Getting / cleaning data 2

Tidy data

Tidy data

All of the material in this section comes directly from Hadley Wickham's paper on tidy data. You will need to read this paper to prepare for the quiz on this section.

Characteristics of tidy data

Characteristics of tidy data are:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Getting your data into a "tidy" format makes it easier to model and plot. By taking the time to tidy your data at the start of an analysis, you will save yourself time, and make it easier to plan out, later steps.

Five common problems

Here are five common problems that Hadley Wickham has identified that keep data from being tidy:

1. Column headers are values, not variable names.
2. Multiple variables are stored in one column.
3. Variables are stored in both rows and columns.
4. Multiple types of observational units are stored in the same table.
5. A single observational unit is stored in multiple tables.

In the following slides, I'll give examples of each of these problems.

Five common problems

(1.) Column headers are values, not variable names.

religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don't know/refused	15	14	15	11	10	35
Evangelical Prot	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically Black Prot	228	244	236	238	197	223
Jehovah's Witness	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

Five common problems

Solution:

religion	income	freq
Agnostic	<$10k	27
Agnostic	$10-20k	34
Agnostic	$20-30k	60
Agnostic	$30-40k	81
Agnostic	$40-50k	76
Agnostic	$50-75k	137
Agnostic	$75-100k	122
Agnostic	$100-150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Five common problems

(2.) Multiple variables are stored in one column.

country	year	column	cases
AD	2000	m014	0
AD	2000	m1524	0
AD	2000	m2534	1
AD	2000	m3544	0
AD	2000	m4554	0
AD	2000	m5564	0
AD	2000	m65	0
AE	2000	m014	2
AE	2000	m1524	4
AE	2000	m2534	4
AE	2000	m3544	6
AE	2000	m4554	5
AE	2000	m5564	12
AE	2000	m65	10
AE	2000	f014	3

Five common problems

Solution:

country	year	sex	age	cases
AD	2000	m	0-14	0
AD	2000	m	15-24	0
AD	2000	m	25-34	1
AD	2000	m	35-44	0
AD	2000	m	45-54	0
AD	2000	m	55-64	0
AD	2000	m	65+	0
AE	2000	m	0-14	2
AE	2000	m	15-24	4
AE	2000	m	25-34	4
AE	2000	m	35-44	6
AE	2000	m	45-54	5
AE	2000	m	55-64	12
AE	2000	m	65+	10
AE	2000	f	0-14	3

Five common problems

(3.) Variables are stored in both rows and columns.

id	year	month	element	d1	d2	d3	d4	d5	d6	d7	d8
MX17004	2010	1	tmax	—	—	—	—	—	—	—	—
MX17004	2010	1	tmin	—	—	—	—	—	—	—	—
MX17004	2010	2	tmax	—	27.3	24.1	—	—	—	—	—
MX17004	2010	2	tmin	—	14.4	14.4	—	—	—	—	—
MX17004	2010	3	tmax	—	—	—	—	32.1	—	—	—
MX17004	2010	3	tmin	—	—	—	—	14.2	—	—	—
MX17004	2010	4	tmax	—	—	—	—	—	—	—	—
MX17004	2010	4	tmin	—	—	—	—	—	—	—	—
MX17004	2010	5	tmax	—	—	—	—	—	—	—	—
MX17004	2010	5	tmin	—	—	—	—	—	—	—	—

Five common problems

Solution:

id	date	element	value
MX17004	2010-01-30	tmax	27.8
MX17004	2010-01-30	tmin	14.5
MX17004	2010-02-02	tmax	27.3
MX17004	2010-02-02	tmin	14.4
MX17004	2010-02-03	tmax	24.1
MX17004	2010-02-03	tmin	14.4
MX17004	2010-02-11	tmax	29.7
MX17004	2010-02-11	tmin	13.4
MX17004	2010-02-23	tmax	29.9
MX17004	2010-02-23	tmin	10.7

id	date	tmax	tmin
MX17004	2010-01-30	27.8	14.5
MX17004	2010-02-02	27.3	14.4
MX17004	2010-02-03	24.1	14.4
MX17004	2010-02-11	29.7	13.4
MX17004	2010-02-23	29.9	10.7
MX17004	2010-03-05	32.1	14.2
MX17004	2010-03-10	34.5	16.8
MX17004	2010-03-16	31.1	17.6
MX17004	2010-04-27	36.3	16.7
MX17004	2010-05-27	33.2	18.2

Five common problems

(4.) Multiple types of observational units are stored in the same table.

year	artist	time	track	date	week	rank
2000	2 Pac	4:22	Baby Don't Cry	2000-02-26	1	87
2000	2 Pac	4:22	Baby Don't Cry	2000-03-04	2	82
2000	2 Pac	4:22	Baby Don't Cry	2000-03-11	3	72
2000	2 Pac	4:22	Baby Don't Cry	2000-03-18	4	77
2000	2 Pac	4:22	Baby Don't Cry	2000-03-25	5	87
2000	2 Pac	4:22	Baby Don't Cry	2000-04-01	6	94
2000	2 Pac	4:22	Baby Don't Cry	2000-04-08	7	99
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-02	1	91
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-09	2	87
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-16	3	92
2000	3 Doors Down	3:53	Kryptonite	2000-04-08	1	81
2000	3 Doors Down	3:53	Kryptonite	2000-04-15	2	70
2000	3 Doors Down	3:53	Kryptonite	2000-04-22	3	68
2000	3 Doors Down	3:53	Kryptonite	2000-04-29	4	67
2000	3 Doors Down	3:53	Kryptonite	2000-05-06	5	66

Five common problems

Solution:

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

id	date	rank
1	2000-02-26	87
1	2000-03-04	82
1	2000-03-11	72
1	2000-03-18	77
1	2000-03-25	87
1	2000-04-01	94
1	2000-04-08	99
2	2000-09-02	91
2	2000-09-09	87
2	2000-09-16	92
3	2000-04-08	81
3	2000-04-15	70
3	2000-04-22	68
3	2000-04-29	67
3	2000-05-06	66

Five common problems

(5.) A single observational unit is stored in multiple tables.

Example: exposure and outcome data stored in different files:

- File 1: Daily mortality counts
- File 2: Daily air pollution measurements

In-course exercise

We'll take a break now to do the In-Course Exercise (Section 1 of the In-course Exercise for Chapter 6).

Joining datasets

Joining datasets

So far, you have only worked with a single data source at a time. When you work on your own projects, however, you typically will need to merge together two or more datasets to create the a data frame to answer your research question.

For example, for air pollution epidemiology, you will often have to join several datasets:

- Health outcome data (e.g., number of deaths per day)
- Air pollution concentrations
- Weather measurements (since weather can be a confounder)
- Demographic data

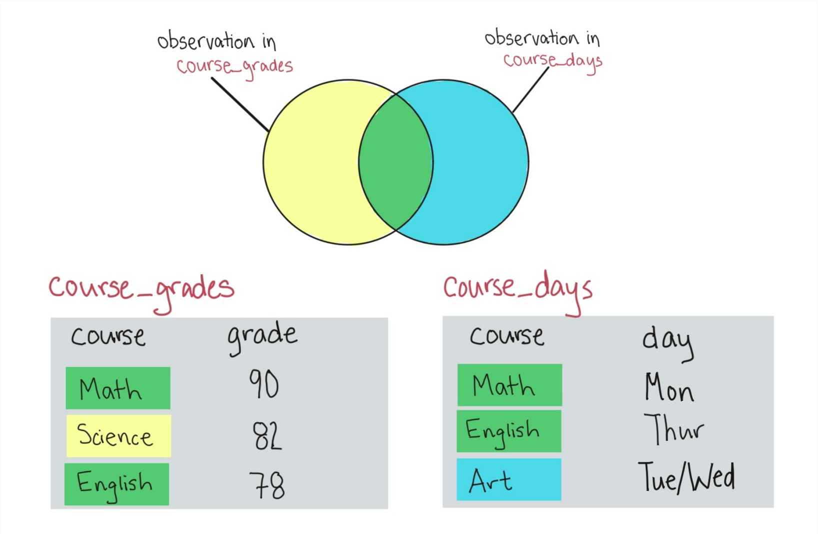
`*_join` functions

The `dplyr` package has a family of different functions to join two dataframes together, the `*_join` family of functions. These include:

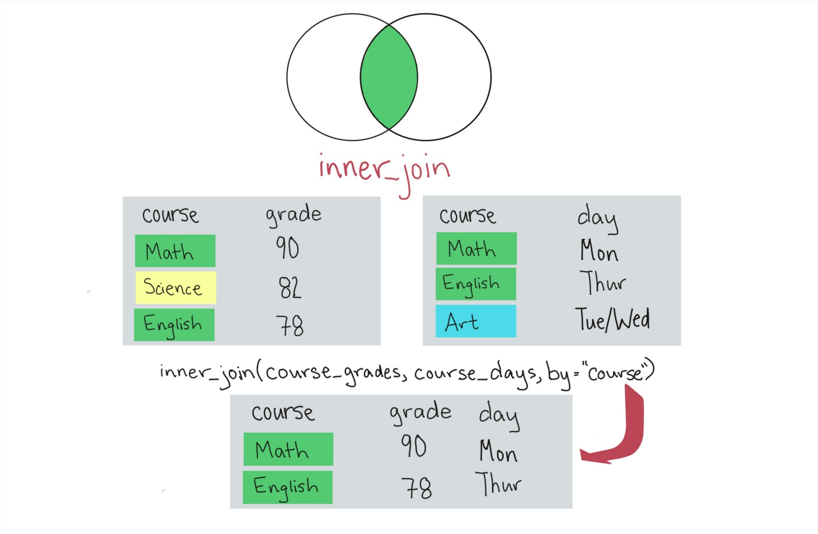
- `inner_join`
- `full_join`
- `left_join`
- `right_join`

All combine two dataframes, which I'll call `course_grades` and `course_days` here.

