```
library(dplyr)

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

R FOR DATA SCIENCE: DATA WRANGLING IN THE TIDYVERSE

Hello!

Welcome to R-Ladies

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")`

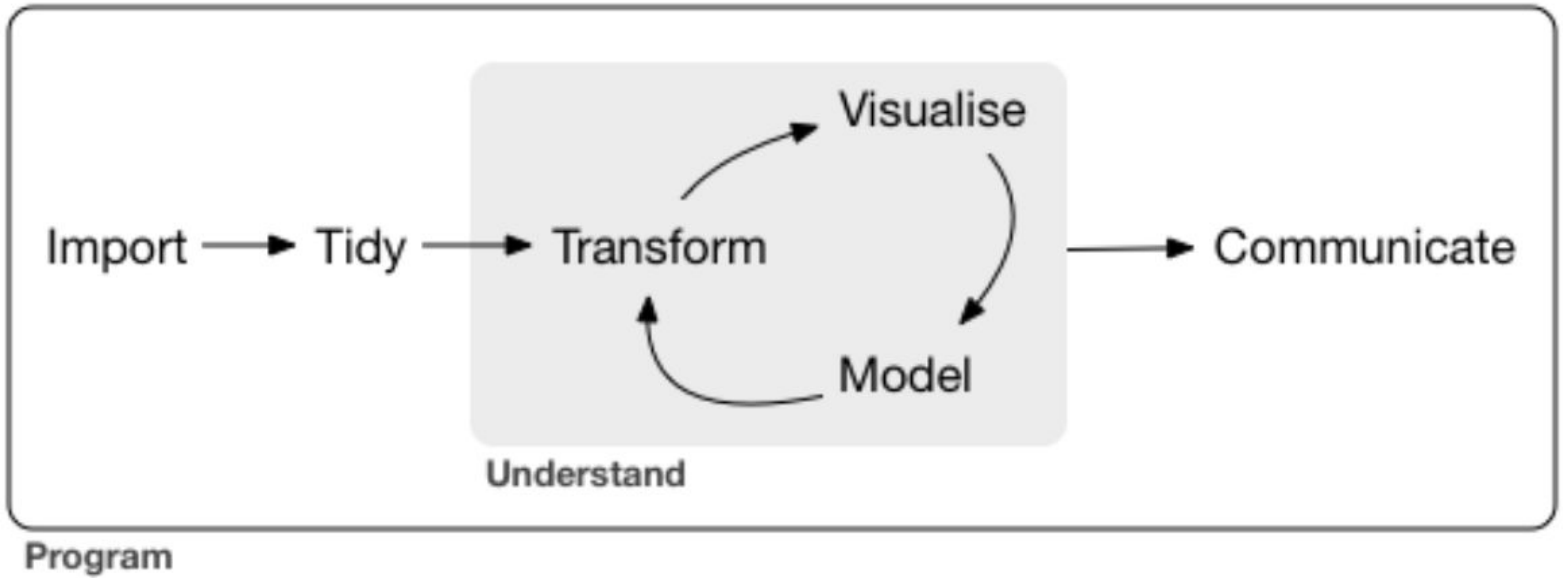
1b. Introduction

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.
Tibbles with tibble

C

What is a tibble?

A tibble is an opinionated data frame that makes the tidyverse work a little easier.

+ Coerce a data frame to a tibble using `as_tibble()`
+ Create a new tibble from vectors with `tibble()`

Tibbles are opinionated

Tibbles are “opinionated” because they never:

+ Change the type of the inputs (like turning strings into factors!)
+ Change the the name of variables
+ Create row names

Additionally, tibbles can have nonsyntactic names

+ `` `:)` `` and `` `spaced names` `` are okay!

tibbles vs. data.frame

Two main differences:

- You can `print()` the data frame and control the number of rows (n) and width of display
  - Can also hardcode print options like min and max
- Tibbles are more strict than data.frames on selecting columns -- they never do partial matching and will generate a warning of column you want doesn't exist
  - Need to use special placeholder .

