```
library(dplyr)

rladies_global %>%
  filter(city == 'Austin')
```

R FOR DATA SCIENCE: PROGRAMMING -- FUNCTIONS, VECTORS, AND ITERATION

Hello!

Welcome to R-Ladies

Thanks to our sponsors!

R Studio | Web.com

1. Introduction

R language, RStudio,
R4DS Workshop series

Three things
you'll need to install

1. **Install R** -- this is the open-source programming language we'll use (download via CRAN -- Comprehensive R Archive Network)
2. **Install RStudio** -- this is the most popular IDE for R and will make your life a lot easier (download from rstudio.com/download)
3. **Install the tidyverse** -- this is the group of packages we'll use within R to work with data. Install with one line of code in R:
   `install.packages("tidyverse")`

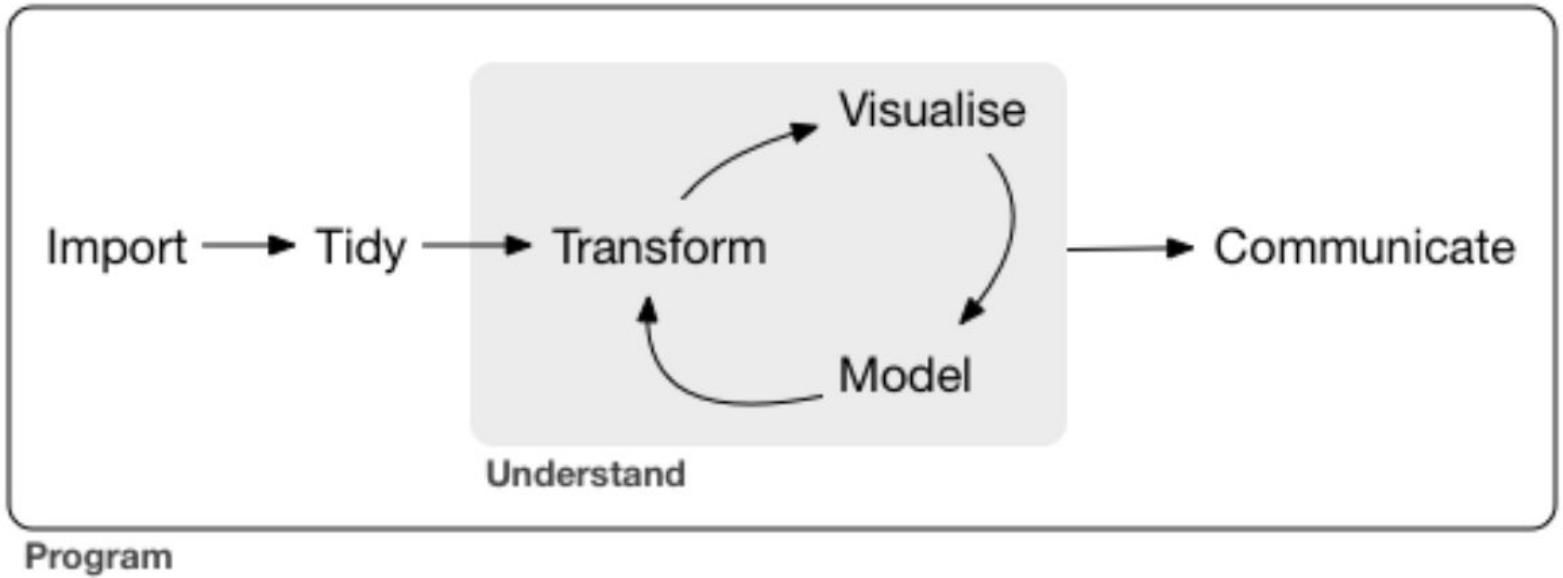
1b. Introduction

R for Data Science Workshop Series

R4DS
Workshop Series

- **Exploring Data with ggplot2 + dplyr** [DONE]
- **Exploratory Data Analysis and Workflow** [DONE]
- **Data Wrangling in the Tidyverse** [November 28]
- **Programming -- Functions, Vectors, and Iteration** [December 13]
- **Modeling with modelr, purrr, and broom** [January 24]
- **Communicating Results with rmarkdown and ggplot2** [February 21]

The data science process (tidied)



What is
the tidyverse?

