10
```

```
map(v1, ~ . * 3)
```

"this is what I want to do to each element"

"this is an anonymous function"

"this is where the thing I'm iterating over goes"

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

[[2]]
[1] 6
…

[[10]]
[1] 30
```

Great if we want to continue iterating!

If we *really* want a new vector:

```
map_dbl(v1, ~ . * 3)
[1]  3  6  9 12 15 18 21 24 27 30
```

Examples of other map variants:

```
map_chr(v1, ~ LETTERS[.])
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

```
map2_dbl(v1, v2, sum)
[1]  2  4  6  8 10 12 14 16 18 20
```

# MAP EXAMPLES

## + USING ANONYMOUS FUNCTIONS IN PURRR

```
v1 <- v2 <- 1:10
```

"this is what I want to do to each element"

```
map(v1, ~ . * 3)
```

"this is an anonymous function"

"this is where the thing I'm iterating over goes"

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

[[2]]
[1] 6
…

[[10]]
[1] 30
```

Great if we want to continue iterating!
If we *really* want a new vector:

```
map_dbl(v1, ~ . * 3)
[1]  3  6  9 12 15 18 21 24 27 30
```

Examples of other map variants:

```
map_chr(v1, ~ LETTERS[.])
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"

map2_dbl(v1, v2, sum)
[1]  2  4  6  8 10 12 14 16 18 20

map_lgl(v1, is.numeric)
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE…
```

# EXAMPLE TIME!

## PLAY ALONG IF YOU LIKE

GITHUB.COM/JENNIFERTHOMPSON/RLADIESINTROTOPURRR

# EXAMPLE TIME!

## PLAY ALONG IF YOU LIKE
## GITHUB.COM/JENNIFERTHOMPSON/RLADIESINTROTOPURRR

- **Goal**: Show lots of `purrr`'s capabilities, give you ideas about how you can use it

# EXAMPLE TIME!

## PLAY ALONG IF YOU LIKE

## GITHUB.COM/JENNIFERTHOMPSON/RLADIESINTROTOPURRR

- **Goal**: Show lots of `purrr`'s capabilities, give you ideas about how you can use it
- **Not goals:**
  - Write most efficient code possible
  - Complex statistical analysis

# EXAMPLE TIME!

## PLAY ALONG IF YOU LIKE

## GITHUB.COM/JENNIFERTHOMPSON/RLADIESINTROTOPURRR

- **Goal**: Show lots of `purrr`'s capabilities, give you ideas about how you can use it
- **Not goals:**
  - Write most efficient code possible
  - Complex statistical analysis
- **What we'll do**:

1. Extract data stored in multiple files and combine it into 3 datasets
2. Fit the same model to three different outcomes
3. Check assumptions for those models
4. If needed, update the model
5. Visualize our model results

# MOTIVATING DATA

## IT'S ALMOST NATIONAL PARK WEEK!

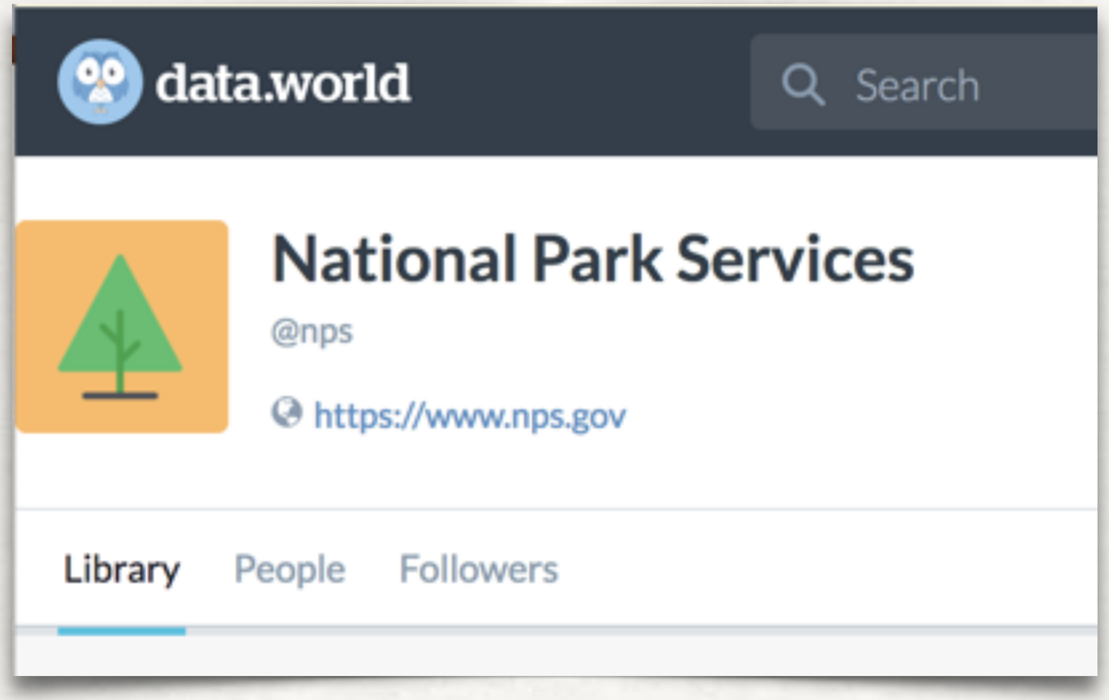
**AUDIENCE PARTICIPATION!**

TO CODE ALONG WITH THIS SECTION, YOU'LL NEED A DATA.WORLD ACCOUNT AND AN API TOKEN

ALTERNATELY: PURRR_DATA.RDATA

# MOTIVATING DATA

## IT'S ALMOST NATIONAL PARK WEEK!




# data.world/nps

*We'll look at annual total recreation, tent camping, and backcountry camping visits from the years 2007-2016.*

*We'll also use the Glossary to restrict our data to only national parks.*

**AUDIENCE PARTICIPATION!**

TO CODE ALONG WITH THIS SECTION, YOU'LL NEED A DATA.WORLD ACCOUNT AND AN API TOKEN

ALTERNATELY: PURRR_DATA.RDATA

# TOTAL RECREATIONAL VISITS

## PRETTY SURE THIS MEANS EVERYBODY

# TOTAL RECREATIONAL VISITS

## PRETTY SURE THIS MEANS EVERYBODY

10 files

**Recreation_Visits_2007.csv**
Request more info

| | parkname | # rank | # value | # percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 17,352,286 | 6.3% |
| 2 | Golden Gate NRA | 2 | 14,397,313 | 5.22% |
| 3 | Great Smoky Mountains NP | 3 | 9,372,253 | 3.4% |
| 4 | Gateway NRA | 4 | 8,813,204 | 3.2% |
| 5 | Lake Mead NRA | 5 | 7,622,139 | 2.77% |

Showing 1-5 of 359 rows, 4 columns See all

**Recreation_Visits_2008.csv**
Request more info

| | parkname | # rank | # value | # percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 16,309,307 | 5.93% |
| 2 | Golden Gate NRA | 2 | 14,554,750 | 5.3% |
| 3 | Gateway NRA | 3 | 9,431,021 | 3.43% |
| 4 | Great Smoky Mountains NP | 4 | 9,044,010 | 3.29% |
| 5 | Lake Mead NRA | 5 | 7,601,863 | 2.77% |

Showing 1-5 of 360 rows, 4 columns See all

**Recreation_Visits_2009.csv**
Request more info

| | parkname | # rank | # value | # percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 15,936,316 | 5.58% |
| 2 | Golden Gate NRA | 2 | 15,036,372 | 5.27% |

# TOTAL RECREATIONAL VISITS

## PRETTY SURE THIS MEANS EVERYBODY

10 files

**Recreation_Visits_2007.csv**
Request more info

| | parkname | # rank | # value | # percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 17,352,286 | 6.3% |
| 2 | Golden Gate NRA | 2 | 14,397,313 | 5.22% |
| 3 | Great Smoky Mountains NP | 3 | 9,372,253 | 3.4% |
| 4 | Gateway NRA | 4 | 8,813,204 | 3.2% |
| 5 | Lake Mead NRA | 5 | 7,622,139 | 2.77% |

Showing 1-5 of 359 rows, 4 columns See all

**Recreation_Visits_2008.csv**
Request more info

| | parkname | # rank | # value | # percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 16,309,387 | 5.93% |
| 2 | Golden Gate NRA | 2 | 14,554,750 | 5.3% |
| 3 | Gateway NRA | 3 | 9,431,021 | 3.43% |
| 4 | Great Smoky Mountains NP | 4 | 9,044,010 | 3.29% |
| 5 | Lake Mead NRA | 5 | 7,601,863 | 2.77% |

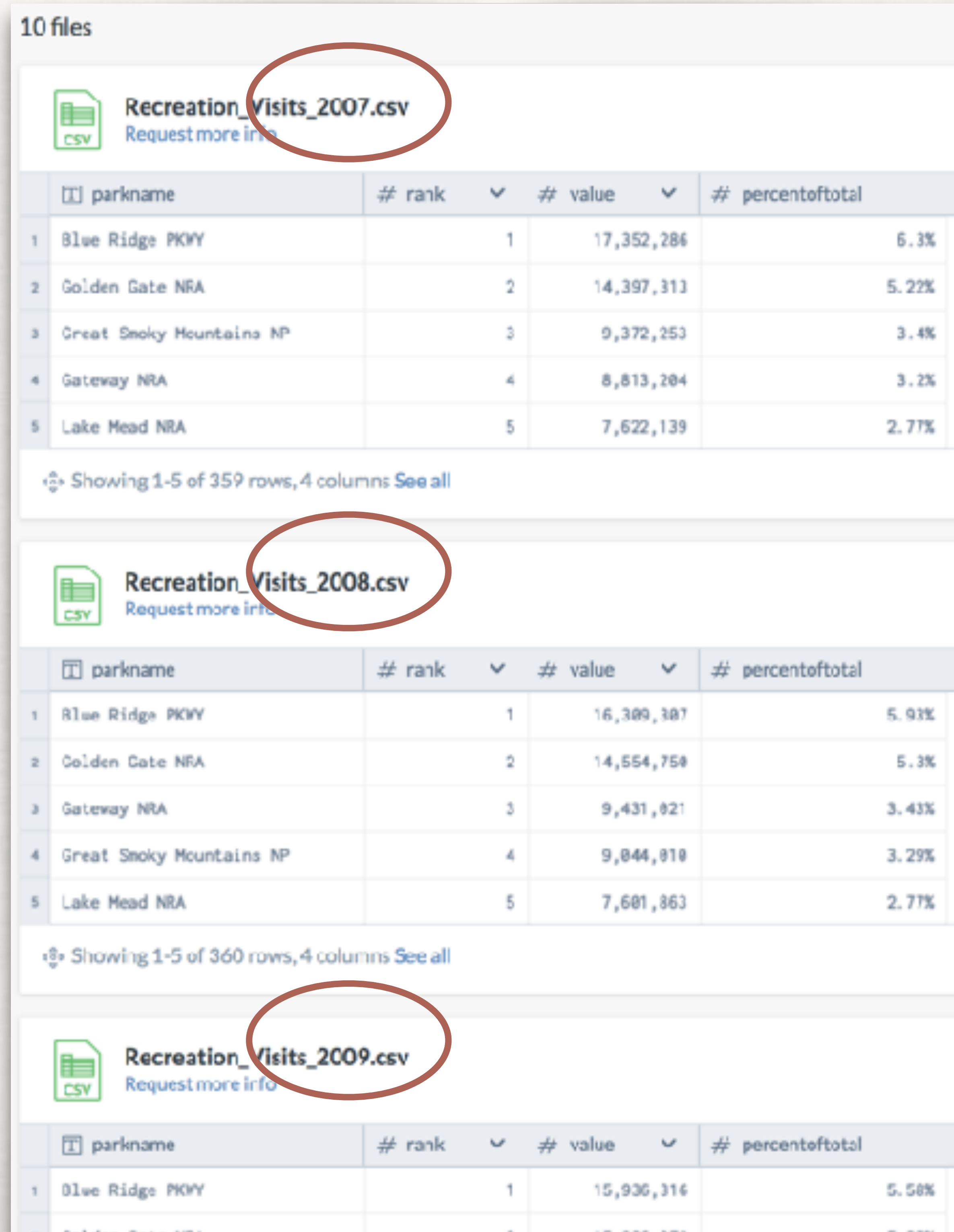
Showing 1-5 of 360 rows, 4 columns See all