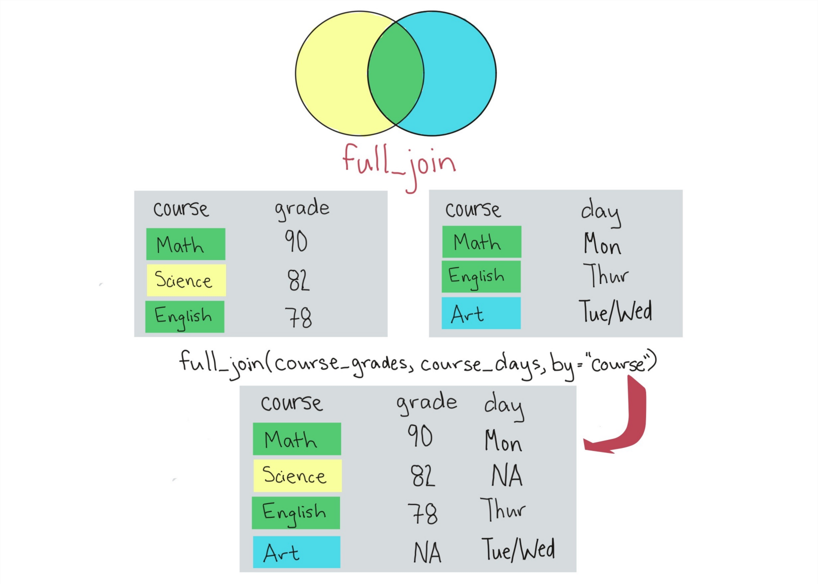
*_join functions



inner_join

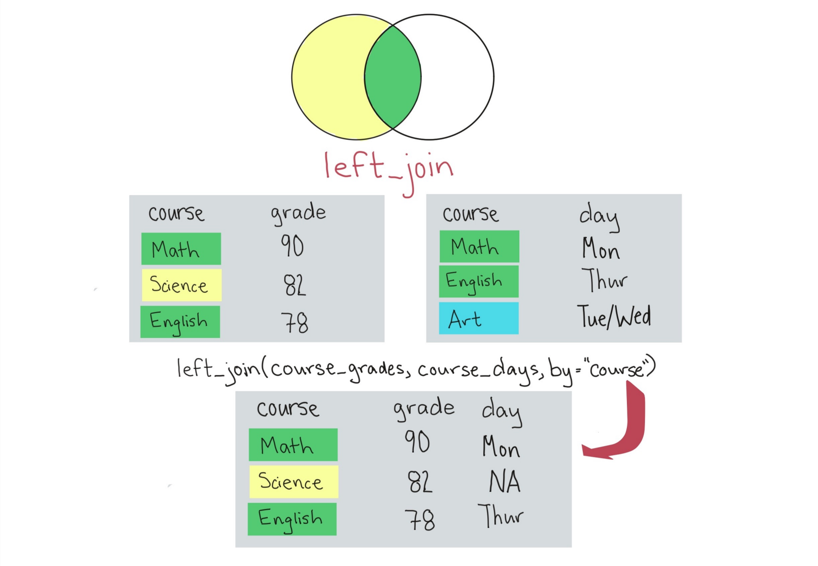

inner_join
course
grade
Math
90
Science
82
English
78
course
day
Math
Mon
English
Thur
Art
Tue/Wed
inner_join(course_grades, course_days, by="course")
course
grade
day
Math
90
Mon
English
78
Thur

full_join

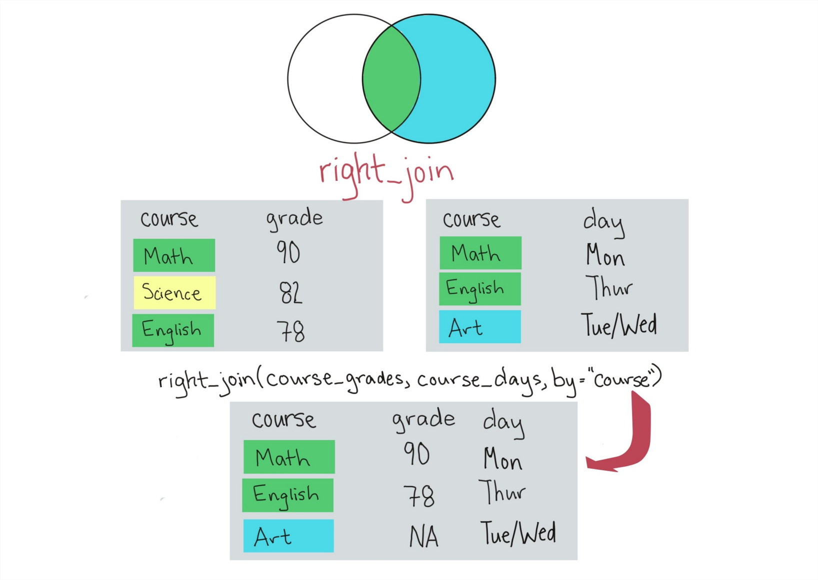


full_join
course
grade
Math
90
Science
82
English
78
course
day
Math
Mon
English
Thur
Art
Tue/Wed
full_join(course_grades, course_days, by="course")
course
grade
day
Math
90
Mon
Science
82
NA
English
78
Thur
Art
NA
Tue/Wed

left_join


left_join
course grade
Math 90
Science 82
English 78
course day
Math Mon
English Thur
Art Tue/Wed
left_join(course_grades, course_days, by="course")
course grade day
Math 90 Mon
Science 82 NA
English 78 Thur

right_join


right_join
course
grade
Math
90
Science
82
English
78
course
day
Math
Mon
English
Thur
Art
Tue/Wed
right_join(course_grades, course_days, by = "course")
course
grade
day
Math
90
Mon
English
78
Thur
Art
NA
Tue/Wed

*_join functions

For some more complex examples of using join, I'll use these example datasets (x and y):

```
## # A tibble: 4 x 3
##   course grade student
##   <chr>  <dbl> <chr>
## 1 x         92 a
## 2 x         90 b
## 3 y         82 a
## 4 z         78 b
```

```
## # A tibble: 4 x 3
##   class day       student
##   <chr> <chr>     <chr>
## 1 w     Tues      a
## 2 x     Mon / Fri a
## 3 x     Mon / Fri b
## 4 y     Tue       a
```

*_join functions

If you have two datasets you want to join, but the column names for the joining column are different, you can use the by argument:

```
full_join(x, y, by = list(x = "course", y = "class"))
```

```
## # A tibble: 7 x 5
##   course grade student.x day       student.y
##   <chr>  <dbl> <chr>     <chr>     <chr>
## 1 x         92 a         Mon / Fri a
## 2 x         92 a         Mon / Fri b
## 3 x         90 b         Mon / Fri a
## 4 x         90 b         Mon / Fri b
## 5 y         82 a         Tue       a
## 6 z         78 b         <NA>      <NA>
## 7 w         NA <NA>      Tues      a
```

`*_join` functions

A few things to note about this example:

- The joining column name for the "left" dataframe (`x` in this case) is used as the column name for the joined data
- `student` was a column name in both `x` and `y`. If we're not using it to join the data, the column names are changed in the joined data to `student.x` and `student.y`.
- Values are recycled for rows where there were multiple matches across the dataframe (e.g., rows for course "x")

*_join functions

Sometimes, you will want to join by more than one column. In this example data, it would make sense to join the data by matching both course and student. You can do this by using a vector of all columns to join on:

```
full_join(x, y, by = list(x = c("course", "student"),
                          y = c("class", "student")))
```

```
## # A tibble: 5 x 4
##   course grade student day
##   <chr>  <dbl> <chr>   <chr>
## 1 x         92 a       Mon / Fri
## 2 x         90 b       Mon / Fri
## 3 y         82 a       Tue
## 4 z         78 b       <NA>
## 5 w         NA a       Tues
```

In-course exercise

We'll take a break now to do the In-Course Exercise (Section 2 of the In-course Exercise for Chapter 6).

Longer and wider data

`pivot_longer` / `pivot_wider`

There are two functions from the `tidyr` package (another member of the tidyverse) that you can use to change between wide and long data: `pivot_longer` and `pivot_wider`.

