Data

Harlley Lima & Manel Slokom

November 21, 2018

Overview

Overview

Packages

Packages

- All R functions and datasets are stored in packages
- Some packages are installed with R and automatically loaded at the start of an R session:
  - The `base` package, where functions such as sqrt are defined
  - The `graphics` package, which allows plots to be generated

Packages

There are thousands of contributed packages for R. Some of them implement specialized statistical methods, others give access to data [1]...

Install

- Contributed packages can be downloaded and installed with the: install.packages function
- To download and install the package, type:
  `> install.packages("name_package")`

Load

- In order to use a package, we need to load the package with library:
  `library("name_package")`

[1] More about packages in R: http://r-pkgs.had.co.nz/intro.html

Overview

Tibbles

Prerequisites

In this tutorial we'll explore the `tibble` package, part of the core `tidyverse`:

- `library(tidyverse)`
- A tibble is a modern reimagining of the `data.frame`, keeping what time has proven to be effective, and throwing out what is not.

Creating tibbles

- You can create a new tibble from individual vectors with `tibble()`
- `tibble()` will automatically recycle inputs of length 1

```
tibble(
   x = 1:5,
   y = 1,
   z = x ^ 2 + y
 )
```

```
# A tibble: 5 x 3
      x     y     z
  <int> <dbl> <dbl>
1     1     1     2
2     2     1     5
3     3     1    10
4     4     1    17
5     5     1    26
```

Creating tibbles

If you're already familiar with `data.frame()`, note that `tibble()` does much less:

- it never changes the type of the inputs (e.g. it never converts strings to factors!)
- it never changes the names of variables, and it never creates row names

Creating tibbles

You might want to coerce a data frame to a tibble. You can do that with `as_tibble()`:

```
> as_tibble(iris)
# A tibble: 150 x 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl> <fct>
 1          5.1         3.5          1.4         0.2 setosa
 2          4.9         3            1.4         0.2 setosa
 3          4.7         3.2          1.3         0.2 setosa
 4          4.6         3.1          1.5         0.2 setosa
 5          5           3.6          1.4         0.2 setosa
 6          5.4         3.9          1.7         0.4 setosa
 7          4.6         3.4          1.4         0.3 setosa
 8          5           3.4          1.5         0.2 setosa
 9          4.4         2.9          1.4         0.2 setosa
10          4.9         3.1          1.5         0.1 setosa
# ... with 140 more rows
```

Tibbles vs. data.frame

Printing:

- Tibbles have a refined print method that shows only the first 10 rows
- All the columns that fit on screen
- This makes it much easier to work with large data

Tibbles vs. data.frame

Printing:

```
a <- tibble(
  #now = The current time
  #today = The current date
  #runif = uniform distribution
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  # Sample = it takes a sample of the specified size from the elements of x
  e = sample(letters, 1e3, replace = TRUE)
)
# A tibble: 1,000 x 5
   a                   b              c       d e
   <dttm>              <date>     <int>   <dbl> <chr>
 1 2018-11-19 15:28:40 2018-12-12     1 0.0813  n
 2 2018-11-20 06:41:09 2018-12-07     2 0.980   f
 3 2018-11-20 11:14:18 2018-12-16     3 0.731   e
 4 2018-11-20 02:34:51 2018-11-19     4 0.362   o
 5 2018-11-20 12:42:59 2018-11-27     5 0.00210 u
 6 2018-11-19 19:38:22 2018-11-28     6 0.260   i
 7 2018-11-20 07:56:00 2018-11-22     7 0.929   s
 8 2018-11-20 06:35:34 2018-12-14     8 0.732   m
 9 2018-11-20 01:47:59 2018-11-27     9 0.261   j
10 2018-11-20 03:08:34 2018-11-28    10 0.469   q
# ... with 990 more rows
```

Tibbles vs. data.frame

But sometimes you need more output than the default display:

```
print(a, n = 40, width = Inf)
```