**Recreation_Visits_2009.csv**
Request more info

| | parkname | # rank | # value | # percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 15,936,316 | 5.58% |
| 2 | Golden Gate NRA | 2 | 15,036,372 | 5.27% |

# TOTAL RECREATIONAL VISITS

## PRETTY SURE THIS MEANS EVERYBODY


10 files

Recreation_Visits_2007.csv
Request more info

| | parkname | rank | value | percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 17,352,286 | 6.3% |
| 2 | Golden Gate NRA | 2 | 14,397,313 | 5.22% |
| 3 | Great Smoky Mountains NP | 3 | 9,372,253 | 3.4% |
| 4 | Gateway NRA | 4 | 8,813,204 | 3.2% |
| 5 | Lake Mead NRA | 5 | 7,622,139 | 2.77% |

Showing 1-5 of 359 rows, 4 columns See all

Recreation_Visits_2008.csv
Request more info

| | parkname | rank | value | percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 16,309,307 | 5.93% |
| 2 | Golden Gate NRA | 2 | 14,554,750 | 5.3% |
| 3 | Gateway NRA | 3 | 9,431,021 | 3.43% |
| 4 | Great Smoky Mountains NP | 4 | 9,044,010 | 3.29% |
| 5 | Lake Mead NRA | 5 | 7,601,863 | 2.77% |

Showing 1-5 of 360 rows, 4 columns See all

Recreation_Visits_2009.csv
Request more info

| | parkname | rank | value | percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 15,936,316 | 5.58% |
| 2 | Golden Gate NRA | 2 | 15,036,372 | 5.27% |

- Each year's data is stored as a separate CSV file
- Each file has the same columns, and same name, except for the year
- Datasets for tent campers, backcountry campers are formatted the same way
- This seems like a prime target for…

# TOTAL RECREATIONAL VISITS

## PRETTY SURE THIS MEANS EVERYBODY

10 files

**Recreation_Visits_2007.csv**
Request more info

| | parkname | rank | value | percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 17,352,286 | 6.3% |
| 2 | Golden Gate NRA | 2 | 14,397,313 | 5.22% |
| 3 | Great Smoky Mountains NP | 3 | 9,372,253 | 3.4% |
| 4 | Gateway NRA | 4 | 8,813,204 | 3.2% |
| 5 | Lake Mead NRA | 5 | 7,622,139 | 2.77% |

Showing 1-5 of 359 rows, 4 columns See all

**Recreation_Visits_2008.csv**
Request more info

| | parkname | rank | value | percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 16,309,307 | 5.93% |
| 2 | Golden Gate NRA | 2 | 14,554,750 | 5.3% |
| 3 | Gateway NRA | 3 | 9,431,021 | 3.43% |
| 4 | Great Smoky Mountains NP | 4 | 9,044,010 | 3.29% |
| 5 | Lake Mead NRA | 5 | 7,601,863 | 2.77% |

Showing 1-5 of 360 rows, 4 columns See all

**Recreation_Visits_2009.csv**
Request more info

| | parkname | rank | value | percentoftotal |
|---|---|---|---|---|
| 1 | Blue Ridge PKWY | 1 | 15,936,316 | 5.58% |
| 2 | Golden Gate NRA | 2 | 15,036,372 | 5.27% |

- Each year's data is stored as a separate CSV file
- Each file has the same columns, and same name, except for the year
- Datasets for tent campers, backcountry campers are formatted the same way
- This seems like a prime target for…

**AUDIENCE PARTICIPATION!**

1. FILL IN THE BLANK

2. WHAT ARE WAYS YOU MIGHT HAVE DONE THIS WITHOUT PURRR?

# USING MAP + USER-DEFINED FUNCTIONS

# USING MAP + USER-DEFINED FUNCTIONS

## STEP 1: WRITE FUNCTION

```
download_nps_year <- function(
  year = 2007:2016,
  table_prefix,
  url
){
  data.world::query(
    data.world::qry_sql(
      sprintf(
        "SELECT * FROM %s_%s”,
        table_prefix, year
      )
    ),
    dataset = url
  ) %>%
    ## As long as we're customizing…
    select(parkname, value) %>%
    mutate(year = year)
}
```

# USING MAP + USER-DEFINED FUNCTIONS

## STEP 1: WRITE FUNCTION

```
download_nps_year <- function(
  year = 2007:2016,
  table_prefix,
  url
){
  data.world::query(
    data.world::qry_sql(
      sprintf(
        "SELECT * FROM %s_%s”,
        table_prefix, year
      )
    ),
    dataset = url
  ) %>%
    ## As long as we're customizing…
    select(parkname, value) %>%
    mutate(year = year)
}
```

allows us to do the same for tent, backcountry data

# USING MAP + USER-DEFINED FUNCTIONS

## STEP 1: WRITE FUNCTION

```
download_nps_year <- function(
  year = 2007:2016,
  table_prefix,
  url
){
  data.world::query(
    data.world::qry_sql(
      sprintf(
        "SELECT * FROM %s_%s”,
        table_prefix, year
      )
    ),
    dataset = url
  ) %>%
    ## As long as we're customizing…
    select(parkname, value) %>%
    mutate(year = year)
}
```

allows us to do the same for tent, backcountry data

## STEP 2: EXTRACT EACH DATASET

# USING MAP + USER-DEFINED FUNCTIONS

## STEP 1: WRITE FUNCTION

```
download_nps_year <- function(
  year = 2007:2016,
  table_prefix,
  url
){
  data.world::query(
    data.world::qry_sql(
      sprintf(
        "SELECT * FROM %s_%s”,
        table_prefix, year
      )
    ),
    dataset = url
  ) %>%
    ## As long as we're customizing…
    select(parkname, value) %>%
    mutate(year = year)
}
```

allows us to do the same for tent, backcountry data

## STEP 2: EXTRACT EACH DATASET

```
## Download recreation data
url_recvisits <-
  "https://data.world/nps/annual-park-ranking-recreation-visits"

rec_visits <- map_df(
  .x = 2007:2016,
  .f = download_nps_year,
  table_prefix = "recreation_visits",
  url = url_recvisits
)
```

# USING MAP + USER-DEFINED FUNCTIONS

## STEP 1: WRITE FUNCTION

```
download_nps_year <- function(
  year = 2007:2016,
  table_prefix,
  url
){
  data.world::query(
    data.world::qry_sql(
      sprintf(
        "SELECT * FROM %s_%s”,
        table_prefix, year
      )
    ),
    dataset = url
  ) %>%
    ## As long as we're customizing…
    select(parkname, value) %>%
    mutate(year = year)
}
```

allows us to do the same for tent, backcountry data

## STEP 2: EXTRACT EACH DATASET

```
## Download recreation data
url_recvisits <-
  "https://data.world/nps/annual-park-ranking-recreation-visits"

rec_visits <- map_df(
  .x = 2007:2016,
  .f = download_nps_year,
  table_prefix = "recreation_visits",
  url = url_recvisits
)
```

what we’re iterating over

# USING MAP + USER-DEFINED FUNCTIONS

## STEP 1: WRITE FUNCTION

```
download_nps_year <- function(
  year = 2007:2016,
  table_prefix,
  url
){
  data.world::query(
    data.world::qry_sql(
      sprintf(
        "SELECT * FROM %s_%s”,
        table_prefix, year
      )
    ),
    dataset = url
  ) %>%
    ## As long as we're customizing…
    select(parkname, value) %>%
    mutate(year = year)
}
```

allows us to do the same for tent, backcountry data

## STEP 2: EXTRACT EACH DATASET

```
## Download recreation data
url_recvisits <-
  "https://data.world/nps/annual-park-ranking-recreation-visits"

rec_visits <- map_df(
  .x = 2007:2016,
  .f = download_nps_year,
  table_prefix = "recreation_visits",
  url = url_recvisits
)
```

what we’re iterating over

what we’re doing

# USING MAP + USER-DEFINED FUNCTIONS

## STEP 1: WRITE FUNCTION

```
download_nps_year <- function(
  year = 2007:2016,
  table_prefix,
  url
){
  data.world::query(
    data.world::qry_sql(
      sprintf(
        "SELECT * FROM %s_%s”,
        table_prefix, year
      )
    ),
    dataset = url
  ) %>%
    ## As long as we're customizing…
    select(parkname, value) %>%
    mutate(year = year)
}
```

allows us to do the same for tent, backcountry data

## STEP 2: EXTRACT EACH DATASET

```
## Download recreation data
url_recvisits <-
  "https://data.world/nps/annual-park-ranking-recreation-visits"

rec_visits <- map_df(
  .x = 2007:2016,
  .f = download_nps_year,
  table_prefix = "recreation_visits",
  url = url_recvisits
)
```

what we’re iterating over

what we’re doing

what stays the same each time

# USING MAP + USER-DEFINED FUNCTIONS

## STEP 1: WRITE FUNCTION

```
download_nps_year <- function(
  year = 2007:2016,
  table_prefix,
  url
){
  data.world::query(
    data.world::qry_sql(
      sprintf(
        "SELECT * FROM %s_%s”,
        table_prefix, year
      )
    ),
    dataset = url
  ) %>%
    ## As long as we're customizing…
    select(parkname, value) %>%
    mutate(year = year)
}
```

allows us to do the same for tent, backcountry data

## STEP 2: EXTRACT EACH DATASET

```
## Download recreation data
url_recvisits <-
  "https://data.world/nps/annual-park-ranking-recreation-visits"

rec_visits <- map_df(
  .x = 2007:2016,
  .f = download_nps_year,
  table_prefix = "recreation_visits",
  url = url_recvisits
)
```

what we’re iterating over

what we’re doing

what stays the same each time

```
## Same thing for backcountry, tent campers
```

# USING MAP + USER-DEFINED FUNCTIONS

## STEP 1: WRITE FUNCTION

```
download_nps_year <- function(
  year = 2007:2016,
  table_prefix,
  url
){
  data.world::query(
    data.world::qry_sql(
      sprintf(
        "SELECT * FROM %s_%s”,
        table_prefix, year
      )
    ),
    dataset = url
  ) %>%
    ## As long as we're customizing…
    select(parkname, value) %>%
    mutate(year = year)
}
```

allows us to do the same for tent, backcountry data

## STEP 2: EXTRACT EACH DATASET

```
## Download recreation data
url_recvisits <-
  "https://data.world/nps/annual-park-ranking-recreation-visits"

rec_visits <- map_df(
  .x = 2007:2016,
  .f = download_nps_year,
  table_prefix = "recreation_visits",
  url = url_recvisits
)
```

what we're iterating over

what we're doing

what stays the same each time

```
## Same thing for backcountry, tent campers

## Download tent camper data
url_tent <-
  “https://data.world/nps/annual-
park-ranking-tent-campers"

tent_visits <- map_df(
  .x = 2007:2016,
  .f = download_nps_year,
  table_prefix = "tent_campers",
  url = url_tent
)
```