```
df %>% .$x
```

3.
Data Import with readr

v

read_csv()
syntax

You can read in 75% of csv files with only this slide!

- `read_csv()` read in comma-delimited files
- `read_csv2()` read in semicolon separated files
  - Plus others
- `read_csv("filename.csv", skip= , comment= , col_names= , na = )`
  - Skip--skip rows that aren't data at the top
  - Comment--drop lines that start with a common pattern
  - col_names= boolean or vector of names
  - Na --is there a symbol that indicates NA?

read_csv() vs read.csv()

Use `read_csv()` and readr!

- Faster
- No need for `stringsasFactors=FALSE`
- No munging of col names

Concept of parsing

`parse_*(c(VectorToParse), na="")`

- E.g. `parse_integer(c("1", "231","abc", "145.35"))`
- If fails, show problems with `problems()`

```
> library(readr)
> parse_integer(c("1","231","abc","145.23"))
Warning: 2 parsing failures.
row # A tibble: 2 x 4 col     row   col               expected actual expected   <int> <int>
    3    NA             an integer    abc row 2     4    NA no trailing characters    .23

[1]   1 231  NA  NA
attr(,"problems")
# A tibble: 2 x 4
    row   col               expected actual
  <int> <int>                  <chr>  <chr>
1     3    NA             an integer    abc
2     4    NA no trailing characters    .23
Warning message:
In rbind(names(probs), probs_f) :
  number of columns of result is not a multiple of vector length (arg 1)
> |
```

Parsing

`parse_character()`

- Characters are stored as hexadecimals (encoded)
  - ASCII
- UTF-8--better, more inclusive
  - Used as default in readr
  - `locale=locale(encoding="Latin1")`
    - Use this to change
  - `guess_encoding()`

Parsing

`parse_factor()`

- Fewer issues
- May be best to leave as character if many unexpected values

`parse_number()`

- Ignores characters before ($) and after (%)
- Ignore grouping mark (“.”, “,”,”’”)

Parsing

`parse_datetime()`

`parse_date()`

- Expects ISO8601 (biggest to smallest)
- YYYY-MM-DD HH:MM:SS
- Can supply your own formats if different
- E.g. %I, %b
- challenging!

How readr parses a file

- Reads 1000 rows and uses heuristic to find best fit
- `guess_parser()`
- Strings if nothing found

Potential problems

Many missing values to start

First 1000 rows a special case

problems(DataframeName)

- Copy and paste the column specs into original call
- See example in R

Writing a file

```
write_csv(dataframe, “filename.csv”)
```

- Always back to UTF-8
- Dates and time in ISO8601

Other types of data

haven package → Stata and SAS files

Others:

readxl

jsonlite

RMySQL

4.
Tidy Data with tidyr

C

Tidy data: Review

1. Each variable must have its own column
2. Each observation must have its own row.
3. Each value must have its own cell.

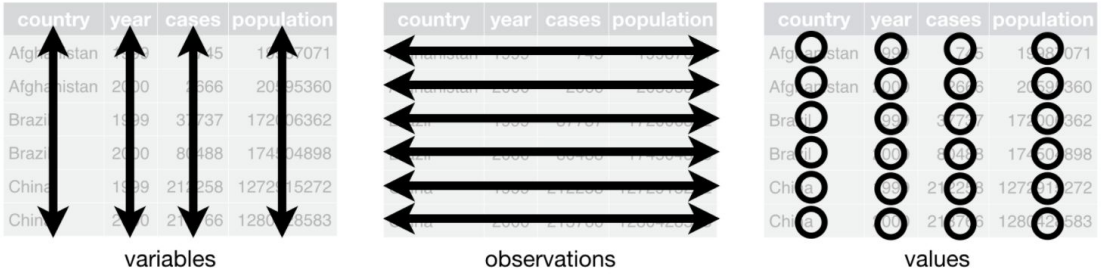


Figure 12.1: Following three rules makes a dataset tidy: variables are in columns, observations are in rows, and values are in cells.

Tidyr:
`gather()`

Problem: some column names are values of a variable

Solution: *gather* those columns into a new pair of variables, which requires:
+ The set of columns that represent values instead of variables
+ The name of the variable whose values form the column names (the *key*)
+ The name of the variables whose values are spread over cells (the *value*)

```
gather(data, key, value)
```

Tidyr:
`spread()`

Problem: observations are scattered across multiple rows

Solution: *spread* those values into new columns, which requires:
+ The column that containes variable names (the *key*)
+ The columns that contains values forms multiple variables (the *value* column)

```
spread(data, key, value)
```

Tidyr:
`separate()`

Problem: one column contains two variables

Solution: *separate* those values into new columns, which requires:
+ The name of the column to separate
+ The names of the columns to separate into

By default, `separate()` will split values wherever it sees a non-alphanumeric character -- you can override this by passing a character to the “sep” argument

```
separate(data, vol_to_split, into = c(“col1”, “col2”), sep = “/”)
```

Tidyr:
`unite()`

Problem: a single variable is contained in multiple columns

Solution: *unite* those values into a single column, which requires:
+ The names of the columns to combine

By default, tidyr will place an underscore between the values from different columns -- if we don't want a separator, we can use sep = ""