- You can explicitly `print()` the data frame and control:
  - the number of rows (`n`)
  - the width of the display. width = Inf will display all columns
- A final option is to use RStudios built-in data viewer to get a scrollable view of the complete dataset: `View(a)`

Interacting with older code

Some older functions don't work with tibbles. If you encounter one of these functions, use: `as.data.frame()` to turn a tibble back to a data.frame

- `class(as.data.frame(tb))`
  `[1] "data.frame"`

Overview

Data import

Data import

In this tutorial, you'll learn how to load flat files in R with the `readr` package, which is part of the core tidyverse:

- `library(tidyverse)`

Data import

- We have seen how to load data sets included in packages
- However, we use to work with our own data sets
- We will describe different ways to import data into the R system

Importing data from text files

- We will import data from a very simple text file. Open the file `inputfile` with a text editor to see how the data are arranged
- The file contains three observations for 3 variables (x, y, z). The names of the variables are in the first line (header = TRUE). The columns are separated by white spaces (sep = " ")

```
x y z
10 2 3
4 12 6
13 14 65
```

- `data1 <- read.table("data/inputfile", header = TRUE, sep = " ")`

Importing data from text files

- Run the code and check that the object data1 has been created

```
> print(data1)
   x  y  z
1 10  2  3
2  4 12  6
3 13 14 65
```

- `> class(data1)`

  `"data.frame"`

- Function `str`: compactly display the internal structure of an R structure

```
> str(data1)
'data.frame':   3 obs. of  3 variables:
 $ x: int  10 4 13
 $ y: int  2 12 14
 $ z: int  3 6 65
```

Importing data from CSV files

- CSV: Comma-separated values

Year	Make	Model	Description	Price
1997	Ford	E350	ac, abs, moon	3000.00
1999	Chevy	Venture "Extended Edition"		4900.00
1999	Chevy	Venture "Extended Edition, Very Large"		5000.00
1996	Jeep	Grand Cherokee	MUST SELL! air, moon roof, loaded	4799.00

```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture ""Extended Edition, Very Large""",,5000.00
1996,Jeep,Grand Cherokee,"MUST SELL!
air, moon roof, loaded",4799.00
```

Importing data from CSV files

- The first argument to `read_csv()` is the most important: its the path to the file to read:
  `tree <- read_csv("data/trees.csv")`

Importing data from CSV files

When you run `read_csv()` it prints out a column specification that gives the name and type of each column

```
> tree <- read_csv("data/trees.csv")
Parsed with column specification:
cols(
    Index = col_double(),
    `Girth (in)` = col_double(),
    `Height (ft)` = col_double(),
    `Volume(ft3)` = col_double()
)
```

Importing data from CSV files

Sometimes there are a few lines of metadata at the top of the file. You can use `skip = n` to skip the first n lines

- `tree2 <- read_csv("data/trees2.csv", skip = 2)`

Or use comment = "#" to drop all lines that start with #

- `tree3 <- read_csv("data/trees3.csv", comment = "#")`

The data might not have column names. You can use `col_names = FALSE`

- `tree4 <- read_csv("data/trees4.csv", comment = "#", col_names = FALSE)`

Exporting data

You can also export the object as a comma-separated values files with the functions `write.csv`:

- `write.csv(tree, file = "data/treesOut.csv")`

Overview

What is tidy data?

- "Tidy" up your room!
- Please write your homework in a "tidy" way so that it is easier to grade and to provide feedback

"**Tidy data** means that your data follows a standardized format. This makes it easier for you and others to *visualize* your data, to *wrangle/transform* your data, and to *model* your data" ([a])

[a]https://moderndive.com/4-tidy.html

Read more here:
`https://www.jstatsoft.org/article/view/v059i10/v59i10.pdf`

Prerequisites

- tidyverse provides a bunch of tools to help tidy up your messy datasets
- `library(tidyverse)`