# USING MAP + USER-DEFINED FUNCTIONS

## STEP 1: WRITE FUNCTION

```
download_nps_year <- function(
  year = 2007:2016,
  table_prefix,
  url
){
  data.world::query(
    data.world::qry_sql(
      sprintf(
        "SELECT * FROM %s_%s”,
        table_prefix, year
      )
    ),
    dataset = url
  ) %>%
    ## As long as we're customizing…
    select(parkname, value) %>%
    mutate(year = year)
}
```

allows us to do the same for tent, backcountry data

## STEP 2: EXTRACT EACH DATASET

```
## Download recreation data
url_recvisits <-
  "https://data.world/nps/annual-park-ranking-recreation-visits"

rec_visits <- map_df(
  .x = 2007:2016,
  .f = download_nps_year,
  table_prefix = "recreation_visits",
  url = url_recvisits
)
```

what we’re iterating over

what we’re doing

what stays the same each time

```
## Same thing for backcountry, tent campers
```

```
## Download tent camper data
url_tent <-
  “https://data.world/nps/annual-park-ranking-tent-campers"

tent_visits <- map_df(
  .x = 2007:2016,
  .f = download_nps_year,
  table_prefix = "tent_campers",
  url = url_tent
)
```

```
## Download backcountry data
url_back <-
  “https://data.world/nps/annual-park-ranking-backcountry-campers"

back_visits <- map_df(
  .x = 2007:2016,
  .f = download_nps_year,
  table_prefix = “backcountry_campers",
  url = url_back
)
```

# WHAT'D WE GET?

## TOTAL RECREATIONAL VISITS

```
dplyr::sample_n(rec_visits, size = 10)
```

| parkname <chr> | value <int> | year <int> |
|---|---|---|
| Fort Stanwix NM | 86678 | 2015 |
| Pictured Rocks NL | 593587 | 2012 |
| Klondike Gold Rush NHP Alaska | 975043 | 2007 |
| Jimmy Carter NHS | 62057 | 2014 |
| Pipestone NM | 70748 | 2015 |
| Cowpens NB | 206740 | 2015 |
| Cumberland Island NS | 91996 | 2010 |
| Lassen Volcanic NP | 536068 | 2016 |
| Cape Hatteras NS | 2237378 | 2007 |
| Channel Islands NP | 360806 | 2007 |

1-10 of 20 rows    Previous 1 2 Next

# WHAT'D WE GET?

## TOTAL RECREATIONAL VISITS

```
dplyr::sample_n(rec_visits, size = 10)
```

| parkname <chr> | value <int> | year <int> |
|---|---|---|
| Fort Stanwix NM | 86678 | 2015 |
| Pictured Rocks NL | 593587 | 2012 |
| Klondike Gold Rush NHP Alaska | 975043 | 2007 |
| Jimmy Carter NHS | 62057 | 2014 |
| Pipestone NM | 70748 | 2015 |
| Cowpens NB | 206740 | 2015 |
| Cumberland Island NS | 91996 | 2010 |
| Lassen Volcanic NP | 536068 | 2016 |
| Cape Hatteras NS | 2237378 | 2007 |
| Channel Islands NP | 360806 | 2007 |

1-10 of 20 rows Previous 1 2 Next

*We're skipping some data management which*

1. *Restricts all our data to national parks only*
2. *Determines the region each park is in*

# CREATE THE FINAL LIST THAT STARTS IT ALL

# CREATE THE FINAL LIST THAT STARTS IT ALL

*This* `map()` *call iterates over our three separate datasets, merges park region onto each, and gives us a* ***list*** *as our final result*

```
datalist <- map(
  ## Initial list = all three datasets
  .x = list(rec_visits, tent_visits, back_visits),
  ## For each, reduce() uses left_join to merge on state/region by parkname
  .f = ~ purrr::reduce(list(., park_index), left_join, by = "parkname")
)
```

# CREATE THE FINAL LIST THAT STARTS IT ALL

*This* `map()` *call iterates over our three separate datasets, merges park region onto each, and gives us a* ***list*** *as our final result*

```
datalist <- map(
  ## Initial list = all three datasets
  .x = list(rec_visits, tent_visits, back_visits),
  ## For each, reduce() uses left_join to merge on state/region by parkname
  .f = ~ purrr::reduce(list(., park_index), left_join, by = "parkname")
)
```

particularly handy if you have >2 data.frames to merge!

# CREATE THE FINAL LIST THAT STARTS IT ALL

*This `map()` call iterates over our three separate datasets, merges park region onto each, and gives us a **list** as our final result*

```
datalist <- map(
  ## Initial list = all three datasets
  .x = list(rec_visits, tent_visits, back_visits),
  ## For each, reduce() uses left_join to merge on state/region by parkname
  .f = ~ purrr::reduce(list(., park_index), left_join, by = "parkname")
)
```

particularly handy if you have >2 data.frames to merge!

```
> head(datalist[[1]])
# A tibble: 6 x 7
             parkname  value  year                name  type location       region
                <chr>  <int> <int>               <chr> <chr>    <chr>       <fctr>
1      Kings Canyon NP 580129  2007      Kings Canyon    NP       CA   Pacific NW
2    Virgin Islands NP 571382  2007    Virgin Islands    NP       VI   Eastern US
3 Petrified Forest NP 563590  2007 Petrified Forest    NP       AZ Intermountain
4      Capitol Reef NP 554907  2007      Capitol Reef    NP       UT Intermountain
5        Mesa Verde NP 541102  2007        Mesa Verde    NP       CO Intermountain
6          Biscayne NP 517442  2007          Biscayne    NP       FL   Eastern US
```

# CREATE THE FINAL LIST THAT STARTS IT ALL

*This `map()` call iterates over our three separate datasets, merges park region onto each, and gives us a **list** as our final result*

```
datalist <- map(
  ## Initial list = all three datasets
  .x = list(rec_visits, tent_visits, back_visits),
  ## For each, reduce() uses left_join to merge on state/region by parkname
  .f = ~ purrr::reduce(list(., park_index), left_join, by = "parkname")
)
```

particularly handy if you have >2 data.frames to merge!

**AUDIENCE PARTICIPATION!**

IF YOU'RE USING PURRR_DATA.RDATA, JUMP IN HERE!

```
> head(datalist[[1]])
# A tibble: 6 x 7
            parkname  value  year              name  type location       on
               <chr>  <int> <int>             <chr> <chr>    <chr>   <fctr>
1     Kings Canyon NP 580129  2007      Kings Canyon    NP       CA   Pacific NW
2   Virgin Islands NP 571382  2007    Virgin Islands    NP       VI   Eastern US
3 Petrified Forest NP 563590  2007  Petrified Forest    NP       AZ Intermountain
4     Capitol Reef NP 554907  2007      Capitol Reef    NP       UT Intermountain
5       Mesa Verde NP 541102  2007        Mesa Verde    NP       CO Intermountain
6         Biscayne NP 517442  2007          Biscayne    NP       FL   Eastern US
```

# LET'S RUN SOME MODELS!

## PREDICT # VISITORS BY YEAR, REGION, INTERACTION

# LET'S RUN SOME MODELS!

## PREDICT # VISITORS BY YEAR, REGION, INTERACTION

`datalist`: list of data.frames

# LET'S RUN SOME MODELS!

## PREDICT # VISITORS BY YEAR, REGION, INTERACTION

`datalist`: list of data.frames

- REC VISITS DATA
- TENT CAMPER DATA
- BACK-COUNTRY CAMPER DATA

```
map(datalist, ~ lm(value = year * region, data = .))
```

# LET'S RUN SOME MODELS!

## PREDICT # VISITORS BY YEAR, REGION, INTERACTION

`datalist`: list of data.frames

- REC VISITS DATA
- TENT CAMPER DATA
- BACK-COUNTRY CAMPER DATA

```
map(datalist, ~ lm(value = year * region, data = .))
```

`orgmod_list`: list of `lm` fits

- REC VISITS MODEL
- TENT CAMPER MODEL
- BACK-COUNTRY CAMPER MODEL

# NO ERRORS MEANS WE'RE GOOD, RIGHT?

## ANY STATISTICIAN WILL TELL YOU IT'S NOT THAT EASY

# NO ERRORS MEANS WE'RE GOOD, RIGHT?

## ANY STATISTICIAN WILL TELL YOU IT'S NOT THAT EASY

- **Goal**: Check model assumptions by looking at residual vs fitted plots
- **Strategy**: Use `purrr::walk()` to iterate over all our model fits, extract residuals and fitted values, and plot them

# NO ERRORS MEANS WE'RE GOOD, RIGHT?

## ANY STATISTICIAN WILL TELL YOU IT'S NOT THAT EASY

- **Goal**: Check model assumptions by looking at residual vs fitted plots
- **Strategy**: Use `purrr::walk()` to iterate over all our model fits, extract residuals and fitted values, and plot them

**`walk()`** is very similar to **`map()`**, but we use `walk` when we want **side effects** - printed output, plots, saved files, etc - rather than an **object** returned

# NO ERRORS MEANS WE'RE GOOD, RIGHT?

## ANY STATISTICIAN WILL TELL YOU IT'S NOT THAT EASY

- **Goal**: Check model assumptions by looking at residual vs fitted plots
- **Strategy**: Use `purrr::walk()` to iterate over all our model fits, extract residuals and fitted values, and plot them

**`walk()`** is very similar to **`map()`**, but we use `walk` when we want **side effects** - printed output, plots, saved files, etc - rather than an **object** returned

```
par(mfrow = c(1, 3))
walk(
  orgmod_list,
  ~ plot(resid(.) ~ predict(.))
)
```

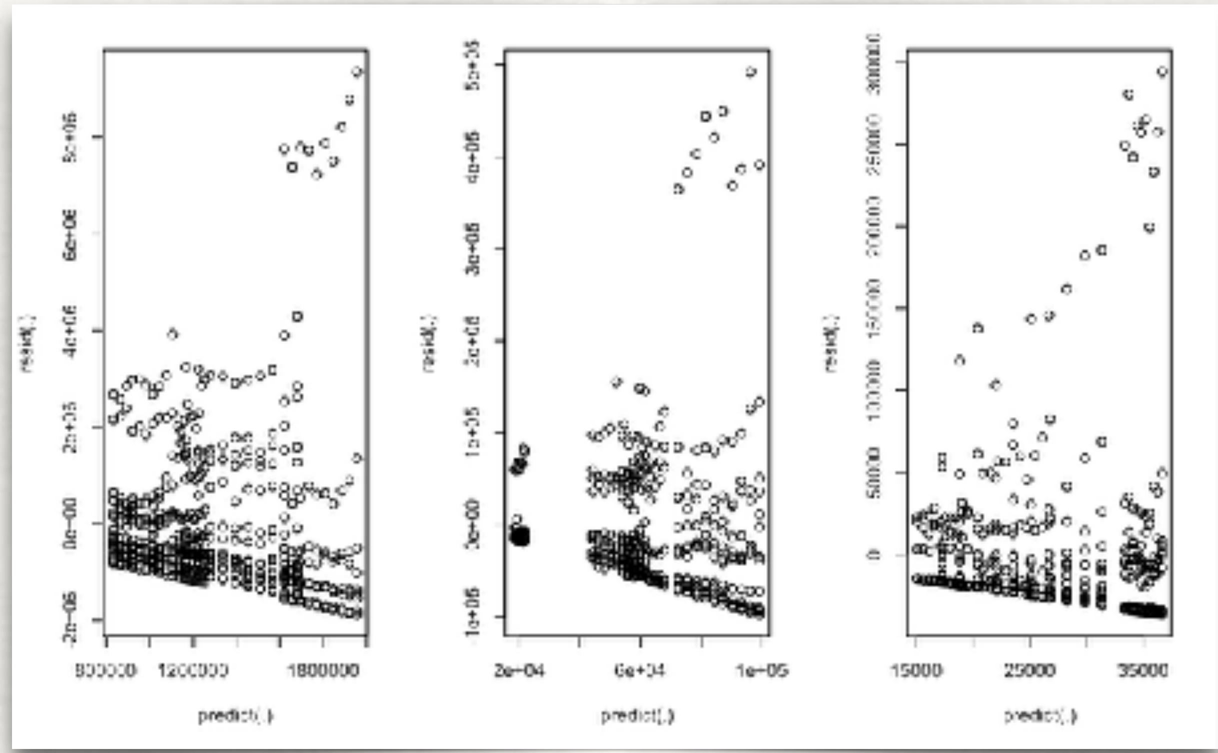
# NO ERRORS MEANS WE'RE GOOD, RIGHT?