- Collection of R packages based on tidy data principles
- Designed to work together
- An easier way to code!
- AKA “Hadleyverse” (most packages written by Hadley Wickham)

What is
the tidyverse?

What is
tidy data?

- Each variable is a column
- Each observation is a row
- Each type of observational unit is a table

id	artist	track	time
1	2 Pac	Baby Don't Cry	4:22
2	2Ge+her	The Hardest Part Of ...	3:15
3	3 Doors Down	Kryptonite	3:53
4	3 Doors Down	Loser	4:24
5	504 Boyz	Wobble Wobble	3:35
6	98^0	Give Me Just One Nig...	3:24
7	A*Teens	Dancing Queen	3:44
8	Aaliyah	I Don't Wanna	4:15
9	Aaliyah	Try Again	4:03
10	Adams, Yolanda	Open My Heart	5:30
11	Adkins, Trace	More	3:05
12	Aguilera, Christina	Come On Over Baby	3:38
13	Aguilera, Christina	I Turn To You	4:00
14	Aguilera, Christina	What A Girl Wants	3:18
15	Alice Deejay	Better Off Alone	6:50

2.
Pipes with magrittr

v

Magrittr!! (%>%)

`library(tidyverse)`

-->loads magrittr automatically!

→ Read it as “and then” or “followed by”

What’s the point?

→ helps you write code that is easier to read and understand

→ multiple steps in one go--less likely to make transcription errors

```
foo_foo_1 <- hop(foo_foo, through = forest)
foo_foo_2 <- scoop(foo_foo_1, up = field_mice)
foo_foo_3 <- bop(foo_foo_2, on = head)
```

VS.

```
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mouse) %>%
  bop(on = head)
```

When not to use the pipe

- If more than 10 steps
  - Break it up instead
- If you need to combine two or more objects

Other tools from magrittr

- Say you want to print one of the intermediate steps
  - Tee pipe! %T>%

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()
#>  NULL

rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
#>  num [1:50, 1:2] -0.387 -0.785 -1.057 -0.796 -1.756 ...
```

3. Functions

C

“Writing good functions is a lifetime journey.”

-R for Data Science

Functions:
general notes

- We're focusing on writing functions in base R, so you don't need any extra packages
- Write a function whenever you've copied and pasted a block of code more than twice
- It's easier to turn existing code into a function than to write one from scratch
- Use comments (lines starting with #) to explain the "why" of your code

Functions
vs copy/paste

Three advantages of functions over copy/paste:

+ Simply by naming a function, you can make your code easier to understand
+ As requirements change, you only need to update code in one place (instead of many)
+ Reduces the room for errors

Steps for
building functions

1. Choose a name (this is harder than it sounds!)
2. List inputs (*arguments*) to the function inside the function parentheses
3. Put code that “does something” in the body of the function (inside the curly brackets)

Functions:
naming

- Names should be short and clear (but keep in mind that RStudio has autocomplete if you need it!)
- Functions should be named verbs; arguments should be nouns
- Be consistent with formatting when you have multiple words -- like `snake_case` or `camelCase`

Functions:
naming, cont.

+ If you create a family of functions that do similar things, have consistent names and arguments
+ Avoid overriding existing functions and variables
+ Remember you can always rename functions

Functions:
conditional execution

An if statement lets you conditionally execute code:

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

Functions:
multiple conditions

You can chain multiple if statements together:

```
if (this) {
    # do that
} else if (that) {
    # do something else
} else {
    # do a final option
}
```

Functions:
arguments

Arguments to a function generally fall into two categories:

1. Supplies data to compute on
2. Supplies arguments that control the details of computation

Generally, data args should come first, followed by detail arguments, which should have default values

Functions:
arguments, cont.

- Defaults should almost always be the most common value
  - Exception: don't default `na.rm = TRUE` ; it's better to be informed of missings than mask them
- Typically you can omit the names of data arguments
- If you want to override a default value, you should use the full name

Functions:
arguments, cont.

Some common short argument names:

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

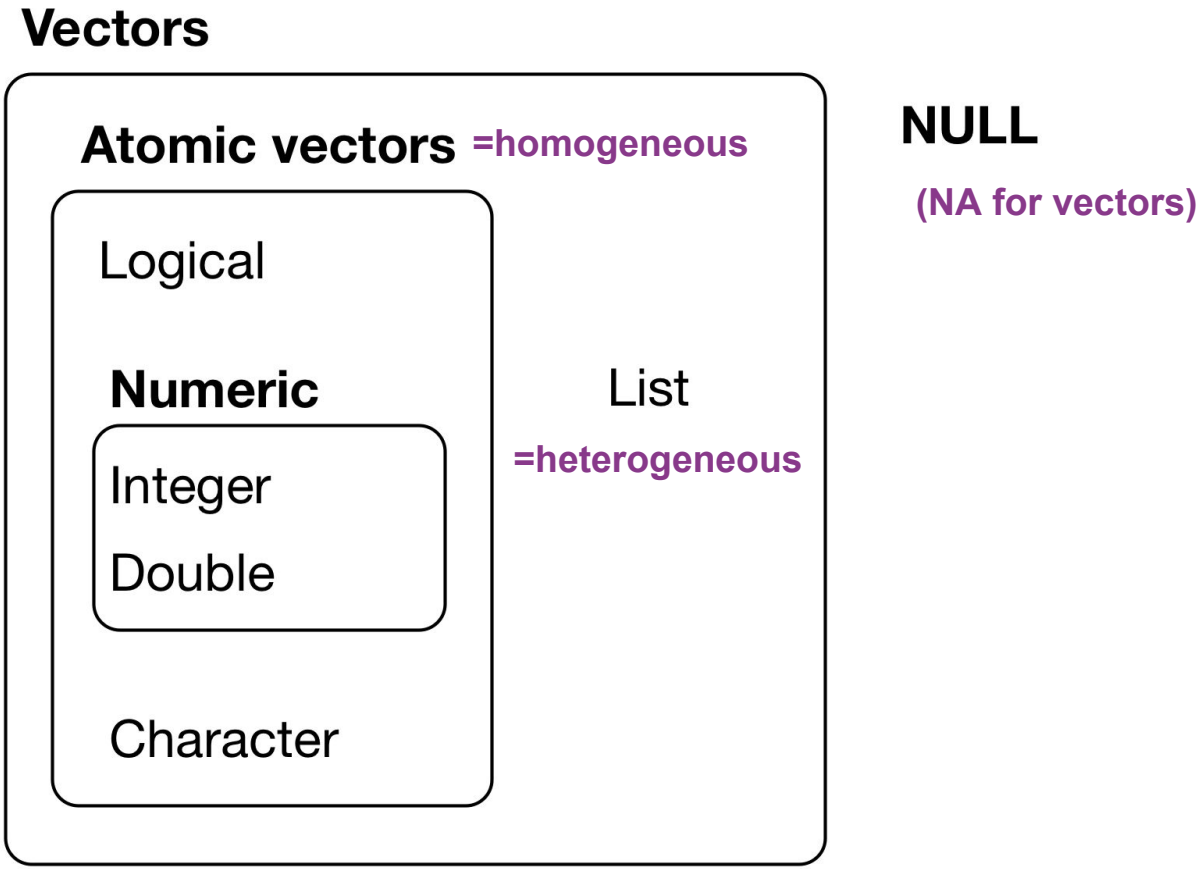
Functions:
odds and ends

- You can break your file into easily readable chunks using long lines of -
  ```
  # example----------------------------------
  ```
- It's best to place spaces around operators and after commas (for readability)
- Lazy evaluation: arguments are lazily evaluated (not computed until they're needed)

4. Vectors

v

Vectors!


Vectors
Atomic vectors
=homogeneous
Logical
Numeric
Integer
Double
Character
List
=heterogeneous
NULL
(NA for vectors)

vectors

1. Its **type**, which you can determine with `typeof()`.