Tidy data

```
table1
#> # A tibble: 6 x 4
#>   country      year  cases population
#>   <chr>       <int>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
#> 3 Brazil       1999  37737  172006362
#> 4 Brazil       2000  80488  174504898
#> 5 China        1999 212258 1272915272
#> 6 China        2000 213766 1280428583
```

```
table2
#> # A tibble: 12 x 4
#>   country      year type           count
#>   <chr>       <int> <chr>          <int>
#> 1 Afghanistan  1999 cases            745
#> 2 Afghanistan  1999 population  19987071
#> 3 Afghanistan  2000 cases           2666
#> 4 Afghanistan  2000 population  20595360
#> 5 Brazil       1999 cases          37737
#> 6 Brazil       1999 population 172006362
#> # ... with 6 more rows
```

```
table3
#> # A tibble: 6 x 3
#>   country      year rate
#> * <chr>       <int> <chr>
#> 1 Afghanistan  1999 745/19987071
#> 2 Afghanistan  2000 2666/20595360
#> 3 Brazil       1999 37737/172006362
#> 4 Brazil       2000 80488/174504898
#> 5 China        1999 212258/1272915272
#> 6 China        2000 213766/1280428583
```

```
# Spread across two tibbles
table4a  # cases
#> # A tibble: 3 x 3
#>   country     `1999` `2000`
#> * <chr>        <int>  <int>
#> 1 Afghanistan    745   2666
#> 2 Brazil       37737  80488
#> 3 China       212258 213766
```

```
table4b  # population
#> # A tibble: 3 x 3
#>   country         `1999`     `2000`
#> * <chr>            <int>      <int>
#> 1 Afghanistan   19987071   20595360
#> 2 Brazil       172006362  174504898
#> 3 China       1272915272 1280428583
```

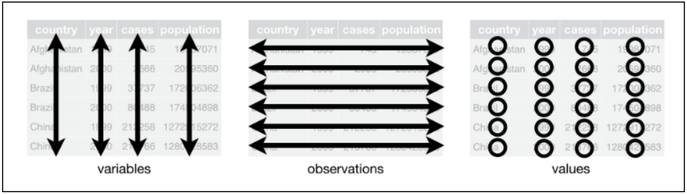
These are all representations of the same underlying data, but they are not equally easy to use

Tidy data

Rules

There are three interrelated rules which make a dataset tidy:

- Each variable must have its own column
- Each observation must have its own row
- Each value must have its own cell



Tidy data

Advantages

1. If you have a consistent data structure, it's easy to learn the tools that work with it
2. to place variables in columns

mutate

`mutate()`: adds new variables and preserves existing

```
> # Compute rate per 10,000
>   mutate(table1, rate = cases / population * 10000)
# A tibble: 6 x 5
  country      year    cases  population   rate
  <chr>       <dbl>    <dbl>       <dbl>  <dbl>
1 Afghanistan 1999.     745.   19987071. 0.373
2 Afghanistan 2000.    2666.   20595360. 1.29
3 Brazil      1999.   37737.  172006362. 2.19
4 Brazil      2000.   80488.  174504898. 4.61
5 China       1999.  212258. 1272915272. 1.67
6 China       2000.  213766. 1280428583. 1.67
```

Spreading and gathering

Spreading and gathering

Most data that you will encounter will be untidy:

- Most people aren't familiar with the principles of tidy data,
- Data is often organised to facilitate some use other than analysis.

Common problems

1. One **variable** might be **spread** across multiple columns.
2. One **observation** might be **scattered** across multiple rows.

Gathering

gather()

Spreading

spread()

Gathering

```
> table4a <- tibble (
+   country = c('Afghanistan', 'Brazil', 'China'),
+   `1999` = c(745, 37737, 212258),
+   `2000` = c(2666, 80488, 213766)
+
+ )
> table4a
# A tibble: 3 x 3
  country      `1999`  `2000`
  <chr>         <dbl>   <dbl>
1 Afghanistan    745.   2666.
2 Brazil       37737.  80488.
3 China       212258. 213766.
```

- A common problem: the column names are not names of variables, but values of a variable.
  - The column names 1999 and 2000 represent values of the variable year

Gathering

Todo

To gather those columns into a new pair of variables, we need three parameters:

1. The set of columns that represent values, not variables ("1999" and "2000")
2. The name of the variable whose values form the column names ("year")
3. The name of the variable whose values are spread over the cells ("cases")

Together those parameters generate the call to gather():

```
> gather(table4a, `1999`, `2000`, key = "year", value = "cases")
# A tibble: 6 x 3
  country     year   cases
  <chr>       <chr>  <dbl>
1 Afghanistan 1999     745.
2 Brazil      1999   37737.
3 China       1999  212258.
4 Afghanistan 2000    2666.
5 Brazil      2000   80488.
6 China       2000  213766.
```

Gathering

```
> gather(table4b, `1999`, `2000`, key = "year", value = "population")
# A tibble: 6 x 3
  country     year  population
  <chr>       <chr>      <int>
1 Afghanistan 1999    19987071
2 Brazil      1999   172006362
3 China       1999  1272915272
4 Afghanistan 2000    20595360
5 Brazil      2000   174504898
6 China       2000  1280428583
```

To combine the tidied versions of table4a and table4b into a single tibble

```
> tidy4a <- gather(table4a, `1999`, `2000`, key = "year", value = "cases")
> tidy4b <- gather(table4b, `1999`, `2000`, key = "year", value = "population")
> left_join(tidy4a, tidy4b)
Joining, by = c("country", "year")
# A tibble: 6 x 4
  country     year   cases population
  <chr>       <chr>  <dbl>      <dbl>
1 Afghanistan 1999    745.   19987071.
2 Brazil      1999  37737.  172006362.
3 China       1999 212258. 1272915272.
4 Afghanistan 2000   2666.   20595360.
5 Brazil      2000  80488.  174504898.
6 China       2000 213766. 1280428583.
```

Spreading

- Spreading is the opposite of gathering
- It is used when an observation is scattered across multiple rows

Spreading

Example

An observation is a country in a year, but each observation is spread across two rows.

```
> table2
# A tibble: 6 x 4
  country      year type            count
  <chr>       <dbl> <chr>           <dbl>
1 Afghanistan 1999. cases            745.
2 Afghanistan 1999. population  19987071.
3 Afghanistan 2000. cases           2666.
4 Afghanistan 2000. population  20595360.
5 Brazil      1999. cases          37737.
6 Brazil      1999. population 172006362.
```

```
> spread(table2, key = type, value = count)
# A tibble: 3 x 4
  country      year  cases population
  <chr>       <dbl>  <dbl>      <dbl>
1 Afghanistan 1999.   745.  19987071.
2 Afghanistan 2000.  2666.  20595360.
3 Brazil      1999. 37737. 172006362.
```

Separating and uniting

Separating and uniting

```
> table3 <- tibble(
+   country = c('Afghanistan', 'Afghanistan', 'Brazil', 'Brazil', 'China', 'China'),
+   year = c(1999, 2000, 1999, 2000, 1999, 2000),
+   rate = c('745/19987071', '2666/20595360', '37737/172006362', '80488/174504898', '21
5272', '213766/1280428583')
+ )
> table3
# A tibble: 6 x 3
  country      year rate
  <chr>       <dbl> <chr>
1 Afghanistan 1999. 745/19987071
2 Afghanistan 2000. 2666/20595360
3 Brazil      1999. 37737/172006362
4 Brazil      2000. 80488/174504898
5 China       1999. 212258/1272915272
6 China       2000. 213766/1280428583
```

Table3 has a different problem:
we have one column (rate) that contains two variables (cases and population)

Separate

separate() pulls apart one column into multiple columns.

```
> separate(table3, rate, into = c("cases", "population"))
# A tibble: 6 x 4
  country      year cases  population
  <chr>       <dbl> <chr>  <chr>
1 Afghanistan 1999. 745    19987071
2 Afghanistan 2000. 2666   20595360
3 Brazil      1999. 37737  172006362
4 Brazil      2000. 80488  174504898
5 China       1999. 212258 1272915272
6 China       2000. 213766 1280428583
```

country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Uniting

- unite() is the inverse of separate().
- it combines multiple columns into a single column.