## ANY STATISTICIAN WILL TELL YOU IT'S NOT THAT EASY

- **Goal**: Check model assumptions by looking at residual vs fitted plots
- **Strategy**: Use `purrr::walk()` to iterate over all our model fits, extract residuals and fitted values, and plot them

**`walk()`** is very similar to **`map()`**, but we use `walk` when we want **side effects** - printed output, plots, saved files, etc - rather than an **object** returned

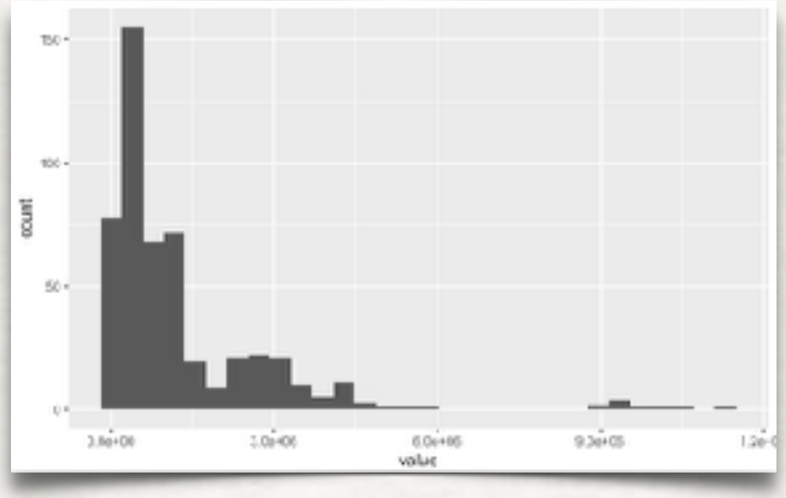
```
par(mfrow = c(1, 3))
walk(
  orgmod_list,
  ~ plot(resid(.) ~ predict(.))
)
```

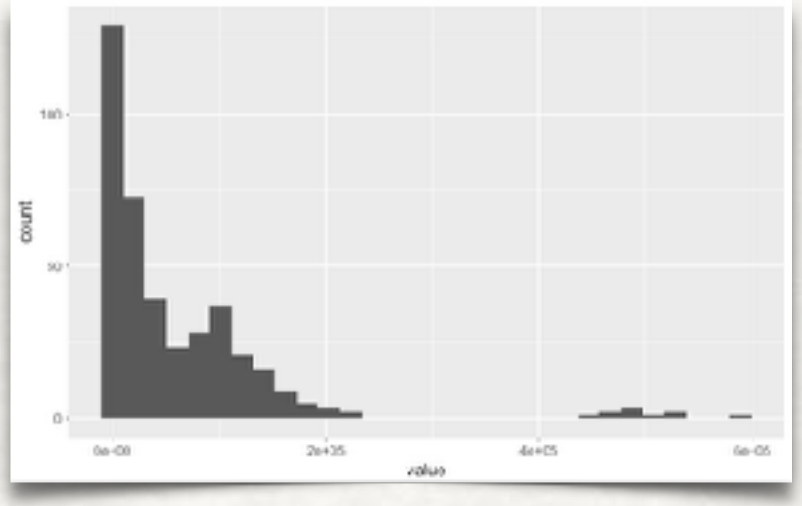


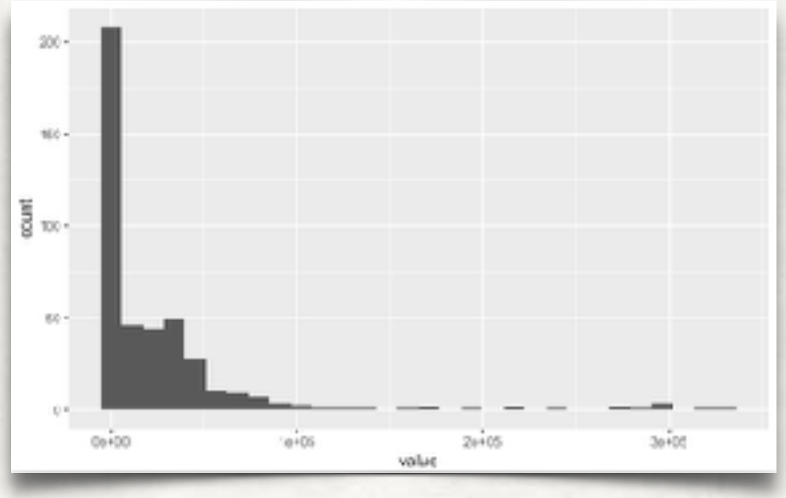
# USE PURRR TO FIX IT

Diagnose the problem: Use **`walk()`** to look at the distribution of our outcome

```
walk(
  datalist,
  ~ print(ggplot(data = ., aes(x = value)) + geom_histogram())
)
```




Perhaps a log transformation would be helpful?

# USE PURRR TO FIX IT

# USE PURRR TO FIX IT

Use **`map()`** (and dplyr) to transform our outcome variable in place

# USE PURRR TO FIX IT

Use **`map()`** (and dplyr) to transform our outcome variable in place

```
datalist <- datalist %>%
  map(~ dplyr::mutate_at(.x, "value", log))
```

# USE PURRR TO FIX IT

Use **map()** (and dplyr) to transform our outcome variable in place

```
datalist <- datalist %>%
  map(~ dplyr::mutate_at(.x, "value", log))
```

Use **map()** to refit the linear model with our transformed outcome, then recheck RP plots

# USE PURRR TO FIX IT

Use **map()** (and dplyr) to transform our outcome variable in place

```
datalist <- datalist %>%
  map(~ dplyr::mutate_at(.x, "value", log))
```

Use **map()** to refit the linear model with our transformed outcome, then recheck RP plots

```
logmod_list <-
  map(datalist, ~ lm(value ~ year * region, data = .))

par(mfrow = c(1, 3))
walk(
  logmod_list,
  ~ plot(resid(.) ~ predict(.))
)
```

# USE PURRR TO FIX IT

Use **map()** (and dplyr) to transform our outcome variable in place

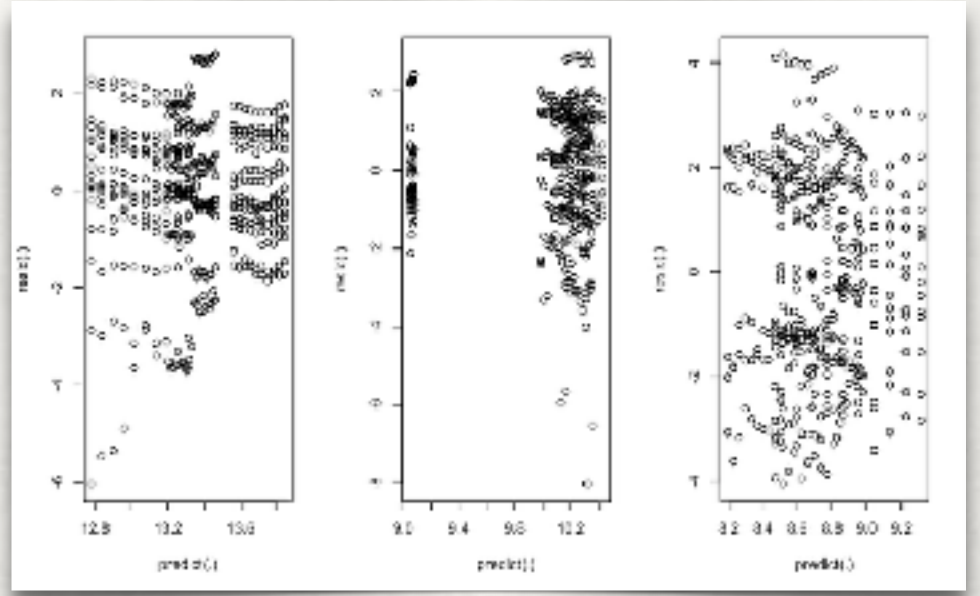
```
datalist <- datalist %>%
  map(~ dplyr::mutate_at(.x, "value", log))
```

Use **map()** to refit the linear model with our transformed outcome,

then recheck RP plots

```
logmod_list <-
  map(datalist, ~ lm(value ~ year * region, data = .))

par(mfrow = c(1, 3))
walk(
  logmod_list,
  ~ plot(resid(.) ~ predict(.))
)
```



# TIME FOR SOME NUMBERS

Let's quickly look at the $R^2$ for each of our models. Know how we can do that?

**AUDIENCE PARTICIPATION!**

HOW CAN WE DO THAT?

# TIME FOR SOME NUMBERS

Let's quickly look at the $R^2$ for each of our models. Know how we can do that?

**AUDIENCE PARTICIPATION!**

HOW CAN WE DO THAT?

`map_dbl!`

# TIME FOR SOME NUMBERS

Let's quickly look at the $R^2$ for each of our models. Know how we can do that?

**AUDIENCE PARTICIPATION!**

HOW CAN WE DO THAT?

**`map_dbl!`**

One-line method:

```
round(map_dbl(logmod_list, ~ summary(.)$adj.r.squared), 2)
```

# TIME FOR SOME NUMBERS

Let's quickly look at the $R^2$ for each of our models. Know how we can do that?

**AUDIENCE PARTICIPATION!**

HOW CAN WE DO THAT?

**map_dbl!**

One-line method:

```
round(map_dbl(logmod_list, ~ summary(.)$adj.r.squared), 2)
```

Pipe method:

```
logmod_list %>%
  map(summary) %>%
  map_dbl(.f = “adj.r.squared") %>%
  round(2)
```

# TIME FOR SOME NUMBERS

Let's quickly look at the $R^2$ for each of our models. Know how we can do that?

**AUDIENCE PARTICIPATION!**

HOW CAN WE DO THAT?

**map_dbl!**

One-line method:

```
round(map_dbl(logmod_list, ~ summary(.)$adj.r.squared), 2)
```

Pipe method:

```
logmod_list %>%
  map(summary) %>%
  map_dbl(.f = "adj.r.squared") %>%
  round(2)
```

From each element of .x, take the element named "adj.r.squared"

# TIME FOR SOME NUMBERS

Let's quickly look at the $R^2$ for each of our models. Know how we can do that?

**AUDIENCE PARTICIPATION!**

HOW CAN WE DO THAT?