These are brand new, and they replace the older `gather` and `spread` functions. To use the new functions, you many need to install the development version of the `tidyr` package. You can do that with:

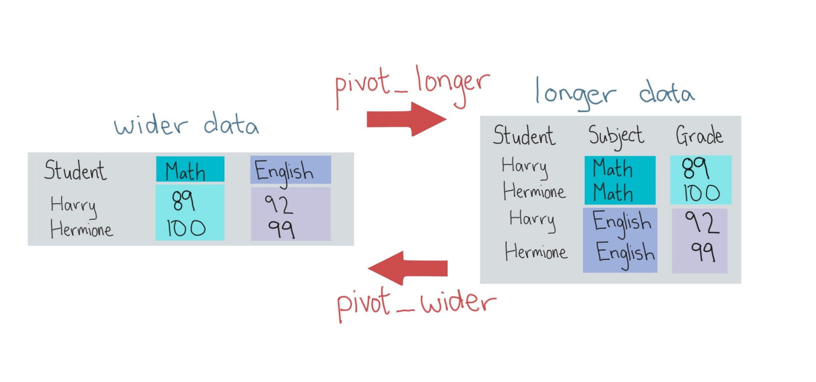
```r
devtools::install_github("tidyverse/tidyr")
```

pivot_longer / pivot_wider

Here is a description of these two functions:

- `pivot_longer`: Takes several columns and pivots them down into two columns. One of the new columns contains the former column names and the other contains the former cell values.
- `pivot_wider`: Takes two columns and pivots them up into multiple columns. Column names for the new columns will come from one column and the cell values from the other.

pivot_longer / pivot_wider



pivot_longer / pivot_wider

The following examples show the effects of making a dataset longer or wider.

Here is some example wide data:

```
hogwarts_wide
```

```
## # A tibble: 2 x 4
##   student   math english science
##   <chr>    <dbl>   <dbl>   <dbl>
## 1 Harry       89      92      93
## 2 Hermione   100      99      98
```

pivot_longer / pivot_wider

In the hogwarts_wide dataset, there are separate columns for three different courses (math, english, and science). Each cell gives the value for a certain stock on a certain day.

```
hogwarts_wide
```

```
## # A tibble: 2 x 4
##   student   math english science
##   <chr>    <dbl>   <dbl>   <dbl>
## 1 Harry       89      92      93
## 2 Hermione   100      99      98
```

This data isn't "tidy", because the identify of the course (math, english, or science) is a variable, and you'll probably want to include it as a variable in modeling.

pivot_longer / pivot_wider

If you want to convert the dataframe to have all stock values in a single column, you can use `pivot_longer` to convert wide data to long data:

```
library("tidyr")
hogwarts_long <- pivot_longer(data = hogwarts_wide,
                              cols = math:science,
                              names_to = "subject",
                              values_to = "grade")
```

pivot_longer / pivot_wider

In this "longer" dataframe, there is now one column that gives the identify of the course (`subject`) and another column that gives the grade a student got for that course (`grade`):

```
hogwarts_long
```

```
## # A tibble: 6 x 3
##   student  subject grade
##   <chr>    <chr>   <dbl>
## 1 Harry    math       89
## 2 Harry    english    92
## 3 Harry    science    93
## 4 Hermione math      100
## 5 Hermione english    99
## 6 Hermione science    98
```

pivot_longer / pivot_wider

The format for a pivots_longer call is:

```
## Generic code
new_df <- pivot_longer(old_df,
                       cols = [name(s) of the columns you want
                               to make longer],
                       names_to = [name of new column to store
                                   the old column names],
                       values_to = [name of new column to store
                                    the old values])
```

`pivot_longer` / `pivot_wider`

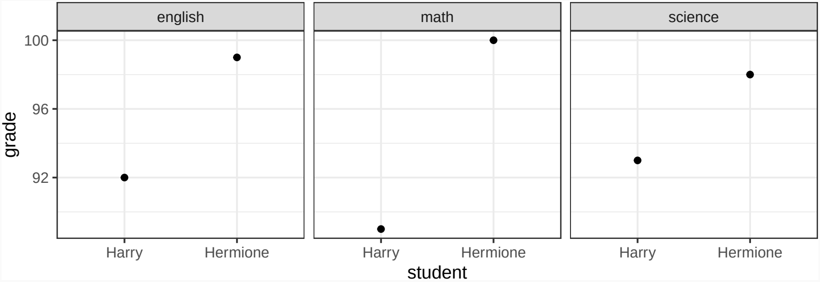
Three important notes:

- Everything is pivoted into one of two columns—one column with the old column names, and one column with the old cell values
- With the `names_to` and `values_to` arguments, you are just providing column names for the two columns that everything's pivoted into. When you are pivoting from "wide" to "long", you get to pick these names.
- If there is a column you don't want to include in the pivot (`date` in the example), use `-` to exclude it in the `cols` argument.

pivot_longer / pivot_wider

Notice how easy it is, now that the data is gathered, to use `subject` for aesthetics of faceting in a `ggplot2` call:

```
ggplot(hogwarts_long, aes(x = student, y = grade)) +
  geom_point() +
  facet_wrap(~ subject) +
  theme_bw()
```



pivot_longer / pivot_wider

If you have data in a "longer" format and would like to make it "wider", you can use `pivot_wider` to do that:

```
hogwarts <- pivot_wider(hogwarts_long,
                        names_from = "subject",
                        values_from = "grade")
```

Notice that this reverses the action of `pivot_longer`.

Further reading

Chapters 12 and 13 of "R for Data Science" are an excellent supplemental resource if you'd like extra practice on tidy data, pivoting, and joining different datasets.

Note: At this time "R for Data Science" uses the `gather` and `spread` instead of `pivot_*`. These are older functions, you should use `pivot_*`.

More with `dplyr`

dplyr

So far, you've used several dplyr functions:

- rename
- filter
- select
- mutate
- group_by
- summarize

Some other useful dplyr functions to add to your toolbox are:

- separate and unite
- mutate and other dplyr functions with group_by
- anti_join and semi_join

separate

Sometimes, you want to take one column and split it into two columns. For example, you may have information for two variables in one column:

```
ebola
```

```
## # A tibble: 4 x 1
##   ebola_key
##   <chr>
## 1 Liberia_Cases
## 2 Liberia_Deaths
## 3 Spain_Cases
## 4 Spain_Deaths
```

separate

If you have a consistent "split" character, you can use the separate function to split one column into two:

```
ebola %>%
  separate(col = ebola_key, into = c("country", "outcome"),
           sep = "_")
```

```
## # A tibble: 4 x 2
##   country outcome
##   <chr>   <chr>
## 1 Liberia Cases
## 2 Liberia Deaths
## 3 Spain   Cases
## 4 Spain   Deaths
```

separate

Here is the generic code for separate:

```
## Generic code
separate([dataframe],
         col = [name of the single column you want to split],
         into = [vector of the names of the columns
                 you want to create],
         sep = [the character that designates where
                you want to split])
```

separate

The default is to drop the original column and only keep the columns into which it was split. However, you can use the argument `remove = FALSE` to keep the first column, as well:

```
ebola %>%
  separate(col = ebola_key, into = c("country", "outcome"),
           sep = "_", remove = FALSE)
```

```
## # A tibble: 4 x 3
##   ebola_key      country outcome
##   <chr>          <chr>   <chr>
## 1 Liberia_Cases  Liberia Cases
## 2 Liberia_Deaths Liberia Deaths
## 3 Spain_Cases    Spain   Cases
## 4 Spain_Deaths   Spain   Deaths
```

You can use the `fill` argument (`fill = "right"` or `fill = "left"`) to control what happens when there are some observations that do not have the split character.

For example, say your original column looked like this:

```
## # A tibble: 4 x 1
##   ebola_key
##   <chr>
## 1 Liberia_Cases
## 2 Liberia
## 3 Spain_Cases
## 4 Spain_Deaths
```

separate

You can use `fill = "right"` to set how to split observations like the second one, where there is no separator character ("_"):

```
ebola %>%
  separate(col = ebola_key, into = c("country", "outcome"),
           sep = "_", fill = "right")
```

```
## # A tibble: 4 x 2
##   country outcome
##   <chr>   <chr>
## 1 Liberia Cases
## 2 Liberia <NA>
## 3 Spain   Cases
## 4 Spain   Deaths
```

In-course exercise

We'll take a break now to do the In-Course Exercise (Section 3 of the In-course Exercise for Chapter 6).

unite

The `unite` function does the reverse of the `separate` function: it lets you join several columns into a single column. For example, say you have data where year, month, and day are split into different columns:

```
## # A tibble: 4 x 3
##    year month   day
##   <dbl> <dbl> <int>
## 1  2016    10     1
## 2  2016    10     2
## 3  2016    10     3
## 4  2016    10     4
```

unite

You can use unite to join these into a single column:

```
date_example %>%
  unite(col = date, year, month, day, sep = "-")
```

```
## # A tibble: 4 x 1
##   date
##   <chr>
## 1 2016-10-1
## 2 2016-10-2
## 3 2016-10-3
## 4 2016-10-4
```

unite

If the columns you want to unite are in a row (and in the right order), you can use the `:` syntax with `unite`:

```
date_example %>%
  unite(col = date, year:day, sep = "-")
```

```
## # A tibble: 4 x 1
##   date
##   <chr>
## 1 2016-10-1
## 2 2016-10-2
## 3 2016-10-3
## 4 2016-10-4
```

Grouping with `mutate` versus `summarize`

So far, we have never used `mutate` with grouping.

You can use `mutate` after grouping– unlike `summarize`, the data will not be collapsed to fewer columns, but the summaries created by `mutate` will be added within each group.

For example, if you wanted to add the mean time by team to the `worldcup` dataset, you could do that with `group_by` and `mutate` (see next slide).