```
unite(data, century, year, sep = "")
```

Tidyr:
Missing Values

“An explicit missing value is the presence of an absence;
An implicit missing value is the absence of a presence.”

Explicit: contains NA (ex: A, B, NA, D)
Implicit: does not occur in dataset (ex: A, B, D)

`na.rm = TRUE` argument ignores missing values while performing calculations

Tidyr:
Missing Values

`complete()`

Takes a set of columns and finds all unique combinations, then ensures the original dataset contains all those values (fills NAs when needed)

`fill()`

Takes a set of columns where you want missing values to be replaced by the most recent nonmissing value (“last observation carried forward”)

5.
Relational Data with dplyr

C

Relational Data: Intro

+ Most data analysis relies on data from multiple tables.

+ To analyze multiple tables, you must combine them.

+ Data in multiple tables is called relational data because it's the relations (not just the individual datasets) that are important.

+ Relations are always defined between a pair of tables -- and dplyr has some families of verbs designed to connect these tables.

Relational Data:
dplyr relational verbs

+ **Mutating joins:** add new variables to one data frame from matching observations in another

+ **Filtering joins:** filter observations from one data frame based on whether or not they match an observation in the other table

+ **Set operations:** treat observations as if they were set elements

Relational Data: Keys

Keys: variables used to connect each pair of tables (unique identifiers)

Primary key: uniquely identifies an observation in its own table

Foreign key: uniquely identifies an observation in another table

Variables can be both primary and foreign keys!

Mutating Joins

Allows you to combine variables from two tables, and adds new variables to the right of the newly created data frame

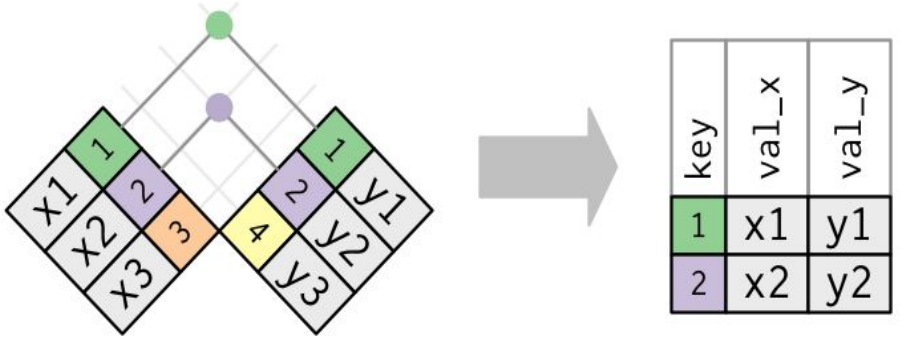
+ Inner join and the three outer joins are all mutating joins
+ Number of matches = number of new rows

x

1	x1
2	x2
3	x3

y

1	y1
2	y2
4	y3

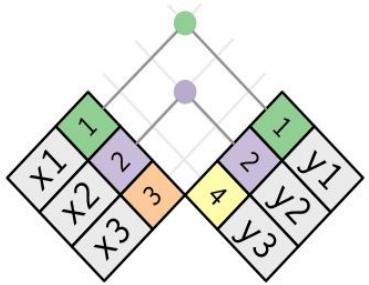


Inner Join

Matches pairs of observations whenever their keys are equal.

+ Output is a new data frame that contains the key, x values, and corresponding y values.
+ Keeps only observations that appear in both tables
+ We use ‘by’ to specify the key:

```
inner_join(x, y, by = “key”)
```



key	val_x	val_y
1	x1	y1
2	x2	y2

Outer Joins

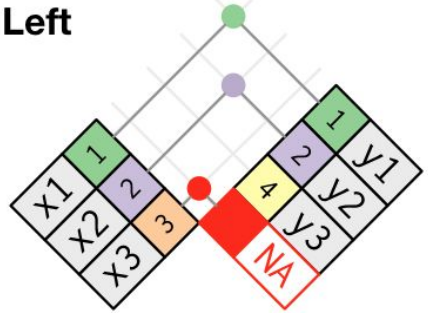
Keeps observations that appear in at least one of the tables. Three types:

+ `left_join` keeps all observations in x
+ `right_join` keeps all observations in y
+ `full_join` keels all observations in x and y

These joins add additional observations to each table, and fills values with NA when there are no values present for a given observation

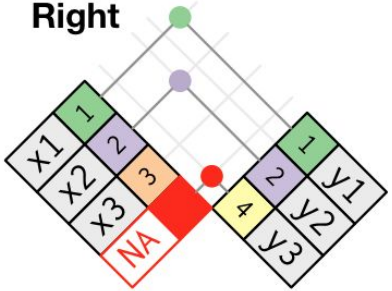
Outer Joins

Left



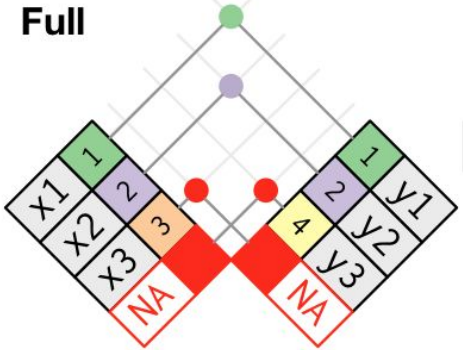
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

Right



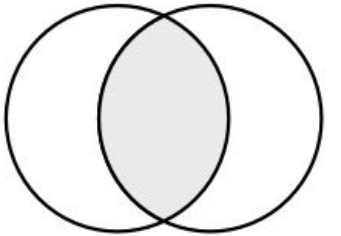
key	val_x	val_y
1	x1	y1
2	x2	y2
4	NA	y3

Full

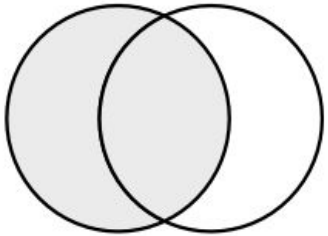


key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

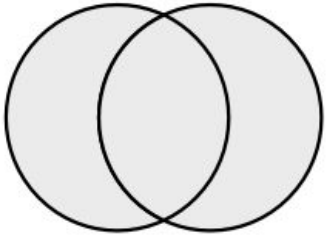
Mutating Joins



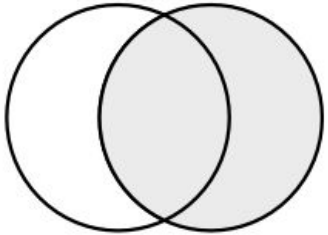
inner_join(x, y)



left_join(x, y)



full_join(x, y)



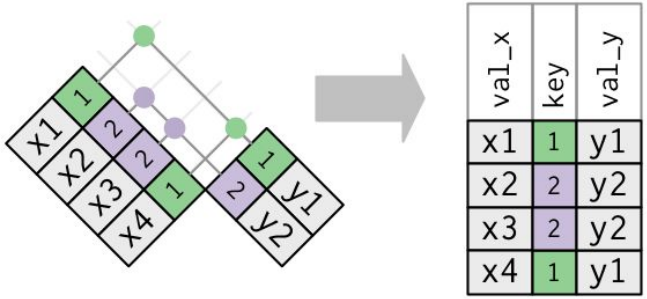
right_join(x, y)

Duplicate Keys

Two possibilities when keys are not unique:

One table has duplicate keys:

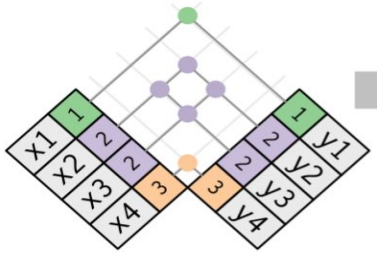
+ Useful when you want to add in additional info as there's a one-to-many relationship



val_x	key	val_y
x1	1	y1
x2	2	y2
x3	2	y2
x4	1	y1

Both tables have duplicate keys:

+ When joined you get all possible combinations; beware of errors (not "unique")



key	val_x	val_y
1	x1	y1
2	x2	y2
2	x2	y3
2	x3	y2
2	x3	y3
3	x4	y4

Specifying Key Columns

How do we define which variables to use to join tables?

+ Default: uses all variables that appear in both tables
+ A character vector: uses only some common variables to join

```
by = "x"
```

+ A named character vector: used when common variables don't share the same name