`map_dbl!`

One-line method:

```
round(map_dbl(logmod_list, ~ summary(.)$adj.r.squared), 2)
```

Pipe method:

```
logmod_list %>%
  map(summary) %>%
  map_dbl(.f = "adj.r.squared") %>%
  round(2)
```

From each element of .x, take the element named "adj.r.squared"

Either way:

```
[1] 0.03 0.05 0.00
```

# VISUALIZE RESULTS

# VISUALIZE RESULTS

For each model, we're going to:

- Create a `data.frame` with predicted values for each year and region
- Plot visitors over time, faceted by region
- Save those plots

# VISUALIZE RESULTS

For each model, we're going to:

- Create a `data.frame` with predicted values for each year and region
- Plot visitors over time, faceted by region
- Save those plots

Points to emphasize:

- `purrr::cross()` for getting all **combinations** of things
- `purrr::pluck()` for **extracting** elements from a list
- using `purrr::map()` in a **pipeline**, starting with one list and taking it through multiple steps

# VISUALIZATION PREP

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

```
pred_list <- logmod_list %>%
```

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

```
pred_list <- logmod_list %>%
  map(.f = predict, newdata = preddata, se.fit = TRUE) %>%
```

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

```
pred_list <- logmod_list %>%
  map(.f = predict, newdata = preddata, se.fit = TRUE) %>%
  map(~ data.frame(fit = pluck(., "fit"), se = .$se.fit) %>%
```

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

```
pred_list <- logmod_list %>%
  map(.f = predict, newdata = preddata, se.fit = TRUE) %>%
  map(~ data.frame(fit = pluck(., "fit"), se = .$se.fit) %>%
          mutate(lcl = fit - qnorm(0.975) * se,
```

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

```
pred_list <- logmod_list %>%
  map(.f = predict, newdata = preddata, se.fit = TRUE) %>%
  map(~ data.frame(fit = pluck(., "fit"), se = .$se.fit) %>%
        mutate(lcl = fit - qnorm(0.975) * se,
               ucl = fit + qnorm(0.975) * se)) %>%
```

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

```
pred_list <- logmod_list %>%
  map(.f = predict, newdata = preddata, se.fit = TRUE) %>%
  map(~ data.frame(fit = pluck(., "fit"), se = .$se.fit) %>%
        mutate(lcl = fit - qnorm(0.975) * se,
               ucl = fit + qnorm(0.975) * se)) %>%
  ## Add year and region onto each
```

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

```
pred_list <- logmod_list %>%
  map(.f = predict, newdata = preddata, se.fit = TRUE) %>%
  map(~ data.frame(fit = pluck(., "fit"), se = .$se.fit) %>%
        mutate(lcl = fit - qnorm(0.975) * se,
               ucl = fit + qnorm(0.975) * se)) %>%
  ## Add year and region onto each
  map(dplyr::bind_cols, preddata)
```

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

```
pred_list <- logmod_list %>%
  map(.f = predict, newdata = preddata, se.fit = TRUE) %>%
  map(~ data.frame(fit = pluck(., "fit"), se = .$se.fit) %>%
          mutate(lcl = fit - qnorm(0.975) * se,
                 ucl = fit + qnorm(0.975) * se)) %>%
  ## Add year and region onto each
  map(dplyr::bind_cols, preddata)
```

List of lm fits ->

List of lists! ->

List of data.frames we transformed ->

List of data.frames we added year/region to

# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

```
pred_list <- logmod_list %>%
  map(.f = predict, newdata = preddata, se.fit = TRUE) %>%
  map(~ data.frame(fit = pluck(., "fit"), se = .$se.fit) %>%
        mutate(lcl = fit - qnorm(0.975) * se,
               ucl = fit + qnorm(0.975) * se)) %>%
  ## Add year and region onto each
  map(dplyr::bind_cols, preddata)
```

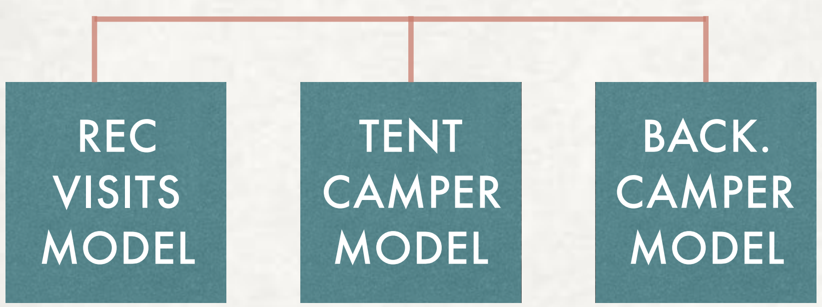
List of `lm` fits ->

List of lists! ->

List of `data.frames` we transformed ->

List of `data.frames` we added year/region to

`logmod_list`: list of lm fits



# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

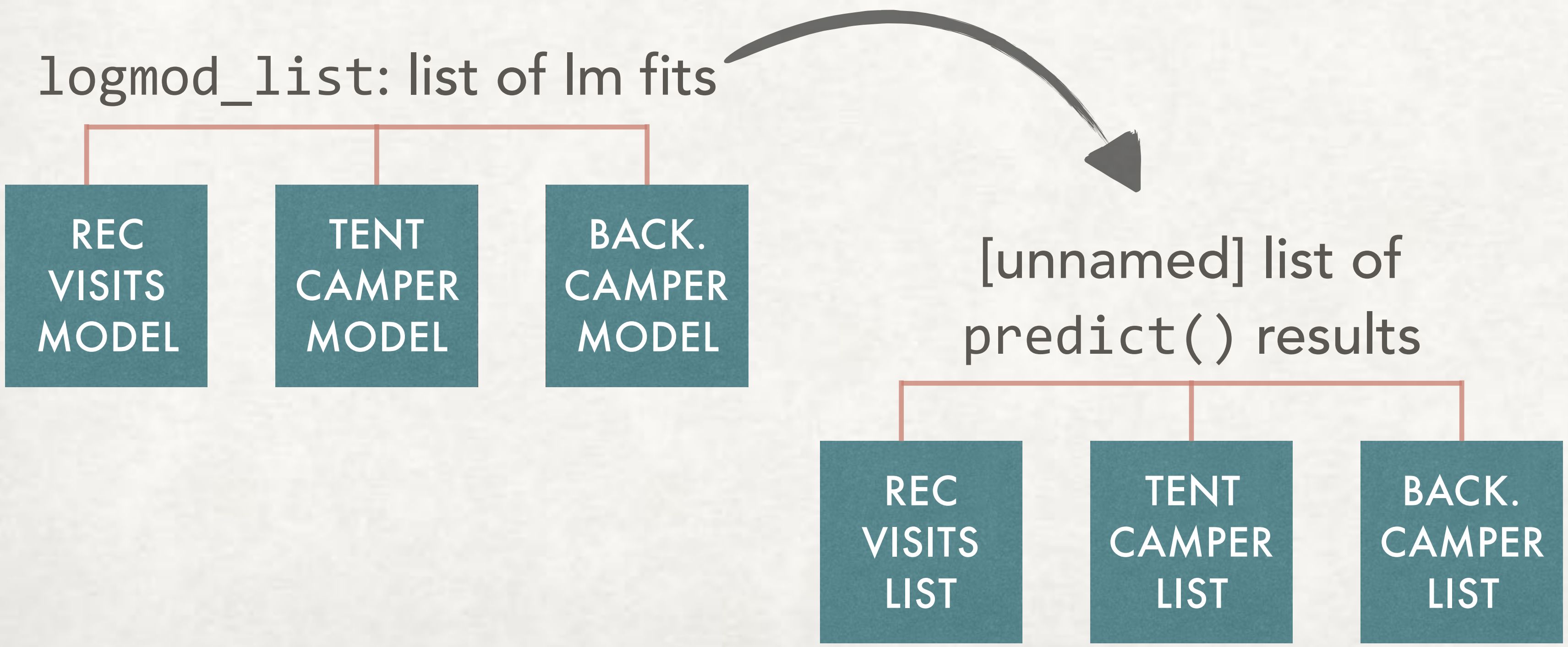
```
pred_list <- logmod_list %>%
  map(.f = predict, newdata = preddata, se.fit = TRUE) %>%
  map(~ data.frame(fit = pluck(., "fit"), se = .$se.fit) %>%
        mutate(lcl = fit - qnorm(0.975) * se,
               ucl = fit + qnorm(0.975) * se)) %>%
  ## Add year and region onto each
  map(dplyr::bind_cols, preddata)
```

List of lm fits ->

List of lists! ->

List of data.frames we transformed ->

List of data.frames we added year/region to



# VISUALIZATION PREP

Create a data.frame of all possible combinations of year and region

```
preddata <- cross_df(
  .l = list("year" = unique(pluck(datalist, 1, "year")),
            "region" = levels(datalist[[1]]$region))
)
```

For each visit type, create data.frame with predicted # for each year/region

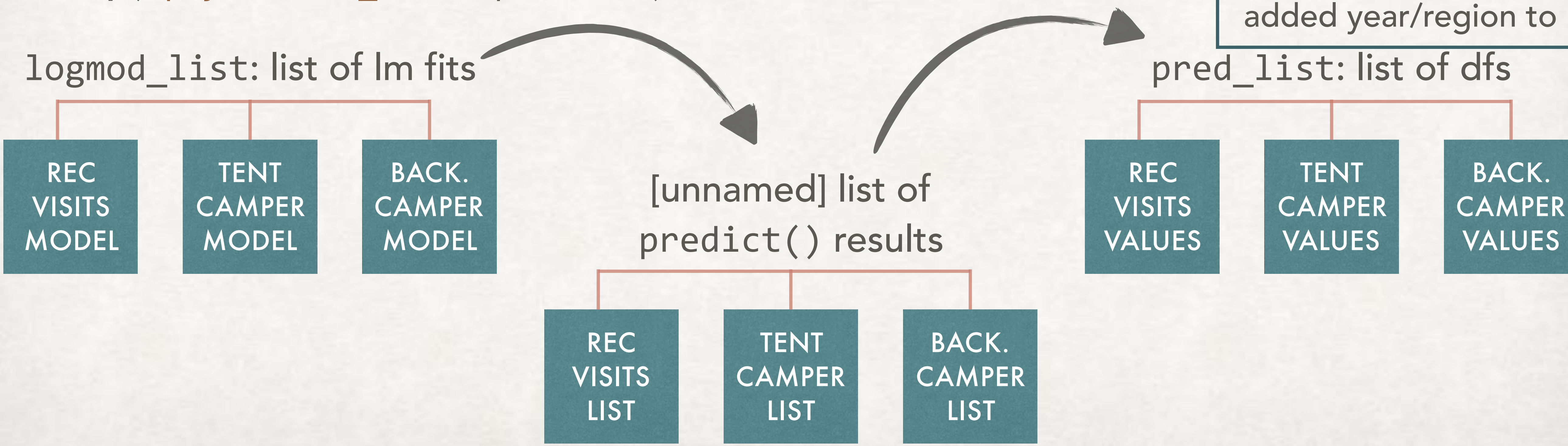
```
pred_list <- logmod_list %>%
  map(.f = predict, newdata = preddata, se.fit = TRUE) %>%
  map(~ data.frame(fit = pluck(., "fit"), se = .$se.fit) %>%
        mutate(lcl = fit - qnorm(0.975) * se,
               ucl = fit + qnorm(0.975) * se)) %>%
  ## Add year and region onto each
  map(dplyr::bind_cols, preddata)
```

List of lm fits ->

List of lists! ->

List of data.frames we transformed ->

List of data.frames we added year/region to



# VISUALIZATION PREP

**AUDIENCE PARTICIPATION!**

We want to make very similar charts for each type of visitor, but we want a few things to be different. What do you think we should do first?

# VISUALIZATION PREP

**AUDIENCE PARTICIPATION!**

We want to make very similar charts for each type of visitor, but we want a few things to be different. What do you think we should do first?

**Write a function!**

# VISUALIZATION PREP

**AUDIENCE PARTICIPATION!**

We want to make very similar charts for each type of visitor, but we want a few things to be different. What do you think we should do first?

**Write a function!**

```
plot_predicted <- function(df, vscale, maintitle){
  ## Make sure df has all the columns we need
  if(!all(c("fit", "se", "lcl", "ucl", "year", "region") %in% names(df))){
    stop("df should have columns fit, se, lcl, ucl, year, region")
  }

  ## Create a plot faceted by region
  p <- ggplot(data = df, aes(x = year, y = fit)) +
    facet_wrap(~ region, nrow = 2) +
    geom_ribbon(aes(ymin = lcl, ymax = ucl, fill = region), alpha = 0.4) +
    geom_line(aes(color = region), size = 2) +
    scale_fill_viridis(option = vscale, discrete = TRUE, end = 0.75) +
    scale_colour_viridis(option = vscale, discrete = TRUE, end = 0.75) +
    labs(title = maintitle,
         x = NULL, y = "Log(Visitors)") +
    theme(legend.position = "none")

  return(p)

}
```

# VISUALIZATION PREP

**AUDIENCE PARTICIPATION!**

We want to make very similar charts for each type of visitor, but we want a few things to be different. What do you think we should do first?