Grouping with mutate versus summarize

```
worldcup %>%
  group_by(Position) %>%
  mutate(mean_time = mean(Time)) %>%
  slice(1:2) %>% select(Team:Time, mean_time)
```

```
## # A tibble: 8 x 4
## # Groups:   Position [4]
##   Team        Position    Time mean_time
##   <fct>       <fct>      <int>     <dbl>
## 1 France      Defender     180      242.
## 2 Ghana       Defender     138      242.
## 3 Cameroon    Forward       46      167.
## 4 Uruguay     Forward       72      167.
## 5 Ivory Coast Goalkeeper   270      315.
## 6 Switzerland Goalkeeper   270      315.
## 7 Algeria     Midfielder    16      192.
## 8 Japan       Midfielder   351      192.
```

slice

You can also group by a factor first using `group_by`. Then, when you use `slice`, you will get the first few rows for each level of the group.

```
worldcup %>%
  group_by(Position) %>%
  slice(1:2)
```

```
## # A tibble: 8 x 7
## # Groups:   Position [4]
##   Team        Position    Time Shots Passes Tackles Saves
##   <fct>       <fct>      <int> <int>  <int>   <int> <int>
## 1 France      Defender     180     0     91       6     0
## 2 Ghana       Defender     138     0     51       2     0
## 3 Cameroon    Forward       46     2     16       0     0
## 4 Uruguay     Forward       72     0     15       0     0
## 5 Ivory Coast Goalkeeper   270     0     23       0     8
## 6 Switzerland Goalkeeper   270     0     75       0    11
## 7 Algeria     Midfielder    16     0      6       0     0
```

arrange with group_by

You can also group by a factor before arranging. In this case, all data for the first level of the factor will show up first, in the order given in arrange, then all data from the second level will show up in the specified order, etc.

```
worldcup %>%
  group_by(Team) %>%
  arrange(desc(Saves)) %>%
  slice(1) %>%
  head(n = 4)
```

```
## # A tibble: 4 x 7
## # Groups:   Team [4]
##   Team      Position    Time Shots Passes Tackles Saves
##   <fct>     <fct>      <int> <int>  <int>   <int> <int>
## 1 Algeria   Goalkeeper   180     0     30       0    12
## 2 Argentina Goalkeeper   450     0     47       0    10
## 3 Australia Goalkeeper   270     0     51       0    13
## 4 Brazil    Goalkeeper   450     0     69       0    10
```

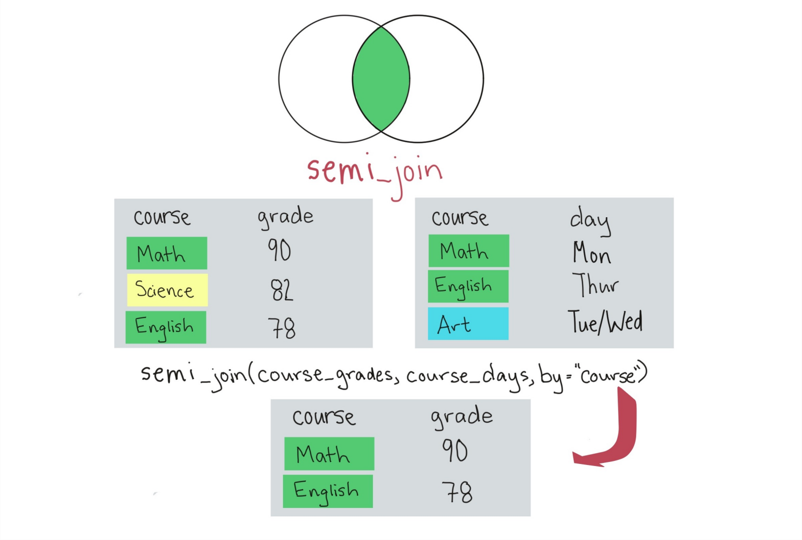
`semi_join` and `anti_join`

There are two more `*_join` functions we'll look at.

These functions allow you to filter one dataframe on only values that **do** have a match in a second dataframe (`semi_join`) or **do not** have a match in a second dataframe (`anti_join`).

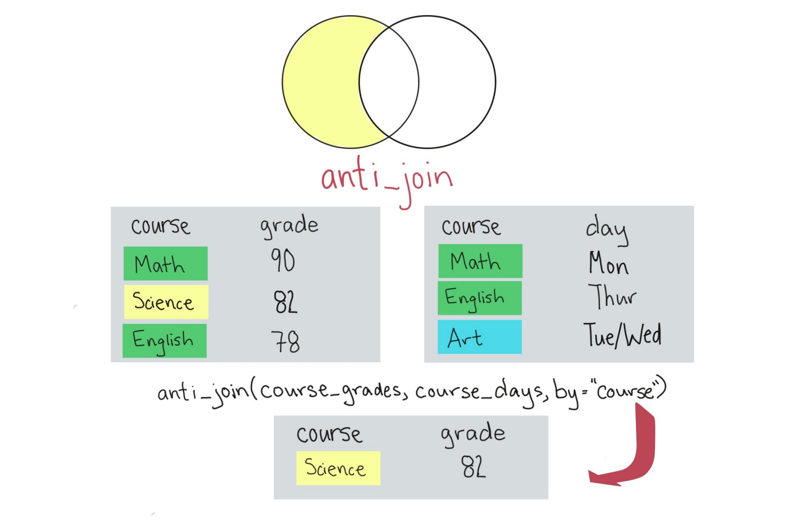
These functions do **not** bring in columns from the second dataset. Instead, they check the second dataset to decide whether or not to keep certain rows in the first dataset.

semi_join



The `semi_join` function filters to observations that **do** have a match in a second dataframe.

`anti_join`



The `anti_join` function filters to observations that **do not** have a match in a second dataframe.

In-course exercise

We'll take a break now to do the In-Course Exercise (Section 4 of the In-course Exercise for Chapter 6).

Tidying with `dplyr`

`VADeaths` data

For this example, I'll use the `VADeaths` dataset that comes with R.

This dataset gives the death rates per 1,000 people in Virginia in 1940. It gives death rates by age, gender, and rural / urban:

```
data("VADeaths")
VADeaths
```

```
##       Rural Male Rural Female Urban Male Urban Female
## 50-54       11.7          8.7       15.4          8.4
## 55-59       18.1         11.7       24.3         13.6
## 60-64       26.9         20.3       37.0         19.3
## 65-69       41.0         30.9       54.6         35.1
## 70-74       66.0         54.3       71.1         50.0
```

`VADeaths` data

There are a few things that make this data untidy:

- One variable (age category) is saved as row names, rather than a column.
- Other variables (gender, rural / urban) are in column names.
- Once you gather the data, you will have two variables (gender, rural / urban) in the same column.

In the following slides, we'll walk through how to tidy this data.

Tidying the `VADeaths` data

(1) One variable (age category) is saved as row names, rather than a column.

To fix this, we need to convert the row names into a new column. We can do this using `mutate` (load `tibble` if needed):

```
VADeaths %>%
  as_tibble() %>% ## Convert from matrix to dataframe
  rownames_to_column(var = "age")
```

```
## # A tibble: 5 x 5
##   age   `Rural Male` `Rural Female` `Urban Male` `Urban Femal
##   <chr>        <dbl>          <dbl>        <dbl>          <db
## 1 1             11.7            8.7         15.4             8
## 2 2             18.1           11.7         24.3            13
## 3 3             26.9           20.3         37              19
## 4 4             41             30.9         54.6            35
## 5 5             66             54.3         71.1            50
```

Tidying the VADeaths data

(2) Two variables (gender, rural / urban) are in column names.

Gather the data to convert column names to a new column:

```
VADeaths %>%
  as_tibble() %>%
  rownames_to_column(var = "age") %>%
  gather(key = gender_loc, value = mort_rate, - age) %>%
  slice(1:4)
```

```
## # A tibble: 4 x 3
##   age   gender_loc mort_rate
##   <chr> <chr>          <dbl>
## 1 1     Rural Male      11.7
## 2 2     Rural Male      18.1
## 3 3     Rural Male      26.9
## 4 4     Rural Male      41
```

Tidying the `VADeaths` data

(3) Two variables (gender, rural / urban) in the same column.

Separate the column into two separate columns for "gender" and "loc" (rural / urban):

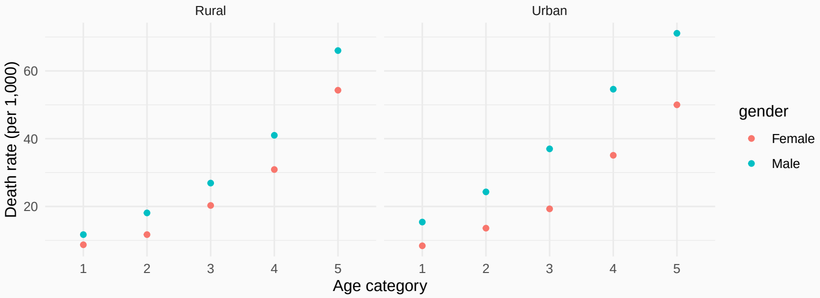
```
VADeaths %>%
  as_tibble() %>%
  rownames_to_column(var = "age") %>%
  gather(key = gender_loc, value = mort_rate, - age) %>%
  separate(col = gender_loc, into = c("gender", "loc"),
           sep = " ") %>%
  slice(1)
```

```
## # A tibble: 1 x 4
##   age   gender loc   mort_rate
##   <chr> <chr>  <chr>     <dbl>
## 1 1     Rural  Male       11.7
```

Tidying the `VADeaths` data

Now that the data is tidy, it's much easier to plot:

```r
ggplot(VADeaths, aes(x = age, y = mort_rate,
                     color = gender)) +
  geom_point() +
  facet_wrap( ~ loc, ncol = 2) +
  xlab("Age category") + ylab("Death rate (per 1,000)") +
  theme_minimal()
```