```
by = c("a" = "b")
```

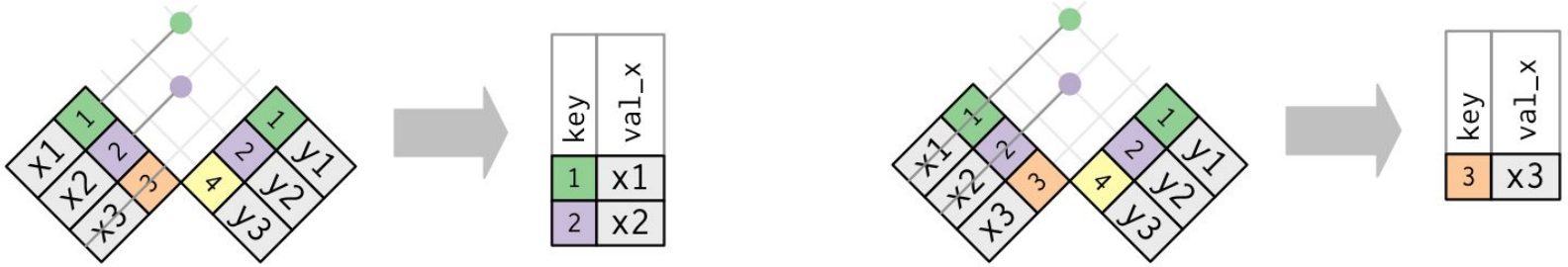
dplyr vs. SQL

dplyr	SQL
`inner_join(x, y, by = "z")`	`SELECT * FROM x INNER JOIN y USING (z)`
`left_join(x, y, by = "z")`	`SELECT * FROM x LEFT OUTER JOIN y USING (z)`
`right_join(x, y, by = "z")`	`SELECT * FROM x RIGHT OUTER JOIN y USING (z)`
`full_join(x, y, by = "z")`	`SELECT * FROM x FULL OUTER JOIN y USING (z)`

Filtering Joins

Match observations in the same way as mutating joins, but impact the observations (not the variables). Two types:

+ `semi_join(x, y)` keeps all observations in x that have a match in y
+ `anti_join(x, y)` drops all observations in x that have a match in y



Set Operations

These work with a complete row, comparing the values of every variable -- they expect x and y to have the same variables and treat the observations like sets:

+ `intersect(x, y)`: returns only observations in both x and y
+ `union(x, y)`: returns unique observations in x and y
+ `setdiff(x, y)`: returns observations in x but not in y

6.
Strings with stringr

v

String basics

Difficultttttt!

You can spend hours doing his stuff

Must load explicitly! Not in tidyverse!

```
library(stringr)
```

- Special characters mean things
- Escape these to use them in a string!

Metacharacter	Literal Meaning	Escape Syntax
.	period or dot	\\.
$	dollar sign	\\$
*	asterisk	*
+	plus sign	\\+
?	question mark	\\?
|	vertical bar	\\\|
\\	double backslash	\\\\
^	caret	\\^
[	square bracket	\\[
{	curly brace	\\{
(	parenthesis	\\(

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Functions with strings

`str_*()`

- `str_c()` → combine strings
- `str_length` → length of string
- `str_sub(object, start, end)` → extract parts of string
- `str_to_upper()` → upper case
- `str_to_lower()` → lower case

Regular expressions

Match patterns--very useful for coding, searching, etc

Basic Matches

Anchors

Character classes and Alternatives

Repetition

Grouping and backreferences

- Each has characters to match these patterns

Regular Expressions -

Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

MATCH CHARACTERS

see <- function(rx) str_view_all("abc ABC 123\t.!?\\(){}\n", rx)

string (type this)	regexp (to mean this)	matches (which matches this)	example	
	a (etc.)	a (etc.)	see("a")	abc ABC 123 .!?\(){}
\\.	**\.**	.	see("\\.")	abc ABC 123 .!?\(){}
\\!	**\!**	!	see("\\!")	abc ABC 123 .!?\(){}
\\?	**\?**	?	see("\\?")	abc ABC 123 .!?\(){}
\\\\	****	\	see("\\\\")	abc ABC 123 .!?\(){}
\\(	**\(**	(	see("\\(")	abc ABC 123 .!?\(){}
\\)	**\)**	)	see("\\)")	abc ABC 123 .!?\(){}
\\{	**\{**	{	see("\\{")	abc ABC 123 .!?\(){}
\\}	**\}**	}	see("\\}")	abc ABC 123 .!?\(){}
\\n	**\n**	new line (return)	see("\\n")	abc ABC 123 .!?\(){}
\\t	**\t**	tab	see("\\t")	abc ABC 123 .!?\(){}
\\s	**\s**	any whitespace (**\S** *for non-whitespaces*)	see("\\s")	abc ABC 123 .!?\(){}
\\d	**\d**	any digit (**\D** *for non-digits*)	see("\\d")	abc ABC 123 .!?\(){}
\\w	**\w**	any word character (**\W** *for non-word chars*)	see("\\w")	abc ABC 123 .!?\(){}
\\b	**\b**	word boundaries	see("\\b")	abc ABC 123 .!?\(){}
	[:digit:] [1]	digits	see("[:digit:]")	abc ABC 123 .!?\(){}
	[:alpha:] [1]	letters	see("[:alpha:]")	abc ABC 123 .!?\(){}
	[:lower:] [1]	lowercase letters	see("[:lower:]")	abc ABC 123 .!?\(){}
	[:upper:] [1]	uppercase letters	see("[:upper:]")	abc ABC 123 .!?\(){}
	[:alnum:] [1]	letters and numbers	see("[:alnum:]")	abc ABC 123 .!?\(){}
	[:punct:] [1]	punctuation	see("[:punct:]")	abc ABC 123 .!?\(){}
	[:graph:] [1]	letters, numbers, and punctuation	see("[:graph:]")	abc ABC 123 .!?\(){}
	[:space:] [1]	space characters (i.e. \s)	see("[:space:]")	abc ABC 123 .!?\(){}
	[:blank:] [1]	space and tab (but not new line)	see("[:blank:]")	abc ABC 123 .!?\(){}
	.	every character except a new line	see(".")	abc ABC 123 .!?\(){}

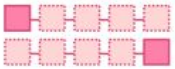
[1] Many base R functions require classes to be wrapped in a second set of [ ], e.g. **[[:digit:]]**

ALTERNATES