**Write a function!**

```
plot_predicted <- function(df, vscale, maintitle){
  ## Make sure df has all the columns we need
  if(!all(c("fit", "se", "lcl", "ucl", "year", "region") %in% names(df))){
    stop("df should have columns fit, se, lcl, ucl, year, region")
  }

  ## Create a plot faceted by region
  p <- ggplot(data = df, aes(x = year, y = fit)) +
    facet_wrap(~ region, nrow = 2) +
    geom_ribbon(aes(ymin = lcl, ymax = ucl, fill = region), alpha = 0.4) +
    geom_line(aes(color = region), size = 2) +
    scale_fill_viridis(option = vscale, discrete = TRUE, end = 0.75) +
    scale_colour_viridis(option = vscale, discrete = TRUE, end = 0.75) +
    labs(title = maintitle,
         x = NULL, y = "Log(Visitors)") +
    theme(legend.position = "none")

  return(p)

}
```

Three things can change; everything else remains constant

# VISUALIZE RESULTS

## THREE ARGUMENTS = BREAK OUT THE BIG GUNS

Time for some **parallel mapping!**

First, let's set up a list of arguments.

# VISUALIZE RESULTS

## THREE ARGUMENTS = BREAK OUT THE BIG GUNS

Time for some **parallel mapping!**

First, let's set up a list of arguments.

```
plot_args <- list(
```

# VISUALIZE RESULTS

## THREE ARGUMENTS = BREAK OUT THE BIG GUNS

Time for some **parallel mapping!**

First, let's set up a list of arguments.

```
plot_args <- list(
  "df" = pred_list,
```

# VISUALIZE RESULTS

## THREE ARGUMENTS = BREAK OUT THE BIG GUNS

Time for some **parallel mapping!**

First, let's set up a list of arguments.

```
plot_args <- list(
  "df" = pred_list,
  "vscale" = c("D", "A", "C"),
```

# VISUALIZE RESULTS

## THREE ARGUMENTS = BREAK OUT THE BIG GUNS

Time for some **parallel mapping!**

First, let's set up a list of arguments.

```
plot_args <- list(
  "df" = pred_list,
  "vscale" = c("D", "A", "C"),
  "maintitle" = c("Total Recreational Visits",
```

# VISUALIZE RESULTS

## THREE ARGUMENTS = BREAK OUT THE BIG GUNS

Time for some **parallel mapping!**

First, let's set up a list of arguments.

```
plot_args <- list(
  "df" = pred_list,
  "vscale" = c("D", "A", "C"),
  "maintitle" = c("Total Recreational Visits",
                  "Tent Campers",
```

# VISUALIZE RESULTS

## THREE ARGUMENTS = BREAK OUT THE BIG GUNS

Time for some **parallel mapping!**

First, let's set up a list of arguments.

```
plot_args <- list(
  "df" = pred_list,
  "vscale" = c("D", "A", "C"),
  "maintitle" = c("Total Recreational Visits",
                  "Tent Campers",
                  "Backcountry Campers")
```

# VISUALIZE RESULTS

## THREE ARGUMENTS = BREAK OUT THE BIG GUNS

Time for some **parallel mapping!**

First, let's set up a list of arguments.

```
plot_args <- list(
  "df" = pred_list,
  "vscale" = c("D", "A", "C"),
  "maintitle" = c("Total Recreational Visits",
                  "Tent Campers",
                  "Backcountry Campers")
)
```

# VISUALIZE RESULTS

## THREE ARGUMENTS = BREAK OUT THE BIG GUNS

Time for some **parallel mapping!**

First, let's set up a list of arguments.

```
plot_args <- list(
  "df" = pred_list,
  "vscale" = c("D", "A", "C"),
  "maintitle" = c("Total Recreational Visits",
                  "Tent Campers",
                  "Backcountry Campers")
)
```

Once our plotting function is written and our arguments are set up, we can get all our plots with one line:

```
nps_plots <- pmap(plot_args, plot_predicted)
```

# WHAT JUST HAPPENED?

## A SCHEMATIC

# WHAT JUST HAPPENED?

## A SCHEMATIC

`pred_list`: list of dfs (`df`)

- REC VISITS VALUES
- TENT CAMPER VALUES
- BACK. CAMPER VALUES

vector of viridis color scale choices (`vscale`)

- REC VISITS = VIRIDIS
- TENT VISITS = MAGMA
- BACK. VISITS = PLASMA

vector of title strings (`maintitle`)

- TOTAL REC VISITS
- TENT CAMPERS
- BACK-COUNTRY CAMPERS

# WHAT JUST HAPPENED?

## A SCHEMATIC

`pred_list`: list of dfs (`df`)

- REC VISITS VALUES
- TENT CAMPER VALUES
- BACK. CAMPER VALUES

vector of title strings (`maintitle`)

- TOTAL REC VISITS
- TENT CAMPERS
- BACK-COUNTRY CAMPERS

vector of viridis color scale choices (`vscale`)

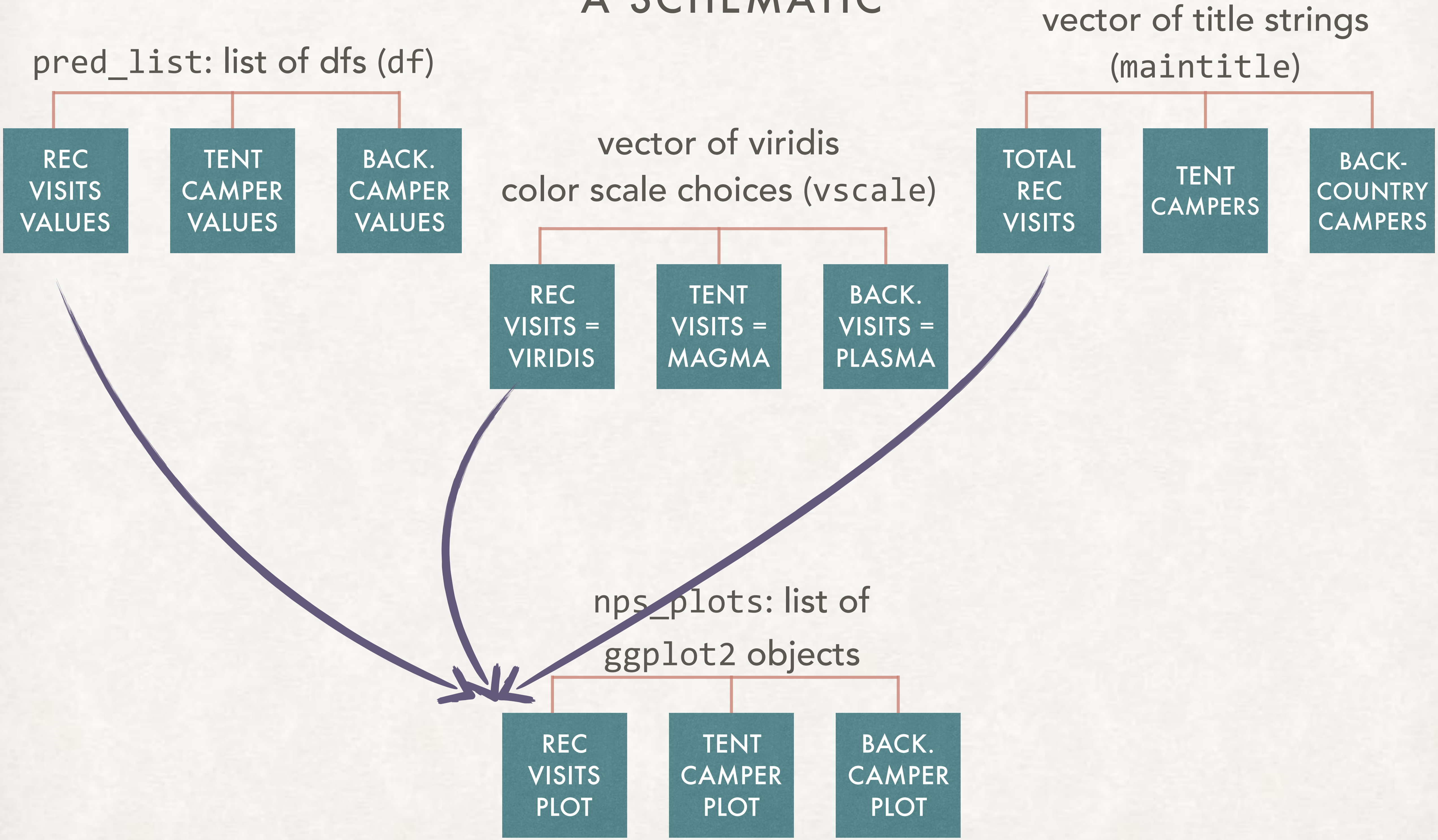
- REC VISITS = VIRIDIS
- TENT VISITS = MAGMA
- BACK. VISITS = PLASMA

`nps_plots`: list of `ggplot2` objects

- REC VISITS PLOT
- TENT CAMPER PLOT
- BACK. CAMPER PLOT

# WHAT JUST HAPPENED?

## A SCHEMATIC



# WHAT JUST HAPPENED?

## A SCHEMATIC

`pred_list`: list of dfs (`df`)

- REC VISITS VALUES
- TENT CAMPER VALUES
- BACK. CAMPER VALUES

vector of viridis color scale choices (`vscale`)

- REC VISITS = VIRIDIS
- TENT VISITS = MAGMA
- BACK. VISITS = PLASMA

vector of title strings (`maintitle`)

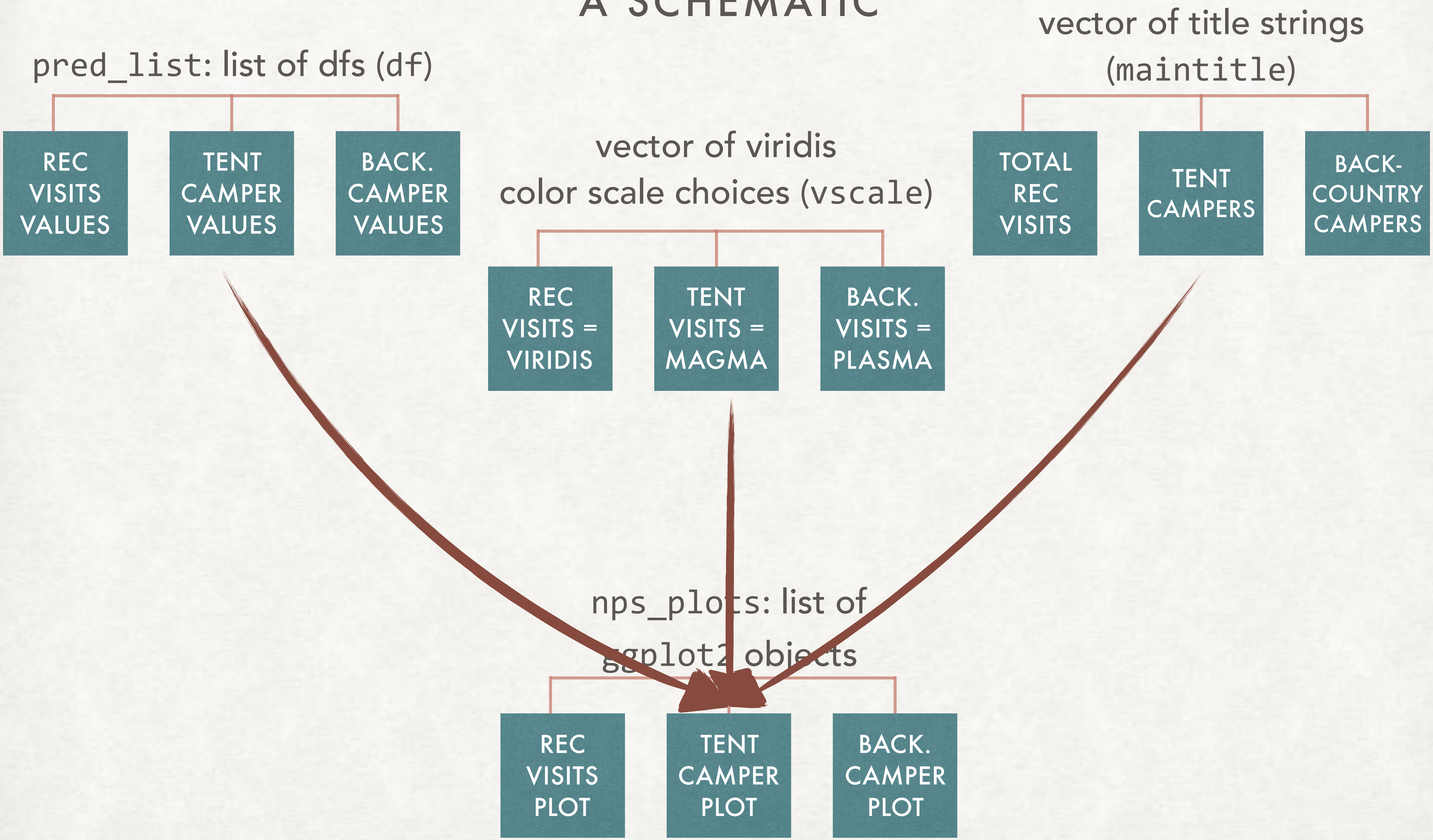
- TOTAL REC VISITS
- TENT CAMPERS
- BACK-COUNTRY CAMPERS