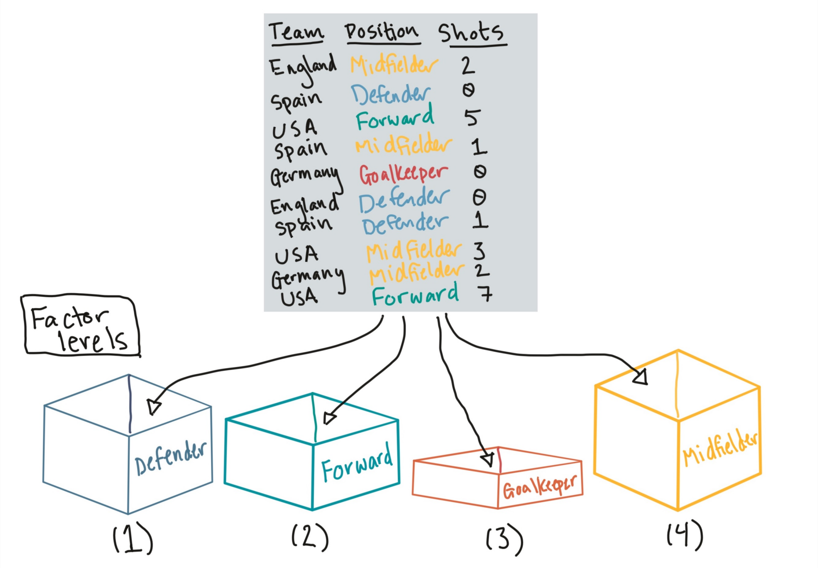
Working with factors

Working with factors

Hadley Wickham has developed a package called `forcats` that helps you work with factors.

```
library("forcats")
```

Factors


Team Position Shots
England Midfielder 2
Spain Defender 0
USA Forward 5
Spain Midfielder 1
Germany Goalkeeper 0
England Defender 0
Spain Defender 1
USA Midfielder 3
Germany Midfielder 2
USA Forward 7
Factor levels
Defender
Forward
Goalkeeper
Midfielder
(1)
(2)
(3)
(4)

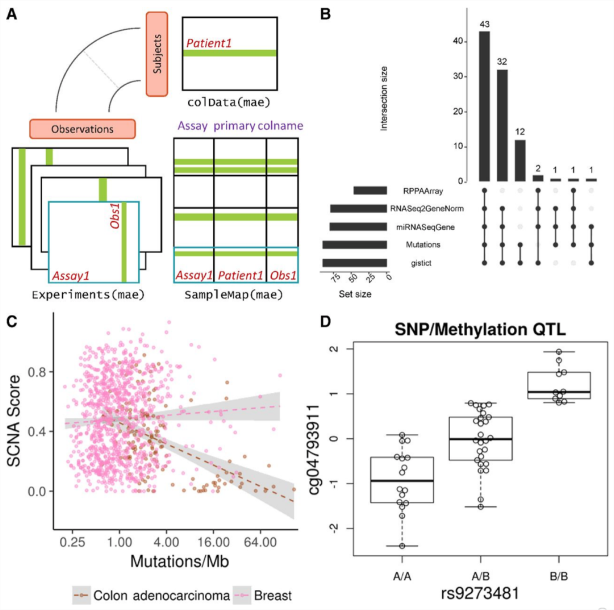
fct_recode

Bioconductor software

Biodiversity data

Bioconductor provides opensource software for bioinfomatics. Bioconductor provides an R package called `microbiome` that includes tools for exploring and analysing microbiome profiling data. The package provides sample datasets.

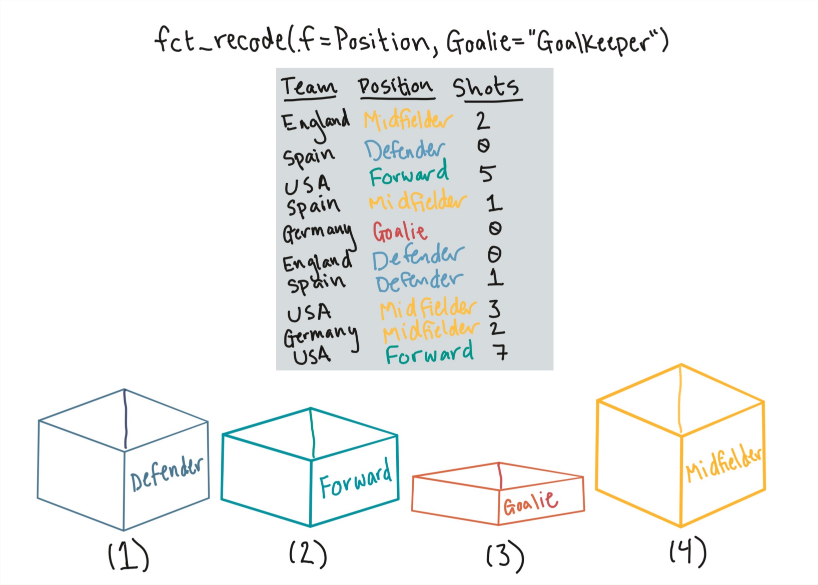
Biodiversity data



In-course exercise

We'll take a break now to do the In-Course Exercise (Section 6.6.5).

forcats


fct_recode(.f=Position, Goalie="GoalKeeper")
Team Position Shots
England Midfielder 2
Spain Defender 0
USA Forward 5
Spain Midfielder 1
Germany Goalie 0
England Defender 0
Spain Defender 1
USA Midfielder 3
Germany Midfielder 2
USA Forward 7
Defender
Forward
Goalie
Midfielder
(1)
(2)
(3)
(4)

fct_recode

The `fct_recode` function can be used to change the labels of a function (along the lines of using `factor` with `levels` and `labels` to reset factor labels).

One big advantage is that `fct_recode` lets you change labels for some, but not all, levels. For example, here are the team names:

```
worldcup %>%
  filter(Team == "USA") %>%
  slice(1:3) %>% select(Team, Position, Time)
```

```
##   Team   Position Time
## 1  USA Midfielder   10
## 2  USA   Defender  390
## 3  USA   Defender  200
```

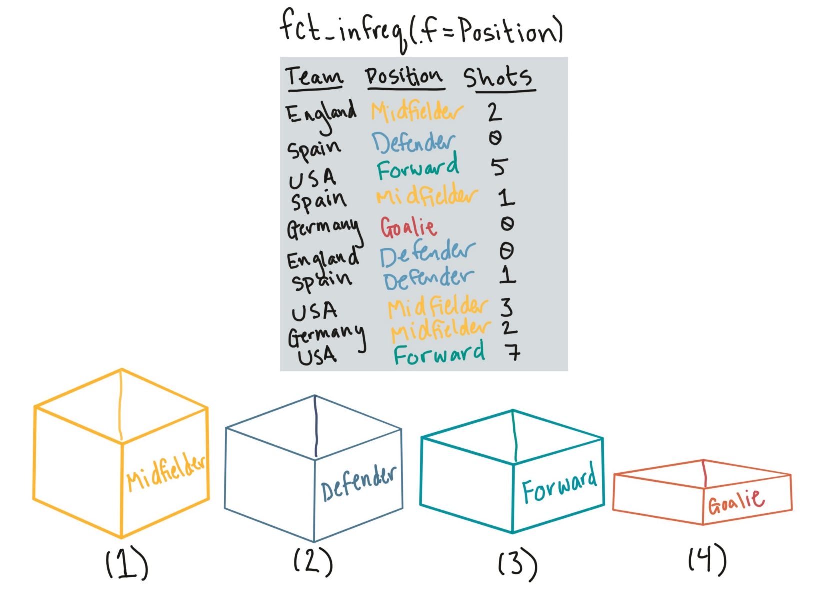
fct_recode

If you just want to change "USA" to "United States", you can run:

```
worldcup <- worldcup %>%
  mutate(Team = fct_recode(Team, `United States` = "USA"))
worldcup %>%
  filter(Team == "United States") %>%
  slice(1:3) %>% select(Team, Position, Time)
```

```
##            Team   Position Time
## 1 United States Midfielder   10
## 2 United States   Defender  390
## 3 United States   Defender  200
```

fct_infreq


fct_infreq(.f = Position)
Team Position Shots
England Midfielder 2
Spain Defender 0
USA Forward 5
Spain Midfielder 1
Germany Goalie 0
England Defender 0
Spain Defender 1
USA Midfielder 3
Germany Midfielder 2
USA Forward 7
Midfielder
(1)
Defender
(2)
Forward
(3)
Goalie
(4)

fct_infreq

You can use the `fct_infreq` function to reorder the levels of a factor from most common to least common:

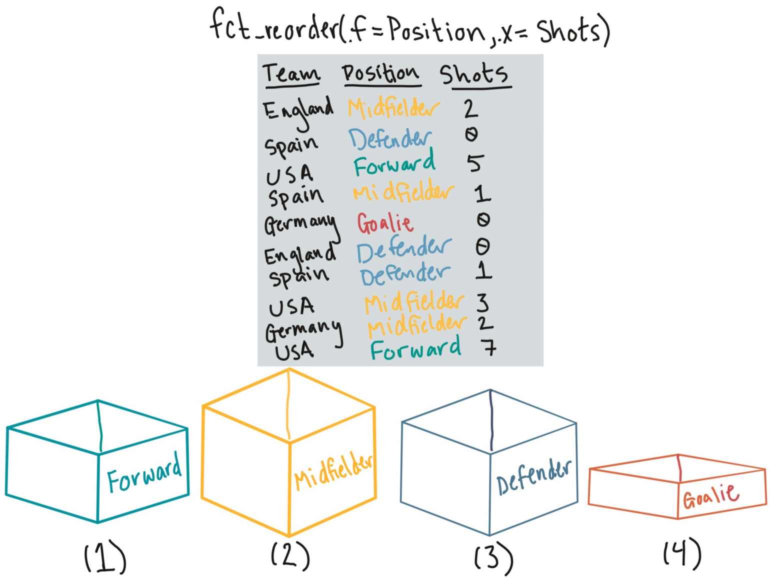
```
levels(worldcup$Position)
```

```
## [1] "Defender"   "Forward"    "Goalkeeper" "Midfielder"
```

```
worldcup <- worldcup %>%
  mutate(Position = fct_infreq(Position))
levels(worldcup$Position)
```

```
## [1] "Midfielder" "Defender"   "Forward"    "Goalkeeper"
```

fct_reorder



fct_reorder(.f = Position, .x = Shots)
Team Position Shots
England Midfielder 2
Spain Defender 0
USA Forward 5
Spain Midfielder 1
Germany Goalie 0
England Defender 0
Spain Defender 1
USA Midfielder 3
Germany Midfielder 2
USA Forward 7
Forward
Midfielder
Defender
Goalie
(1)
(2)
(3)
(4)

fct_reorder

If you want to reorder one factor by another variable (ascending order), you can use `fct_reorder` (e.g., homework 3). For example, to re-level `Position` by the median shots on goals for each position, you can run:

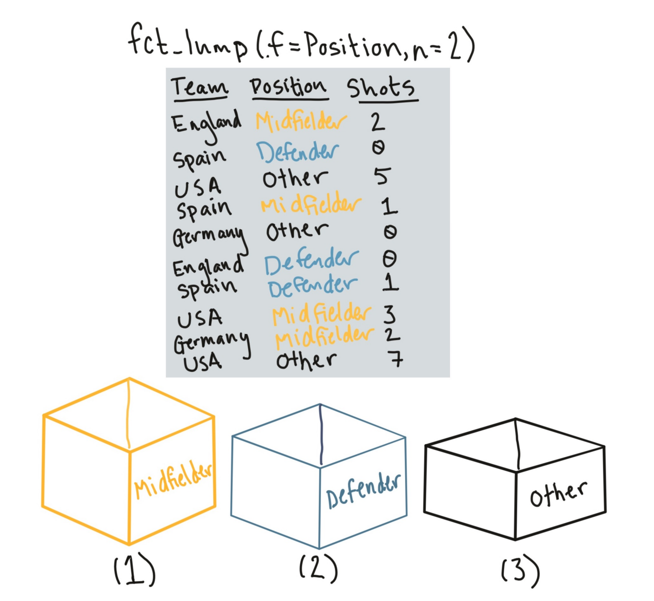
```
levels(worldcup$Position)
```

```
## [1] "Midfielder" "Defender"   "Forward"    "Goalkeeper"
```

```
worldcup <- worldcup %>%
  mutate(Position = fct_reorder(Position, Shots))
levels(worldcup$Position)
```

```
## [1] "Goalkeeper" "Defender"   "Midfielder" "Forward"
```

fct_lump


fct_lump(.f=Position, n=2)
Team Position Shots
England Midfielder 2
Spain Defender 0
USA Other 5
Spain Midfielder 1
Germany Other 0
England Defender 0
Spain Defender 1
USA Midfielder 3
Germany Midfielder 2
USA Other 7
Midfielder
Defender
Other
(1)
(2)
(3)

fct_lump

You can use the `fct_lump` function to lump uncommon factors into an "Other" category. For example, to lump the two least common positions together, you can run (`n` specifies how many categories to keep outside of "Other"):

```
worldcup %>%
  mutate(Position = fct_lump(Position, n = 2)) %>%
  count(Position)
```

```
## # A tibble: 3 x 2
##   Position       n
##   <fct>      <int>
## 1 Defender     188
## 2 Midfielder   228
## 3 Other        179
```

In-course exercise

We'll now take a break to do Section 5 of the In-course Exercise for Chapter 6.

Working with strings

String operations

For these examples, we'll use some data on passengers of the Titanic. You can load this data using:

```
# install.packages("titanic")
library("titanic")
data("titanic_train")
```

We will be using the `stringr` package:

```
library("stringr")
```

String operations

This data includes a column called "Name" with passenger names. This column is somewhat messy and includes several elements that we might want to separate (last name, first name, title).

Here are the first few values of "Name":

```
titanic_train %>% select(Name) %>% slice(1:3)
```

```
##                                                  Name
## 1                             Braund, Mr. Owen Harris
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 3                              Heikkinen, Miss. Laina
```

String operations

The `str_trim` function from the `stringr` package allows you to trim leading and trailing whitespace:

```
with_spaces <- c("    a ", "  bob", " gamma")
with_spaces
```

```
## [1] "    a " "  bob"  " gamma"
```

```
str_trim(with_spaces)
```

```
## [1] "a"     "bob"   "gamma"
```

This is rarer, but if you ever want to, you can add leading and / or trailing whitespace to elements of a character vector with `str_pad` from the `stringr` package.

String operations

There are also functions to change a full character string to uppercase, lowercase, or title case:

```
str_to_upper("Braund, Mr. Owen Harris")
```

```
## [1] "BRAUND, MR. OWEN HARRIS"
```

```
str_to_lower("Braund, Mr. Owen Harris")
```

```
## [1] "braund, mr. owen harris"
```

```
str_to_title("braund, mr. owen harris")
```

```
## [1] "Braund, Mr. Owen Harris"
```

Regular expressions

Regular expressions

We've already done some things to manipulate strings. For example, if we wanted to separate "Name" into last name and first name (including title), we could actually do that with the `separate` function:

```
titanic_train %>%
  select(Name) %>%
  slice(1:3) %>%
  separate(Name, c("last_name", "first_name"), sep = ", ")
```

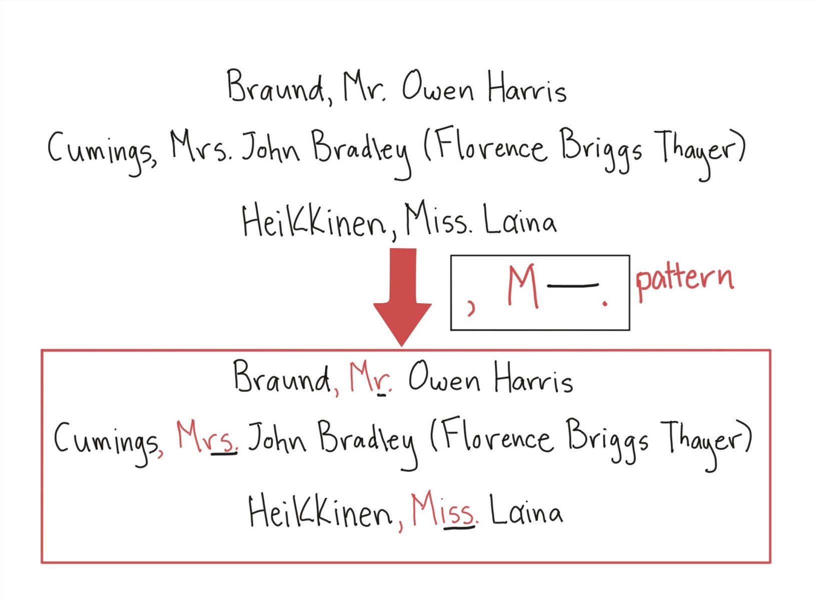
```
##   last_name                                        first_name
## 1    Braund                                   Mr. Owen Harris
## 2   Cumings Mrs. John Bradley (Florence Briggs Thayer)
## 3 Heikkinen                                       Miss. Laina
```

Regular expressions

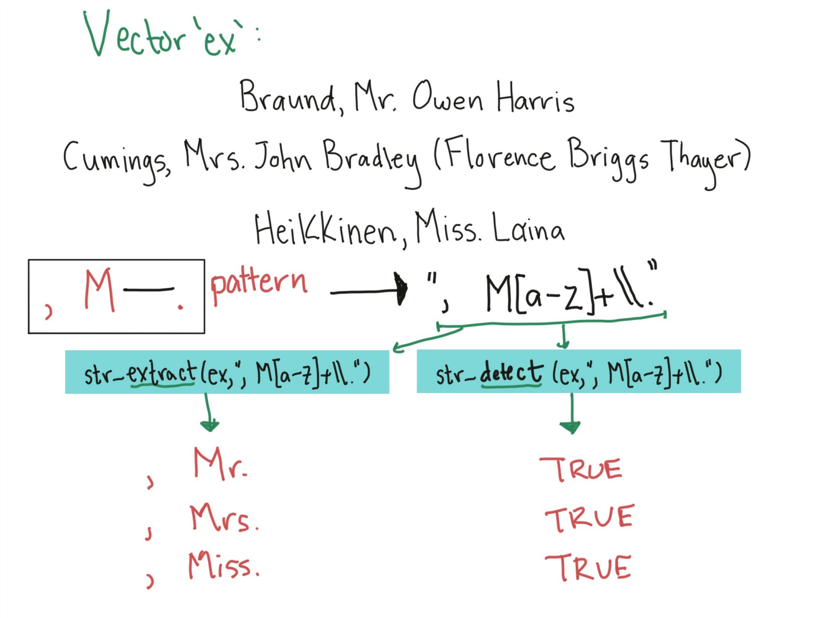
Notice that `separate` is looking for a regular pattern (“,”) and then doing something based on the location of that pattern in each string (splitting the string).