```
alt <- function(rx) str_view_all("abcde", rx)
```

regexp	matches	example	
ab|d	or	alt("ab|d")	abcde
[abe]	one of	alt("[abe]")	abcde
[^abe]	anything but	alt("[^abe]")	abcde
[a-c]	range	alt("[a-c]")	abcde

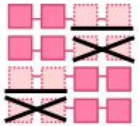
ANCHORS



```
anchor <- function(rx) str_view_all("aaa", rx)
```

regexp	matches	example	
^a	start of string	anchor("^a")	aaa
a$	end of string	anchor("a$")	aaa

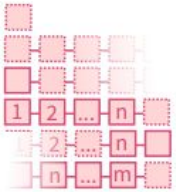
LOOK AROUNDS



```
look <- function(rx) str_view_all("bacad", rx)
```

regexp	matches	example	
a(?=c)	followed by	look("a(?=c)")	bacad
a(?!c)	not followed by	look("a(?!c)")	bacad
(?<=b)a	preceded by	look("(?<=b)a")	bacad
(?<!b)a	not preceded by	look("(?<!b)a")	bacad

QUANTIFIERS




```
quant <- function(rx) str_view_all(".a.aa.aaa", rx)
```

regexp	matches	example	
a?	zero or one	quant("a?")	.a.aa.aaa
a*	zero or more	quant("a*")	.a.aa.aaa
a+	one or more	quant("a+")	.a.aa.aaa
a{n}	exactly **n**	quant("a{2}")	.a.aa.aaa
a{n,}	**n** or more	quant("a{2,}")	.a.aa.aaa
a{n, m}	between **n** and **m**	quant("a{2,4}")	.a.aa.aaa

GROUPS

```
ref <- function(rx) str_view_all("abbaab", rx)
```

Use parentheses to set precedent (order of evaluation) and create groups

regexp	matches	example	
(ab|d)e	sets precedence	alt("(ab|d)e")	abcde

Use an escaped number to refer to and duplicate parentheses groups that occur earlier in a pattern. Refer to each group by its order of appearance

string (type this)	regexp (to mean this)	matches (which matches this)	example (the result is the same as ref("abba"))	
\\1	\1 (etc.)	first () group, etc.	ref("(a)(b)\\2\\1")	abbaab

You have a pattern--now what?

`str_detect()`

- Returns logical vector
- I use this the most--does a string match a pattern?
- E.g. does a string have the word “love”?

`str_count()`

- Tells you how many matches

`str_extract()`

- Returns and extracts only first match
- `str_extract_all()` returns a vector of all them

You have a pattern--now what?

`str_replace()`

- Replace a pattern in a string with a different string

`str_replace_all()`

- Replace multiple different patterns

`str_split()`

- Split up into pieces
- Returns a list, can request a matrix in return with `simplify=TRUE`

7.
Factors with forcats

C

Factors in R

Factors are used to work with categorical variables

+ Represent a fixed and known set of possible values
+ Many functions in base R automatically convert characters to factors

```
stringsAsFactors = TRUE
```

When are factors > strings?