`nps_plots`: list of `ggplot2` objects

- REC VISITS PLOT
- TENT CAMPER PLOT
- BACK. CAMPER PLOT

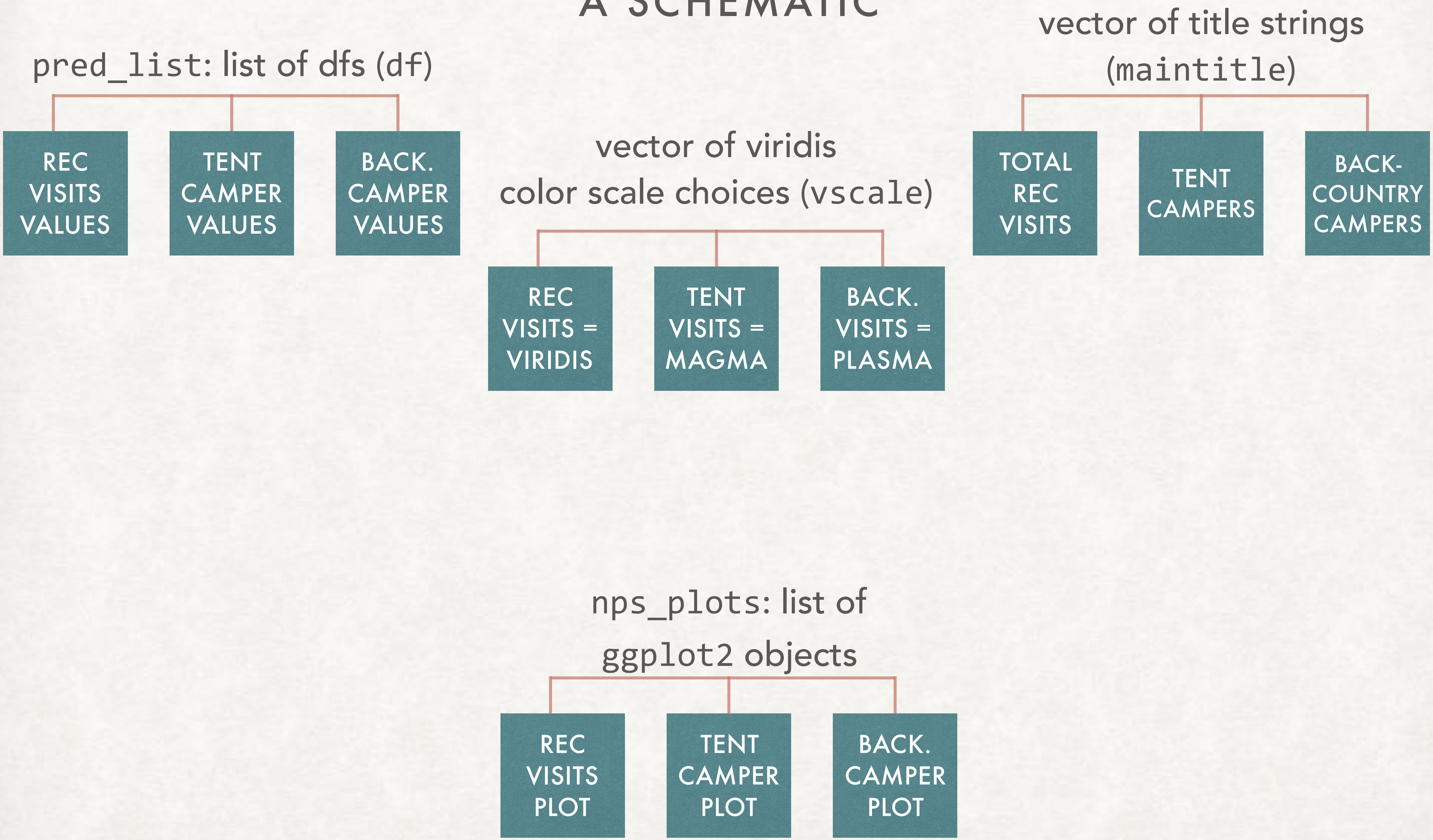
# WHAT JUST HAPPENED?

## A SCHEMATIC



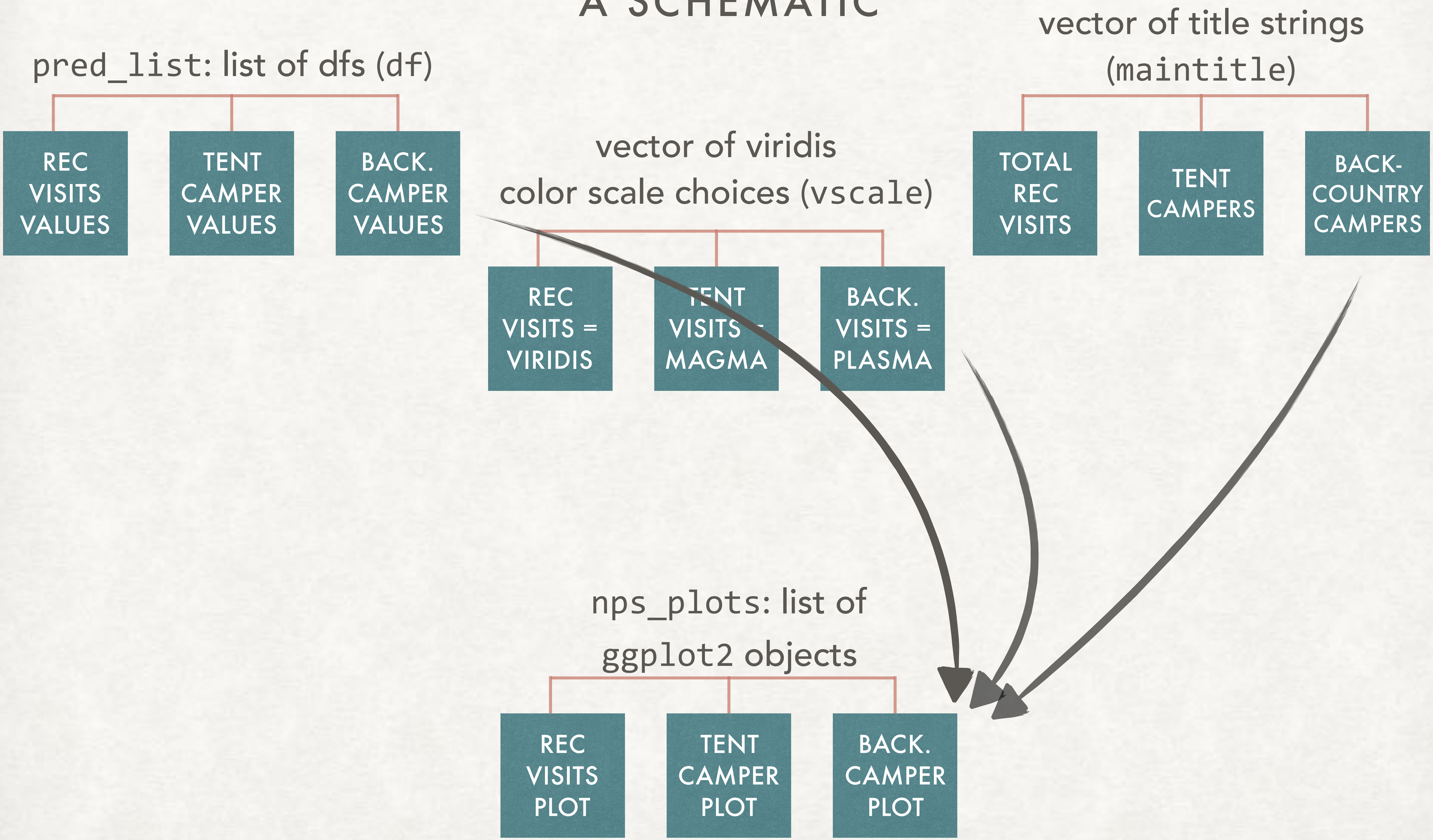
# WHAT JUST HAPPENED?

## A SCHEMATIC



# WHAT JUST HAPPENED?

## A SCHEMATIC



# WHAT JUST HAPPENED?

## A SCHEMATIC

pred_list: list of dfs (df)

- REC VISITS VALUES
- TENT CAMPER VALUES
- BACK. CAMPER VALUES

vector of viridis color scale choices (vscale)

- REC VISITS = VIRIDIS
- TENT VISITS = MAGMA
- BACK. VISITS = PLASMA

vector of title strings (maintitle)

- TOTAL REC VISITS
- TENT CAMPERS
- BACK-COUNTRY CAMPERS

nps_plots: list of ggplot2 objects

- REC VISITS PLOT
- TENT CAMPER PLOT
- BACK. CAMPER PLOT

# BUT... BUT... WHERE ARE THE PLOTS?!

## THEY ARE MERELY OBJECTS IN THE SKY FOR NOW

**AUDIENCE PARTICIPATION!**

Remember, `map()` functions return objects; in order to see our plots, we have to *print* them somehow. How can we do that?

# BUT... BUT... WHERE ARE THE PLOTS?!

## THEY ARE MERELY OBJECTS IN THE SKY FOR NOW

**AUDIENCE PARTICIPATION!**

Remember, `map()` functions return objects; in order to see our plots, we have to *print* them somehow. How can we do that?

```
walk2(.x = c("rec.pdf", "tent.pdf", "backcountry.pdf"),
      .y = nps_plots,
      ggsave,
      width = 8, height = 6)
```

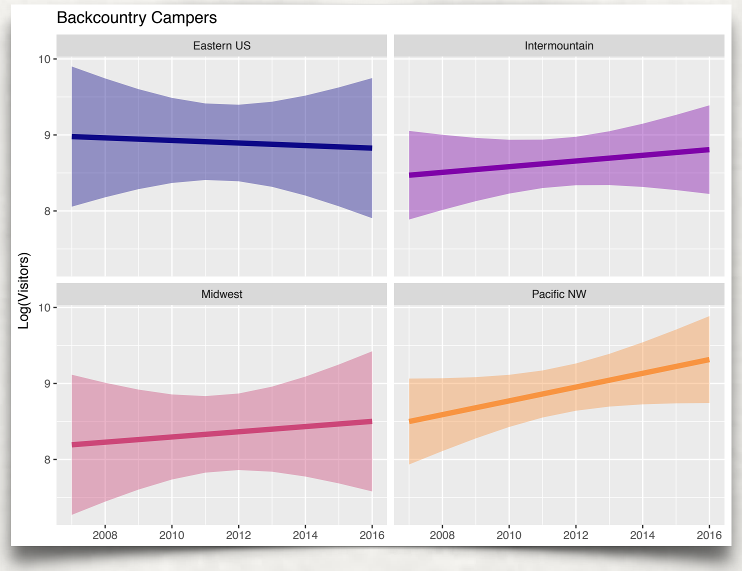
# BUT... BUT... WHERE ARE THE PLOTS?!