There are a variety of functions in R that can perform manipulations based on finding regular patterns in character strings.

Regular expressions



Regular expressions



Regular expressions

pattern: "Mr"

Strings	str_extract result	str_detect result
Mr.	Mr	TRUE
Mrs.	Mr	TRUE
Miss.	NA	FALSE
Dr.	NA	FALSE

Regular expression patterns

The easiest regular expression patterns are literal text. For example, the regular expression pattern if you're trying to match "Mr" is just "Mr":

```
ex_names <- c("Braund, Mr. Owen Harris",
              "Cumings, Mrs. John Bradley",
              "Heikkinen, Miss. Laina")
str_extract(ex_names, pattern = "Mr")
```

```
## [1] "Mr" "Mr" NA
```

Regular expression patterns

Regular expression patterns are case sensitive, so you won't match "Mr" with the pattern "mr":

```
ex_names <- c("Braund, Mr. Owen Harris",
              "Cumings, Mrs. John Bradley",
              "Heikkinen, Miss. Laina")
str_extract(ex_names, pattern = "mr")
```

```
## [1] NA NA NA
```

Regular expression patterns

There are a few characters called **metacharacters** that mean something special in regular expression patterns.

To use any of these literally in a regular expression, you need to "protect" them with two backslashes.

Regular expressions

pattern: "Mr."

Strings	str_extract result	str_detect result
Mr.	Mr.	TRUE
Mrs.	Mrs	TRUE
Miss.	NA	FALSE
Dr.	NA	FALSE

Regular expressions

pattern: "Mr\\."

Strings	str_extract result	str_detect result
Mr.	Mr.	TRUE
Mrs.	NA	FALSE
Miss.	NA	FALSE
Dr.	NA	FALSE

Regular expression patterns

For example, "." is a metacharacter, so to match "Mr.", you need to use the pattern "Mr\.":

```
ex_names <- c("Braund, Mr. Owen Harris",
              "Cumings, Mrs. John Bradley",
              "Heikkinen, Miss. Laina")
str_extract(ex_names, pattern = "Mr\\.")
```

```
## [1] "Mr." NA    NA
```

Regular expression metacharacters

Metacharacter	Use	To match literally
.	match any character	"\\."
*	match ≥0 of something	"*"
+	match ≥1 of something	"\\+"
[ ]	match a character in a subset	"\\[" "\\]"
^	depends on context	"\\^"
( )	extract part of a pattern	"(" ")"
?	match zero or one of something	"\\?"
{ }	customize number of times to match	"\\{" "\\}"
\	escape a metacharacter	"\\\"
$	match a pattern at the end of the string	"\\$"

Regular expression patterns

pattern: "Mr[s]*\\."

0 or more "s"s

Strings	str_extract result	str_detect result
Mr.	Mr.	TRUE
Mrs.	Mrs.	TRUE
Miss.	NA	FALSE
Dr.	NA	FALSE

Regular expression patterns

pattern: $\text{"M}\underbrace{\text{[a-z]+}}_{\text{1 or more lower case letters}}\text{\\\\."}$

strings	str_extract result	str_detect result
Mr.	Mr.	TRUE
Mrs.	Mrs.	TRUE
Miss.	Miss.	TRUE
Dr.	NA	FALSE

Regular expressions

The last pattern used `[a-z]+` to match one or more lowercase letters. The `[a-z]` is a **character class**.

You can also match digits (`[0-9]`), uppercase letters (`[A-Z]`), just some letters (`[aeiou]`), etc.

You can negate a character class by starting it with `^`. For example, `[^0-9]` will match anything that **isn't** a digit.

Regular expression patterns

1 uppercase character

pattern: "[A-Z][a-z]+\\."

1 or more lowercase letters

Strings	str_extract result	str_detect result
Mr.	Mr.	TRUE
Mrs.	Mrs.	TRUE
Miss.	Miss.	TRUE
Dr.	Dr.	TRUE

Regular expressions

The `str_detect` function will look through each element of a character vector for a designated pattern. If the pattern is there, it will return `TRUE`, and otherwise `FALSE`. The convention is:

```
## Generic code
str_detect(string = [vector you want to check],
           pattern = [pattern you want to check for])
```

For example, to create a logical vector specifying which of the Titanic passenger names include "Mrs.", you can call:

```r
mrs <- str_detect(titanic_train$Name, "Mrs\\.")
head(mrs)
```

```
## [1] FALSE  TRUE FALSE  TRUE FALSE FALSE
```

Regular expressions

The result is a logical vector, so `str_detect` can be used in `filter` to subset data to only rows where the passenger's name includes "Mrs.":

```
titanic_train %>%
  filter(str_detect(Name, "Mrs\\.")) %>%
  select(Name) %>%
  slice(1:3)
```

```
##                                                    Name
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 2        Futrelle, Mrs. Jacques Heath (Lily May Peel)
## 3   Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```

Regular expressions

The `str_extract` function can be used to extract a string (if it exists) from each value in a character vector. It follows similar conventions to `str_detect`:

```
## Generic code
str_extract(string = [vector you want to check],
            pattern = [pattern you want to check for])
```

Regular expressions

For example, you might want to extract "Mrs." if it exists in a passenger's name:

```
titanic_train %>%
  mutate(mrs = str_extract(Name, "Mrs\\.")) %>%
  select(Name, mrs) %>%
  slice(1:3)
```

```
##                                                  Name  mrs
## 1                             Braund, Mr. Owen Harris <NA>
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) Mrs.
## 3                              Heikkinen, Miss. Laina <NA>
```

Notice that now we're creating a new column (`mrs`) that either has "Mrs." (if there's a match) or is missing (`NA`) if there's not a match.

Regular expressions

For this first example, we were looking for an exact string ("Mrs"). However, you can use patterns that match a particular pattern, but not an exact string. For example, we could expand the regular expression to find "Mr." or "Mrs.":

```
titanic_train %>%
  mutate(title = str_extract(Name, "Mr[s]*\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
##                                                    Name title
## 1                               Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                                Heikkinen, Miss. Laina  <NA>
```

This pattern uses `[s]*` to match zero or more "s"s at this spot in the pattern.

Regular expressions

In the previous code, we found "Mr." and "Mrs.", but missed "Miss.". We could tweak the pattern again to try to capture that, as well. For all three, we have the pattern that it starts with "M", has some lowercase letters, and then ends with ".".

```
titanic_train %>%
  mutate(title = str_extract(Name, "M[a-z]+\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
##                                                  Name title
## 1                             Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                              Heikkinen, Miss. Laina Miss.
```

Regular expressions

Sometimes, you want to match a pattern, but then only subset a part of it. For example, each passenger seems to have a title ("Mr.", "Mrs.", etc.) that comes after "," and before ".". We can use this pattern to find the title, but then we get some extra stuff with the match:

```
titanic_train %>%
  mutate(title = str_extract(Name, ", [A-Z][a-z]*\\.")) %>%
  select(title) %>%
  slice(1:3)
```

```
##     title
## 1   , Mr.
## 2  , Mrs.
## 3 , Miss.
```

Regular expressions

We are getting things like ", Mr. ", when we really want "Mr". We can use the `str_match` function to do this. We group what we want to extract from the pattern in parentheses, and then the function returns a matrix. The first column is the full pattern match, and each following column gives just what matches within the groups.

```
head(str_match(titanic_train$Name,
          pattern = ", ([A-Z][a-z]*)\\."))
```

```
##      [,1]      [,2]
## [1,] ", Mr."   "Mr"
## [2,] ", Mrs."  "Mrs"
## [3,] ", Miss." "Miss"
## [4,] ", Mrs."  "Mrs"
## [5,] ", Mr."   "Mr"
## [6,] ", Mr."   "Mr"
```

Regular expressions

To get just the title, then, we can run:

```
titanic_train %>%
  mutate(title =
           str_match(Name, ", ([A-Z][a-z]*)\\.")[ , 2]) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
##                                                  Name title
## 1                             Braund, Mr. Owen Harris    Mr
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)   Mrs
## 3                              Heikkinen, Miss. Laina  Miss
```

The `[ , 2]` pulls out just the second column from the matrix returned by `str_match`.

Regular expressions

Here are some of the most common titles:

```
titanic_train %>%
  mutate(title =
           str_match(Name, ", ([A-Z][a-z]*)\\.")[ , 2]) %>%
  group_by(title) %>% summarize(n = n()) %>%
  arrange(desc(n)) %>% slice(1:5)
```

```
## # A tibble: 5 x 2
##   title      n
##   <chr>  <int>
## 1 Mr       517
## 2 Miss     182
## 3 Mrs      125
## 4 Master    40
## 5 Dr         7
```

Regular expressions

The following slides have a few other examples of regular expressions in action with this dataset.

Get just names that start with ("^") the letter "A":

```
titanic_train %>%
  filter(str_detect(Name, "^A")) %>%
  select(Name) %>%
  slice(1:3)
```