country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	century	year	rate
Afghanistan	19	99	745 / 19987071
Afghanistan	20	0	2666 / 20595360
Brazil	19	99	37737 / 172006362
Brazil	20	0	80488 / 174504898
China	19	99	212258 / 1272915272
China	20	0	213766 / 1280428583

Create the table6:

```
> #Create tibble called table6
> table6 <- tibble(
+   country = c('Afghanistan', 'Afghanistan', 'Brazil', 'Brazil', 'China', 'China'),
+   century = c(19, 20, 19, 20, 19, 20),
+   year = c(99, 0, 99, 0, 99, 0),
+   rate = c('745/19987071', '2666/20595360', '37737/172006362', '80488/174504898', '212258/
15272', '213766/1280428583')
+ )
> table6
# A tibble: 6 x 4
  country     century  year rate
  <chr>         <dbl> <dbl> <chr>
1 Afghanistan     19.   99. 745/19987071
2 Afghanistan     20.    0. 2666/20595360
3 Brazil          19.   99. 37737/172006362
4 Brazil          20.    0. 80488/174504898
5 China           19.   99. 212258/1272915272
6 China           20.    0. 213766/1280428583
```

Uniting

We can use `unite()` to rejoin the century and year columns.

```
> unite(table6, new, century, year)
# A tibble: 6 x 3
  country     new   rate
  <chr>       <chr> <chr>
1 Afghanistan 19_99 745/19987071
2 Afghanistan 20_0  2666/20595360
3 Brazil      19_99 37737/172006362
4 Brazil      20_0  80488/174504898
5 China       19_99 212258/1272915272
6 China       20_0  213766/1280428583
```

```
> unite(table6, new, century, year, sep = " ")
# A tibble: 6 x 3
  country     new   rate
  <chr>       <chr> <chr>
1 Afghanistan 19 99 745/19987071
2 Afghanistan 20 0  2666/20595360
3 Brazil      19 99 37737/172006362
4 Brazil      20 0  80488/174504898
5 China       19 99 212258/1272915272
6 China       20 0  213766/1280428583
```

Missing values

Missing values

- Changing the representation of a dataset brings up an important subtlety of missing values
- a value can be missing in two possible ways:

1. **Explicitly** via NA (Not Available)
2. **Implicitly** not present in the data

Missing values

Let's create a tibble called stocks:

```
> stocks <- tibble(
+   year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
+   qtr    = c(   1,    2,    3,    4,    2,    3,    4),
+   return = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
+ )
> stocks
# A tibble: 7 x 3
   year   qtr return
  <dbl> <dbl>  <dbl>
1 2015.    1.  1.88
2 2015.    2.  0.590
3 2015.    3.  0.350
4 2015.    4. NA
5 2016.    2.  0.920
6 2016.    3.  0.170
7 2016.    4.  2.66
```

There are two missing values:

- The return for the fourth quarter of 2015 is explicitly missing
- The return for the first quarter of 2016 is implicitly missing

Missing values

The implicit missing value can be explicit by putting years in the columns:

```
> spread(stocks, year, return)
# A tibble: 4 x 3
    qtr `2015` `2016`
  <dbl>  <dbl>  <dbl>
1    1.  1.88  NA
2    2.  0.590  0.920
3    3.  0.350  0.170
4    4. NA      2.66
```

For some cases, explicit values may not be important, you can set `na.rm = TRUE`.

```
> stocks %>%
+   spread(year, return) %>%
+   gather(year, return, `2015`:`2016`, na.rm = TRUE)
# A tibble: 6 x 3
    qtr year   return
* <dbl> <chr>  <dbl>
1    1. 2015   1.88
2    2. 2015   0.590
3    3. 2015   0.350
4    2. 2016   0.920
5    3. 2016   0.170
6    4. 2016   2.66
```