```
typeof(letters)
#> [1] "character"
typeof(1:10)
#> [1] "integer"
```

2. Its **length**, which you can determine with `length()`.

```
x <- list("a", "b", 1:10)
length(x)
#> [1] 3
```

R studio also tells you!

Atomic Vector Types

1. **Logical--most simple (`TRUE`, `FALSE`, `NA`)**
2. **Numeric--2 types (double, integer)**
3. **Character--most complex**
4. (raw)
5. (complex)

Numeric-double vs. integer

1. Doubles are approximations (think pi)
2. Integers can be `NA`, but doubles can be `NaN, Inf, NA, -Inf`

Avoid using `==` to check for these other special values. Instead use the helper functions `is.finite()`, `is.infinite()`, and `is.nan()`:

	0	Inf	NA	NaN
`is.finite()`	x			
`is.infinite()`		x		
`is.na()`			x	x
`is.nan()`				x

Character

```
NA            # logical
#> [1] NA
NA_integer_   # integer
#> [1] NA
NA_real_      # double
#> [1] NA
NA_character_ # character
#> [1] NA
```

Rarely need to specify type.

Using atomic vectors-coercion (convert)

- Explicit `as.logical(), as.integer()`
- Implicit
  - Use a vector in a context that expects a certain type
  - Example: sum of a logical vector

```r
x <- sample(20, 100, replace = TRUE)
y <- x > 10
sum(y)  # how many are greater than 10?
#> [1] 44
mean(y) # what proportion are greater than 10?
#> [1] 0.44
```

Using atomic vectors-Test functions

	lgl	int	dbl	chr	list
`is_logical()`	X				
`is_integer()`		X			
`is_double()`			X		
`is_numeric()`		X	X		
`is_character()`				X	
`is_atomic()`	X	X	X	X	
`is_list()`					X
`is_vector()`	X	X	X	X	X

Using atomic vectors-naming and subsetting

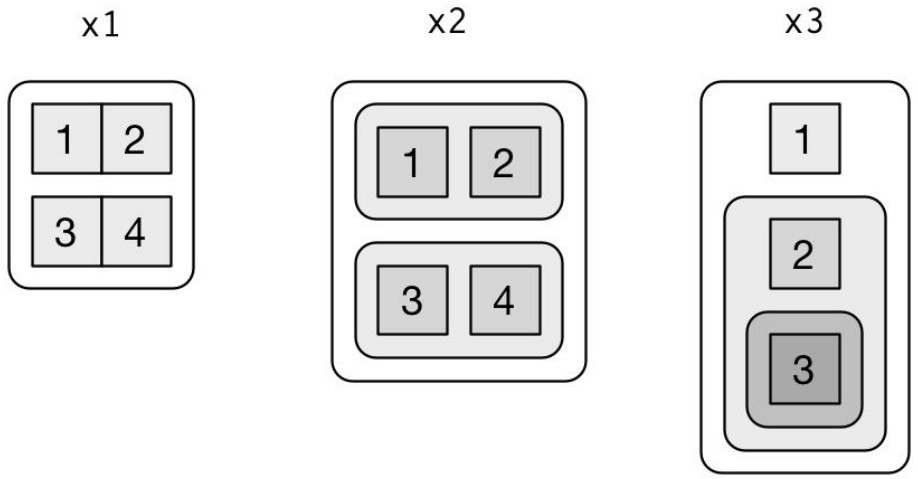
- Name the items within a vector
- Naming is useful for subsetting
  - A way to `filter()` vectors!
  - Power of the `[ ]` bracket!
  - Check it out in R script

Recursive vectors (Lists)

- More complex than atomic vectors
  - Can have lists of lists!
- Use `str()` to check out the contents of a list
  - Can be a mix!

Recursive vectors (lists)