```
##                                                       Name
## 1                                 Allen, Mr. William Henry
## 2                              Andersson, Mr. Anders Johan
## 3 Asplund, Mrs. Carl Oscar (Selma Augusta Emilia Johansson)
```

Regular expressions

Get names with "II" or "III" ({2,} says to match at least two times):

```
titanic_train %>%
  filter(str_detect(Name, "I{2,}")) %>%
  select(Name) %>%
  slice(1:3)
```

```
##                                          Name
## 1  Carter, Master. William Thornton II
## 2 Roebling, Mr. Washington Augustus II
```

Regular expressions

Get names with "Andersen" or "Anderson" (alternatives in square brackets):

```
titanic_train %>%
  filter(str_detect(Name, "Anders[eo]n")) %>%
  select(Name)
```

```
##                                                      Name
## 1 Andersen-Jensen, Miss. Carla Christine Nielsine
## 2                             Anderson, Mr. Harry
## 3                   Walker, Mr. William Anderson
## 4                    Olsvigen, Mr. Thor Anderson
## 5      Soholt, Mr. Peter Andreas Lauritz Andersen
```

Regular expressions

Get names that start with ("^" outside of brackets) the letters "A" and "B":

```
titanic_train %>%
  filter(str_detect(Name, "^[AB]")) %>%
  select(Name) %>%
  slice(1:3)
```

```
##                            Name
## 1  Braund, Mr. Owen Harris
## 2 Allen, Mr. William Henry
## 3 Bonnell, Miss. Elizabeth
```

Regular expressions

Get names that end with ("$") the letter "b" (either lowercase or uppercase):

```
titanic_train %>%
  filter(str_detect(Name, "[bB]$")) %>%
  select(Name)
```

```
##                        Name
## 1   Emir, Mr. Farred Chehab
## 2 Goldschmidt, Mr. George B
## 3           Cook, Mr. Jacob
## 4          Pasic, Mr. Jakob
```

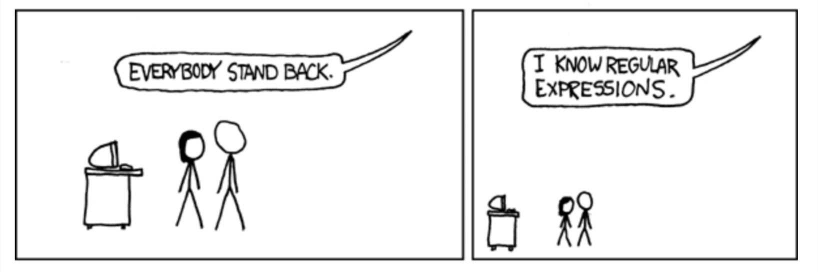
Regular expressions

There is a family of older, base R functions called `grep` that does something very similar.

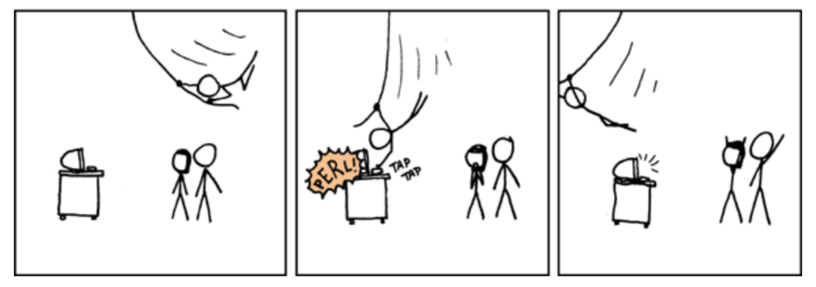
You may see these functions in example code.

Regular expressions

Regular expressions



Regular expressions



Tidy select

There are `tidyverse` functions to make selecting variables more straightforwards. You can call these functions as arguments of the `select` function to streamline variable selection. Examples include: `starts_with()`, `ends_with()`, and `contains()`.

Tidy select (helpers)

Here we use `starts_with("t")` to select all variables that begin with `t`.

```
titanic_train %>%
  select(starts_with("t")) %>%
  slice(1:3)
```

```
##              Ticket
## 1         A/5 21171
## 2          PC 17599
## 3 STON/O2. 3101282
```

Tidy select

The are also tidyverse functions that allow us to easily operate on a selection of variables. These functions are called `scoped varients`. You can identity these functions by these `_all`, `_at`, and `_if` suffixes.

Tidy select (*_if)

Here we use `select_if` to select all the numeric variables in a dataframe and covert their names to lower case (a handy function to tidy the variable names).

```
titanic_train %>%
  select_if(is.numeric, tolower) %>%
  slice(1:3)
```

```
##   passengerid survived pclass age sibsp parch    fare
## 1           1        0      3  22     1     0  7.2500
## 2           2        1      1  38     1     0 71.2833
## 3           3        1      3  26     0     0  7.9250
```

Tidy select (*_if)

The `select_if` function takes the following form.

```
## Generic code
new_df <- select_if(old_df,
                    .predicate [selects the variable to keep],
                    .funs = [the function to apply to
                             the selected columns])
```

Tidy select (*_at)

Here we use `select_at` to select all the variables that contain `ss` in their name and then covert their names to lower case (a handy function to tidy the variable names).

```
titanic_train %>%
  select_at(vars(contains("ss")), tolower) %>%
  slice(1:3)
```

```
##   passengerid pclass
## 1           1      3
## 2           2      1
## 3           3      3
```

Regular expressions

For more on these patterns, see:

- Help file for the `stringi-search-regex` function in the `stringi` package (which should install when you install `stringr`)
- Chapter 14 of R For Data Science
- http://gskinner.com/RegExr: Interactive tool for helping you build regular expression pattern strings

In-course exercise

We'll now take a break to do Section 6 of the In-course Exercise for Chapter 6.

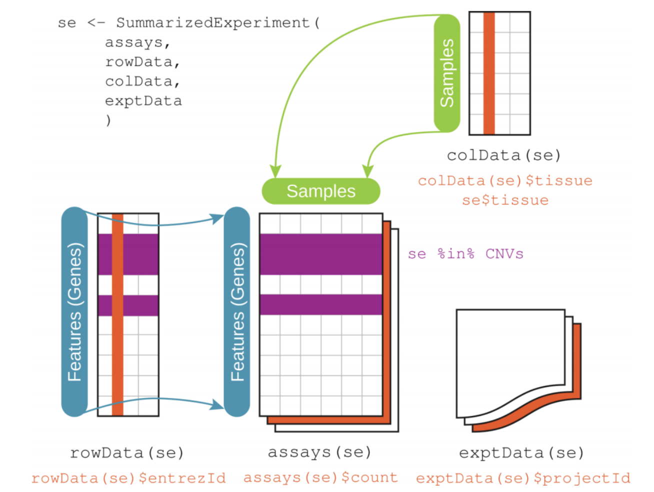
Bioconductor software

Biodiversity data

Bioconductor provides opensource software for bioinfomatics.

Bioconductor often uses some object types to store data that are different from the ones we've focused on so far (vectors and tibbles).

Bioconductor object classes



Source: Huber et al. Nature Methods 2015.

Biodiversity data

Bioconductor provides an R package called `microbiome` that includes tools for exploring and analysing microbiome profiling data. The package provides sample datasets.

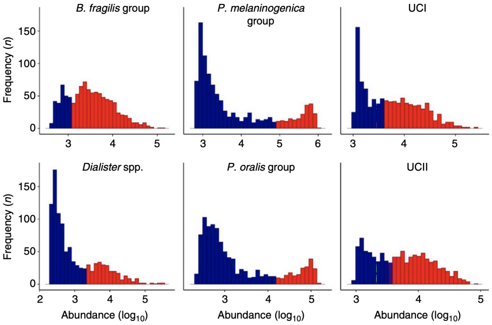
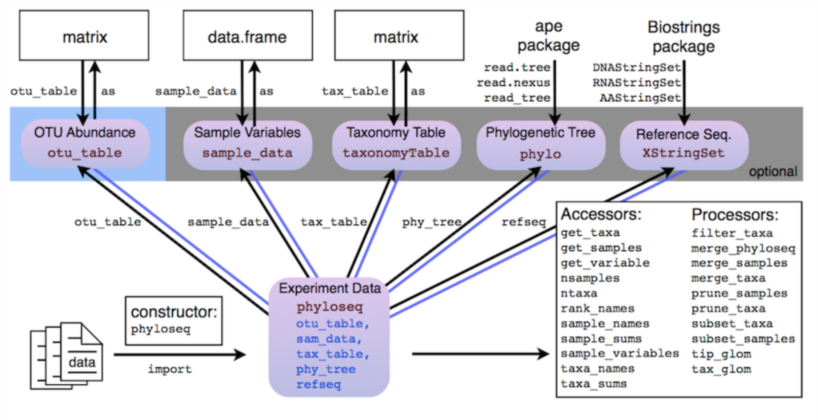


Figure 1 | The bimodal bacteria. Logarithmic abundance distributions of the six bimodal phylogenetic groups that exhibit robust alternative states of low (blue) and high (red) abundance across intestinal microbiota of 1,006 western adults. The UCI and UCII refer to the uncultured Clostridiales I and II, respectively. The frequency of the observations is shown as a function of the phylogenetic microarray $\log_{10}$ signal[12].

Source: Lahti et al. Nature Communications 2014.

Biodiversity data



Source: phyloseq tutorial.

In-course exercise

We'll take a break now to do the In-Course Exercise (Section 7 of the In-course Exercise for Chapter 6).