Missing values

Another important tool for making missing values explicit in tidy data is `complete()`:

```
> complete(stocks, year, qtr)
# A tibble: 8 x 3
   year   qtr return
  <dbl> <dbl>  <dbl>
1 2015.    1.  1.88
2 2015.    2.  0.590
3 2015.    3.  0.350
4 2015.    4. NA
5 2016.    1. NA
6 2016.    2.  0.920
7 2016.    3.  0.170
8 2016.    4.  2.66
```

Missing values

fill(): fills missing values in selected columns using the previous entry

```
> treatment <- tribble(
+   ~ person,             ~ treatment, ~response,
+   "Derrick Whitmore", 1,           7,
+   NA,                 2,           10,
+   NA,                 3,           9,
+   "Katherine Burke",  1,           4
+ )
> treatment
# A tibble: 4 x 3
  person           treatment response
  <chr>                <dbl>    <dbl>
1 Derrick Whitmore        1.       7.
2 NA                      2.      10.
3 NA                      3.       9.
4 Katherine Burke         1.       4.
> fill(treatment, person)
# A tibble: 4 x 3
  person           treatment response
  <chr>                <dbl>    <dbl>
1 Derrick Whitmore        1.       7.
2 Derrick Whitmore        2.      10.
3 Derrick Whitmore        3.       9.
4 Katherine Burke         1.       4.
```

Overview

Relational data

Its rare that a data analysis involves only a single table of data

Relational data

Collectively, multiple tables of data.
Relations are always defined between a pair of tables (between 3 or more where each pair has a relation).

Relational data

To work with relational data we need verbs that work with pairs of tables. There are three families of verbs:

1. **Mutating joins**: add new variables to one data frame from matching observations in another
2. **Filtering joins**: filter observations from one data frame based on whether or not they match an observation in the other table
3. **Set operations**: treat observations as if they were set elements

Prerequisites

- `library(tidyverse)` # is discussed and used before.
- `library(dplyr)` # "dplyr" is specialised to do data analysis. It makes common data analysis operations easier.
- `library(nycflights13)` # is used to learn about relational data.

nycflights13

We will use the nycflights13 package to learn about relational data
`airlines`: the full carrier name from its abbreviated code

```
> airlines
# A tibble: 16 x 2
   carrier name
   <chr>   <chr>
 1 9E      Endeavor Air Inc.
 2 AA      American Airlines Inc.
 3 AS      Alaska Airlines Inc.
 4 B6      JetBlue Airways
 5 DL      Delta Air Lines Inc.
 6 EV      ExpressJet Airlines Inc.
 7 F9      Frontier Airlines Inc.
 8 FL      AirTran Airways Corporation
 9 HA      Hawaiian Airlines Inc.
10 MQ      Envoy Air
11 OO      SkyWest Airlines Inc.
12 UA      United Air Lines Inc.
13 US      US Airways Inc.
14 VX      Virgin America
15 WN      Southwest Airlines Co.
16 YV      Mesa Airlines Inc.
```

nycflights13

`airports`: gives information about each airport, identified by the faa airport code

```
> airports
# A tibble: 1,458 x 8
   faa   name                           lat    lon   alt    tz dst   tzone
   <chr> <chr>                        <dbl>  <dbl> <int> <dbl> <chr> <chr>
 1 04G   Lansdowne Airport             41.1  -80.6  1044    -5 A     America/New_York
 2 06A   Moton Field Municipal Ai…     32.5  -85.7   264    -6 A     America/Chicago
 3 06C   Schaumburg Regional           42.0  -88.1   801    -6 A     America/Chicago
 4 06N   Randall Airport               41.4  -74.4   523    -5 A     America/New_York
 5 09J   Jekyll Island Airport         31.1  -81.4    11    -5 A     America/New_York
 6 0A9   Elizabethton Municipal A…     36.4  -82.2  1593    -5 A     America/New_York
 7 0G6   Williams County Airport       41.5  -84.5   730    -5 A     America/New_York
 8 0G7   Finger Lakes Regional Ai…     42.9  -76.8   492    -5 A     America/New_York
 9 0P2   Shoestring Aviation Airf…     39.8  -76.6  1000    -5 U     America/New_York
10 0S9   Jefferson County Intl         48.1 -123.    108    -8 A     America/Los_Ang…
# ... with 1,448 more rows
```