```r
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```

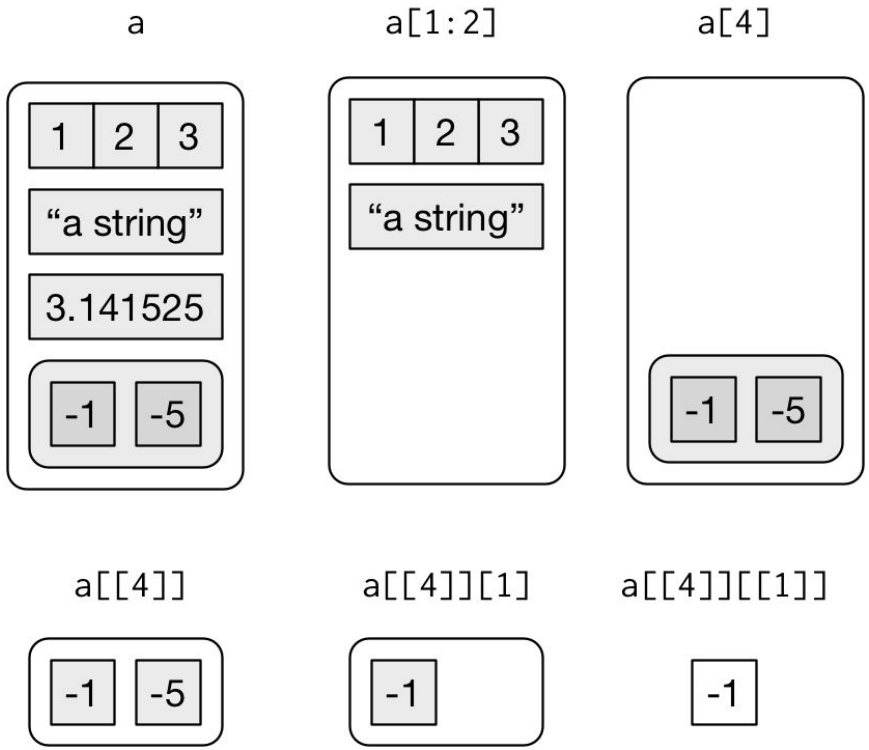


Nests of lists!

Subsetting lists

Three ways to subset lists

1. `[`
2. `[[`
3. `$` --similar to [[but do not need to use quotes

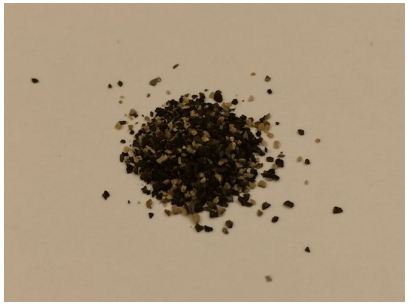


List of condiments

`x`

`x [1]`

`x [[1]]`



`x [[1]][[1]]`

Attributes

Three important attributes

1. **Names**--we already covered this, filtering!
2. **Dimensions**--not covered here
3. **Class**--controls how generic functions work
    a. Key to object oriented programming--functions behave differently based on class
        i. Dive in if you want to, we will skip!

Augmented vectors

Atomic vectors a little something extra (have a class)

Four types

1. Factors
2. Dates
3. Date-times
4. Tibbles

Augmented vectors

Augmented vector	Notes
Factors	For categorical data. Built on integers, has a **levels attribute**
Dates	Built on numeric, **date attribute** which is number of days since 1 Jan 1970
Date-times	Built on numeric, **class `POSIXct`,** n seconds since 1 January 1970. `tzone` attribute is optional.
Tibbles	Augmented lists. Class `tbl_df + tbl + data.frame`. Has **names and row.names attributes.**

5.
Iteration with purrr

C

Iteration:
introduction

- Iteration is a tool for reducing duplication (like functions)
- Used when you need to do the same thing to multiple inputs
  - Ex: repeat same operation on different columns or datasets
- Based on (and replaces) for loops and while loops

Iteration:
for loops

Imagine we want to take the median of each column in a dataset.