```
month <- c("Dec", "Apr", "Jan")
```

Using a string has two problems:

1. Nothing saves you from typos (but only 12 possible months)
2. Doesn't sort in a useful way (alphabetically doesn't make sense here)

Solve both problems with a factor!

Creating Factors

```
month <- c("Dec", "Apr", "Jan")
```

Assign levels & create factor:

```
month_levels <- c("Jan", "Apr", "Dec")
month_factor <- factor(month, levels = month_levels)
```

Any values not in set (levels) will be converted to NA

`"Jam"` becomes `NA`

Working w/ Factors

Use `readr::parse_factor(x, levels = factor_levels)` to see errors

+ If you omit levels, they'll be taken from the data in alphabetical order

Use unique(x) to set levels when creating factor for order of levels to match the order of the first appearance in data:

```
factor(x, levels = unique(x))
```

Use levels() to access set of valid levels directly:

```
levels(x)
```

Working w/ Factors

You can see factor levels with count:

```
dataframe %>% count(factor_name)
```

Or with a bar chart:

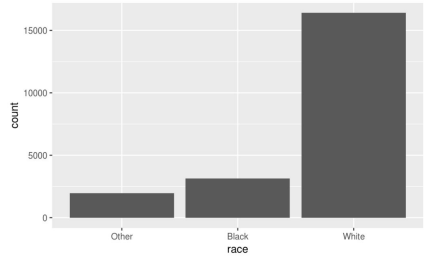
```
ggplot(dataframe, aes(factor_name)) +
geom_bar()
```

Note: ggplot2 will drop levels that don't have values by default -- can force them to display with:

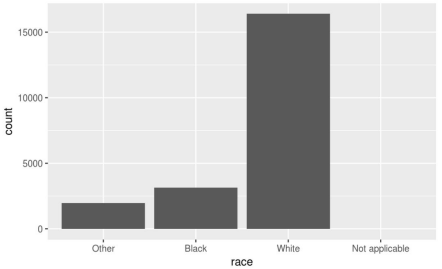
```
ggplot(dataframe, aes(factor_name)) +
geom_bar() + scale_x_discrete(drop = FALSE)
```

```
gss_cat %>%
  count(race)
#> # A tibble: 3 × 2
#>     race     n
#>   <fctr> <int>
#> 1  Other  1959
#> 2  Black  3129
#> 3  White 16395
```

```
ggplot(gss_cat, aes(race)) +
  geom_bar()
```




```
ggplot(gss_cat, aes(race)) +
  geom_bar() +
  scale_x_discrete(drop = FALSE)
```



Modifying Factor Order

Use `fct_reorder()` to reorder factor levels. This takes three arguments:

+ `f`, the factor whose level you want to modify
+ `x`, a numeric vector that you want to use to reorder the levels
+ Optionally, `fun`, a function that's used if there are multiple values of `x` for each value of `f` -- the default value is median

```
fct_reorder(f = factor, x = reorder_by_this)
```

Other Factor Order Modifications

Use `fct_reorder2()` reorders factor by the y-values associated with the largest x-values (and makes a plot easier read)

Use `fct_infreq()` for bar charts to order levels in increasing frequency

Modifying Factor Levels

Use `fct_recode()` to recode or change the value of each level

+ Will leave levels that aren't explicitly mentioned as-is
+ Will warn you if you accidentally refer to a level that doesn't exist
+ Allows you to combine groups by assigning multiple old levels to the same new level

```
mutate(partyid = fct_recode(partyid,
  "Republican, strong"    = "Strong republican",
  "Republican, weak"      = "Not str republican",
  "Independent, near rep" = "Ind,near rep",
  "Independent, near dem" = "Ind,near dem",
  "Democrat, weak"        = "Not str democrat",
  "Democrat, strong"      = "Strong democrat"
)) %>%
```

Modifying Factor Levels

Use `fct_collapse()` if you want to collapse a lot of levels

+ For each new variable, you can provide a vector of old levels that you'd like to collapse to more inclusive levels

```
mutate(partyid = fct_collapse(partyid,
  other = c("No answer", "Don't know", "Other party"),
  rep = c("Strong republican", "Not str republican"),
  ind = c("Ind,near rep", "Independent", "Ind,near dem"),
  dem = c("Not str democrat", "Strong democrat")
)) %>%
```

Use `fct_lump()` if you want to lump together all of the small groups to make a plot or table simpler.

+ Default: progressively lumps together the smallest groups -- can use the `n` parameter to specify how many groups to keep

8.
Dates and Times with lubridate

v

Dates are tricky!

```
library(lubridate)
```

- Must load separately, not part of tidyverse
- Dates and datetimes only
  - No time only class
- Use only dates if possible!