## THEY ARE MERELY OBJECTS IN THE SKY FOR NOW

**AUDIENCE PARTICIPATION!**

Remember, `map()` functions return objects; in order to see our plots, we have to *print* them somehow. How can we do that?

```
walk2(.x = c("rec.pdf", "tent.pdf", "backcountry.pdf"),
      .y = nps_plots,
      ggsave,
      width = 8, height = 6)
```



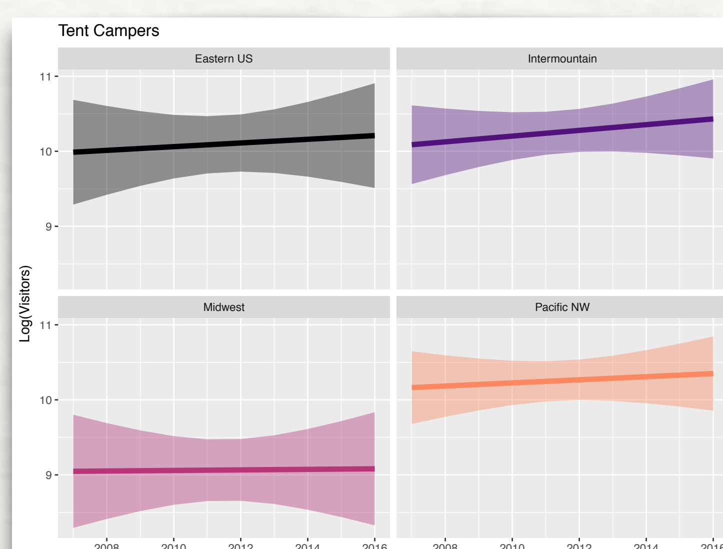
# BUT… BUT… WHERE ARE THE PLOTS?!

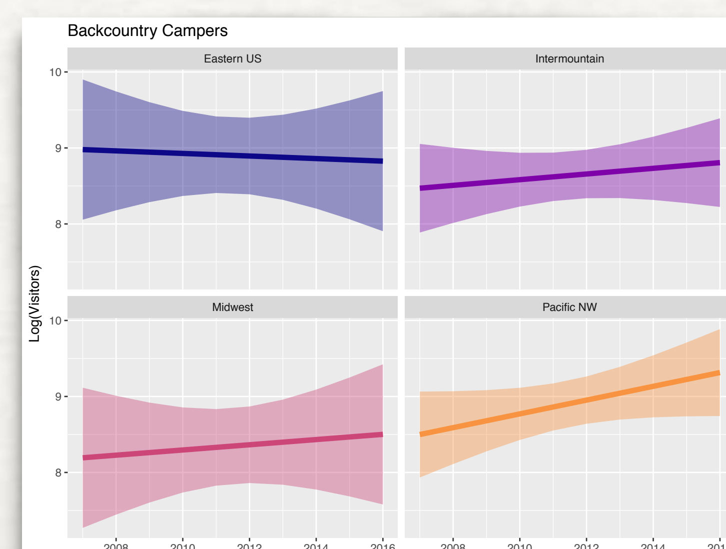
## THEY ARE MERELY OBJECTS IN THE SKY FOR NOW

**AUDIENCE PARTICIPATION!**

Remember, `map()` functions return objects; in order to see our plots, we have to *print* them somehow. How can we do that?

```
walk2(.x = c("rec.pdf", "tent.pdf", "backcountry.pdf"),
      .y = nps_plots,
      ggsave,
      width = 8, height = 6)
```



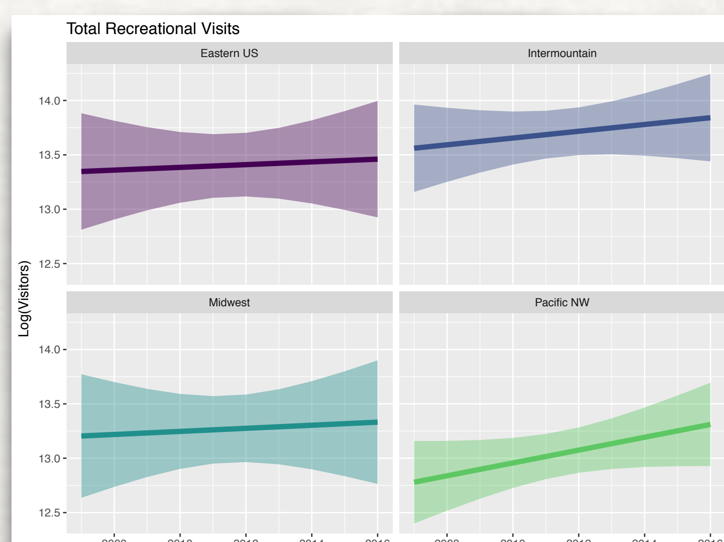
# BUT… BUT… WHERE ARE THE PLOTS?!

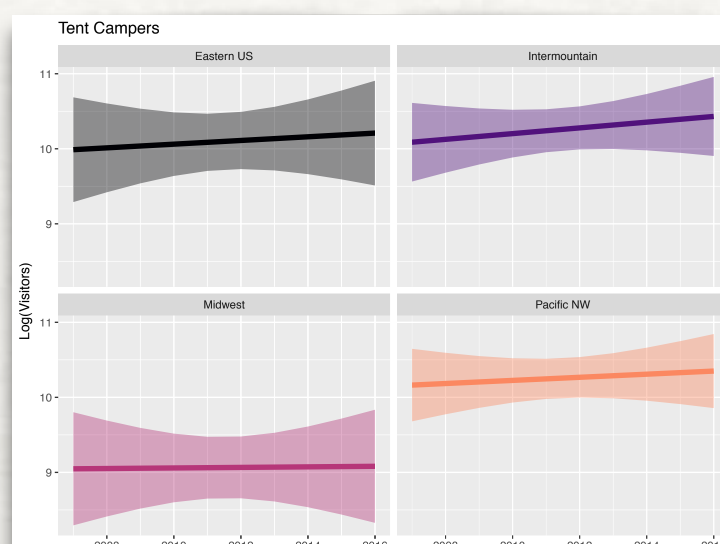
## THEY ARE MERELY OBJECTS IN THE SKY FOR NOW

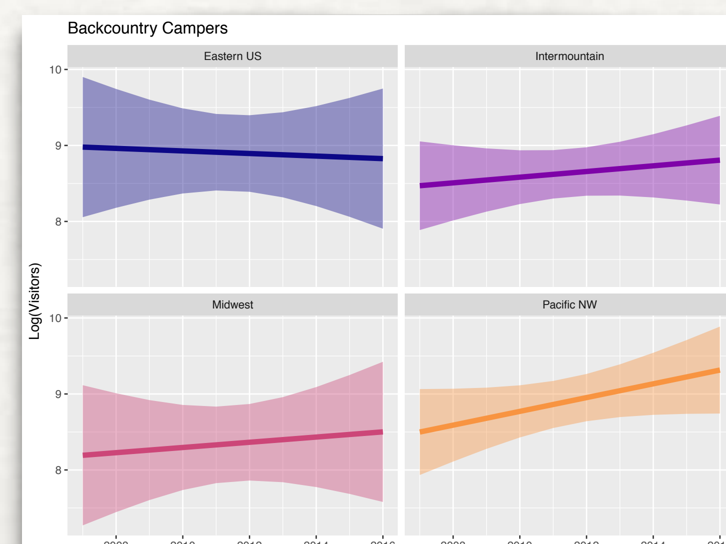
**AUDIENCE PARTICIPATION!**

Remember, `map()` functions return objects; in order to see our plots, we have to *print* them somehow. How can we do that?

```
walk2(.x = c("rec.pdf", "tent.pdf", "backcountry.pdf"),
      .y = nps_plots,
      ggsave,
      width = 8, height = 6)
```



# BONUS MATERIAL

You *may* have been thinking: “given my newfound knowledge of `pmap()`, couldn’t we have extracted that data in one step instead of copying & pasting *almost* the same thing?”

YES. Yes we could.

```
showoff <- pmap_df(
  .l = list(
    "year" = rep(2007:2016, 3),
    "table_prefix" = c(rep("recreation_visits", 10),
                       rep("tent_campers", 10),
                       rep("backcountry_campers", 10)),
    "url" = c(rep(url_recvisits, 10),
              rep(url_tent, 10),
              rep(url_back, 10))
  ),
  .f = download_nps_year
)
```

*(This creates one big data frame; we’d probably want to add a line to `download_nps_year()` to add the visitor type as well as the year.)*

# BUT WAIT! THERE'S MORE!

## RANDOM PURRR THINGS THAT YOU MIGHT LIKE

- `partial()`, for when you want to create a partially specified version of a function (eg, `q25 <- partial(quantile, probs = 0.25, na.rm = TRUE)`)
- `flatten()`, for removing hierarchies from a list
- `safely(), quietly(), possibly()` can be helpful especially when writing functions or packages
- `invoke(), modify()`
- List-columns can be your friend if you want to store complex data, results, etc in a tidy way; `purrr` functions can be really helpful when working with these. Jenny Bryan's tutorial is a great resource here.

# BUT WAIT! THERE'S MORE!

## RANDOM PURRR THINGS THAT YOU MIGHT LIKE

- `partial()`, for when you want to create a partially specified version of a function (eg, `q25 <- partial(quantile, probs = 0.25, na.rm = TRUE)`)
- `flatten()`, for removing hierarchies from a list
- `safely(), quietly(), possibly()` can be helpful especially when writing functions or packages
- `invoke(), modify()`
- List-columns can be your friend if you want to store complex data, results, etc in a tidy way; `purrr` functions can be really helpful when working with these. Jenny Bryan's tutorial is a great resource here.

```
> dplyr::starwars %>% select(name, height, hair_color, skin_color, films, vehicles)
# A tibble: 87 x 6
                 name height    hair_color  skin_color     films  vehicles
                <chr>  <int>         <chr>       <chr>    <list>    <list>
 1     Luke Skywalker    172         blond        fair <chr [5]> <chr [2]>
 2              C-3PO    167          <NA>        gold <chr [6]> <chr [0]>
 3              R2-D2     96          <NA> white, blue <chr [7]> <chr [0]>
 4        Darth Vader    202          none       white <chr [4]> <chr [0]>
 5        Leia Organa    150         brown       light <chr [5]> <chr [1]>
```

# PURRR RESOURCES

## FOR THE CURIOUS


cafepress.com

- Official page: purrr.tidyverse.org
- RStudio cheatsheet (under "Apply Functions")
- R for Data Science: Lists & iteration
- DataCamp: Writing Functions in R
- Charlotte Wickham's purrr tutorial
- Jenny Bryan's purrr tutorial; particularly great if you love the idea of list-columns
- Hadley Wickham on purrr vs *apply
- Fun use cases:
  - A roundup of blog posts curated by Mara Averick
  - Peter Kamerman on bootstrap CIs with purrr
  - Ken Butler on handling errors with safely/possibly

# THANK YOU & HAPPY PURRR-ING

*photo: Nick Strayer, of his cat Flumpert, via Lucy D'Agostino McGowan*
*bad cropping by me*