This is redundant:

```
median(df$a)
median(df$b)
median(df$c)
```

This is better:

```
output <- vector("double", ncol(df))
For (i in seq_along(df)) {
Output[[i]] <- median(df[[i]])
}
```

Iteration:
for loops

Every for loop has three components:

1. `output <- vector("double", length(x))`
   An output -- this is often initialized as a vector of given length;
   grows at each iteration

2. `i in seq_along(df)`
   A sequence -- determines what to loop over (like `1:length(x)`)

3. `output[[i]] <- median(df[[i]])`
   A body -- the code that does the work, which is run repeatedly, each time with a different value for i

Iteration:
for loop variations

Four variations on the basic theme of the for loop:

+ Modifying an existing object instead of creating a new object
+ Looping over names or values (instead of indices)
+ Handling outputs of unknown length
+ Handling sequences of unknown length

Iteration:
modifying an existing object

Example: We want to replace an existing dataframe with each column's median

To solve, think of the three components:

+ Output: we already have this -- same as the input!
+ Sequence: iterate over each column to seq_along(df)
+ Body: apply the function

Result:

```
for (i in seq_along(df)) {
  df[[i]] <- median(df[[i]])
}
```

Iteration:
looping patterns

Three basic ways to loop over a vector:

+ Looping over the numeric indices | `for (i in seq_along(xs))` with `x[[i]]`
  (This is what we've been doing so far)

+ Looping over the elements | `for (x in xs)`
  This is useful if you only care about side effects like plotting or saving a file because it's difficult to save output efficiently

+ Looping over the names | `for (nm in names(xs))`
  This gives you a name that you can use to access value with `x[[nm]]` ; useful if you want to use the name in plot title or filename

Iteration:
unknown output length

If you don't know output length, you could…

+ Progressively grow vector: BUT DON'T! With each iteration, R has to copy all data from previous iterations
+ Save results in a list and combine into vector when loop is done

```
out <- vector("list", length(df))
for (i in seq_ along(df)) {
out[[i]] <- median[[i]])
}
str(unlist(out))
```

Iteration:
unknown sequence length

When sequence length is unknown (like in simulations), use a while loop.

Ex: How many flips of a coin to get a single head?

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

flips <- 0;
nheads <- 0

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

Iteration:
for loops vs. functionals

Functionals are wrapped up versions of for loops

+ More efficient than for loops (less duplication)

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}    # this will do mean, median, sd for each col
```

Iteration with purrr:
the map functions

One map function in purrr for each type of output when looping:

- `map()` makes a list
- `map_lgl()` makes a logical vector
- `map_int()` makes an integer vector
- `map_dbl()` makes a double vector
- `map_chr()` makes a character vector

Each function takes a vector as input and a function to apply to each piece, and then returns a new vector that is the same length as the input

Iteration with purrr:
map function examples

```
df %>% map_dbl(mean)
#>       a       b       c       d
#>  0.2026 -0.2068  0.1275 -0.0917
df %>% map_dbl(median)
#>      a      b      c      d
#>  0.237 -0.218  0.254 -0.133
df %>% map_dbl(sd)
#>     a     b     c     d
#> 0.796 0.759 1.164 1.062
```

Iteration with purrr:
the map functions

One map function in purrr for each type of output when looping:

- `map()` makes a list
- `map_lgl()` makes a logical vector
- `map_int()` makes an integer vector
- `map_dbl()` makes a double vector
- `map_chr()` makes a character vector

Each function takes a vector as input, applies a function to each piece, and then returns a new vector that is the same length as the input

Iteration with purrr:
vs. apply family of functions

Similarities between purrr and apply:

- `lapply()` is almost identical to map
- `sapply()` is a wrapper around `lapply()` that simplifies the output -- but problematic in a function b/c you don't know output
- `vapply()` is safe alternative to `sapply()` b/c you supply type -- but can produce matrices (and purrr functions can only produce vectors)

Iteration with purrr:
dealing with failure

What if your function fails on at least one input?

- `safely()` won't throw an error and returns two elements -- result and error (look for NAs); always succeeds
- `possibly()` always succeees, but you can tell it which default value to return if there is an error
- `quietly()` like `safely()`, but instead of capturing errors, it captures printed output, messages, and warnings

R-Ladies Austin

Upcoming Events

R4DS: Modeling with modelr, purrr, and broom [January 24]

Book Club: Weapons of Math Destruction [January 31]

R4DS: Communicating Results with rmarkdown and ggplot2 [February 21]