```
today()
```

```
now()
```

Dates from strings

R needs to “know” a date is a date

Lubridate helps without using the %m/%d/%Y etc.

`ymd()`

`mdy()`

`ymd_hms()`

`mdy_hm()`

Can get dates from individual components--less common

`make_date()`

Getting components of dates

What if you have a date column but want to subset your data by month?

```
> today<-today()
> month(today)
[1] 11
> month(today, label=TRUE)
[1] Nov
12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < ... < Dec
```

You can pull out individual parts of the date with the accessor functions `year()`, `month()`, `mday()` (day of the month), `yday()` (day of the year), `wday()` (day of the week), `hour()`, `minute()`, and `second()`.

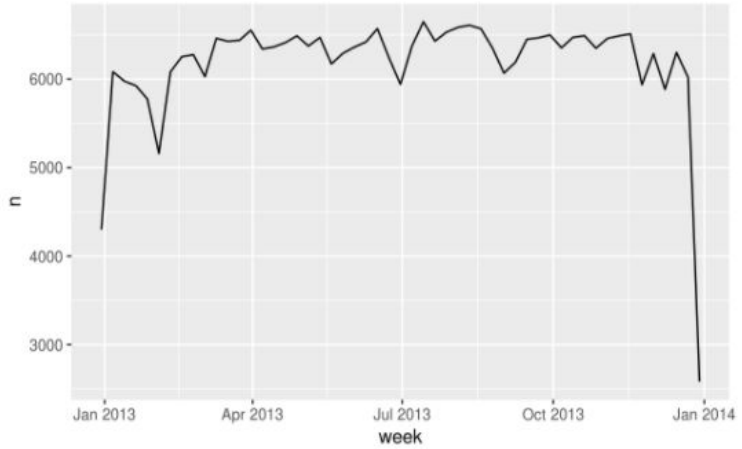
```
datetime <- ymd_hms("2016-07-08 12:34:56")

year(datetime)
#> [1] 2016
month(datetime)
#> [1] 7
mday(datetime)
#> [1] 8

yday(datetime)
#> [1] 190
wday(datetime)
#> [1] 6
```

For `month()` and `wday()` you can set `label = TRUE` to return the abbreviated name of the month or day of the week. Set `abbr = FALSE` to return the full name.

Rounding

An alternative approach to plotting individual components is to round the date to a nearby unit of time, with `floor_date()`, `round_date()`, and `ceiling_date()`. Each function takes a vector of dates to adjust and then the name of the unit round down (floor), round up (ceiling), or round to. This, for example, allows us to plot the number of flights per week:

```
flights_dt %>%
  count(week = floor_date(dep_time, "week")) %>%
  ggplot(aes(week, n)) +
    geom_line()
```



Computing the difference between a rounded and unrounded date can be particularly useful.

Set components of date time

Say you have a date time, `datetime`, and you want to know if something was done within an hour of that datetime--create a variable `datetime` + 1 hour

```
(datetime <- ymd_hms("2016-07-08 12:34:56"))
#> [1] "2016-07-08 12:34:56 UTC"


year(datetime) <- 2020
datetime
#> [1] "2020-07-08 12:34:56 UTC"
month(datetime) <- 01
datetime
#> [1] "2020-01-08 12:34:56 UTC"
hour(datetime) <- hour(datetime) + 1
datetime
#> [1] "2020-01-08 13:34:56 UTC"
```

Time spans

Pick the most simple that you need!

Duration--only physical time

- Subtract two dates--difftime object
- Lubridate always seconds, add/multiply!
- `as.duration(), dseconds(), dminutes()`

Period--works with human times

- `seconds(), hours(), minutes()`, add/multiply!

Interval--how long a span is in human units

- Duration with a starting and ending point

Time spans

	date				date time				duration				period				interval				number			
date	-								-	+			-	+							-	+		
date time					-				-	+			-	+							-	+		
duration	-	+			-	+			-	+		/									-	+	×	/
period	-	+			-	+							-	+							-	+	×	/
interval												/				/								
number	-	+			-	+			-	+	×		-	+	×		-	+	×		-	+	×	/

Time zones

Very complicated!

Ambiguous across the world and within the US

- For example, there are time zones for both “America/New_York” and “America/Detroit”. These cities both currently use Eastern Standard Time but in 1969-1972 Michigan (the state in which Detroit is located), did not follow DST, so it needs a different name. It’s worth reading the raw time zone database

Lubridate always uses UTC (Universal Coordinated Time)

R-Ladies Austin

Upcoming Events

ALL the Ladies in Tech Happy Hour [December 5]

R4DS: Programming -- Functions, Vectors, and Iteration [December 13]

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

Book Club: Weapons of Math Destruction [January 31]