nycflights13

`planes`: gives information about each plane, identified by its tailnum

```
> planes
# A tibble: 3,322 x 9
   tailnum  year type           manufacturer    model   engines seats speed engine
   <chr>   <int> <chr>          <chr>           <chr>     <int> <int> <int> <chr>
 1 N10156   2004 Fixed wing mu… EMBRAER         EMB-14…       2    55    NA Turbo-…
 2 N102UW   1998 Fixed wing mu… AIRBUS INDUST… A320-2…       2   182    NA Turbo-…
 3 N103US   1999 Fixed wing mu… AIRBUS INDUST… A320-2…       2   182    NA Turbo-…
 4 N104UW   1999 Fixed wing mu… AIRBUS INDUST… A320-2…       2   182    NA Turbo-…
 5 N10575   2002 Fixed wing mu… EMBRAER         EMB-14…       2    55    NA Turbo-…
 6 N105UW   1999 Fixed wing mu… AIRBUS INDUST… A320-2…       2   182    NA Turbo-…
 7 N107US   1999 Fixed wing mu… AIRBUS INDUST… A320-2…       2   182    NA Turbo-…
 8 N108UW   1999 Fixed wing mu… AIRBUS INDUST… A320-2…       2   182    NA Turbo-…
 9 N109UW   1999 Fixed wing mu… AIRBUS INDUST… A320-2…       2   182    NA Turbo-…
10 N110UW   1999 Fixed wing mu… AIRBUS INDUST… A320-2…       2   182    NA Turbo-…
# ... with 3,312 more rows
```

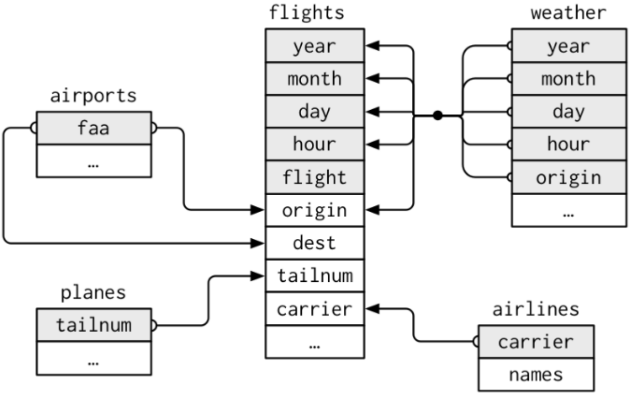
nycflights13

`weather`: gives the weather at each NYC airport for each hour

```
> weather
# A tibble: 26,115 x 15
   origin  year month   day  hour  temp  dewp humid wind_dir wind_speed wind_gust
   <chr>  <dbl> <dbl> <int> <int> <dbl> <dbl> <dbl>    <dbl>      <dbl>     <dbl>
 1 EWR     2013     1     1     1  39.0  26.1  59.4      270      10.4         NA
 2 EWR     2013     1     1     2  39.0  27.0  61.6      250       8.06        NA
 3 EWR     2013     1     1     3  39.0  28.0  64.4      240      11.5         NA
 4 EWR     2013     1     1     4  39.9  28.0  62.2      250      12.7         NA
 5 EWR     2013     1     1     5  39.0  28.0  64.4      260      12.7         NA
 6 EWR     2013     1     1     6  37.9  28.0  67.2      240      11.5         NA
 7 EWR     2013     1     1     7  39.0  28.0  64.4      240      15.0         NA
 8 EWR     2013     1     1     8  39.9  28.0  62.2      250      10.4         NA
 9 EWR     2013     1     1     9  39.9  28.0  62.2      260      15.0         NA
10 EWR     2013     1     1    10  41    28.0  59.6      260      13.8         NA
# ... with 26,105 more rows, and 4 more variables: precip <dbl>, pressure <dbl>,
#   visib <dbl>, time_hour <dttm>
```

nycflights13

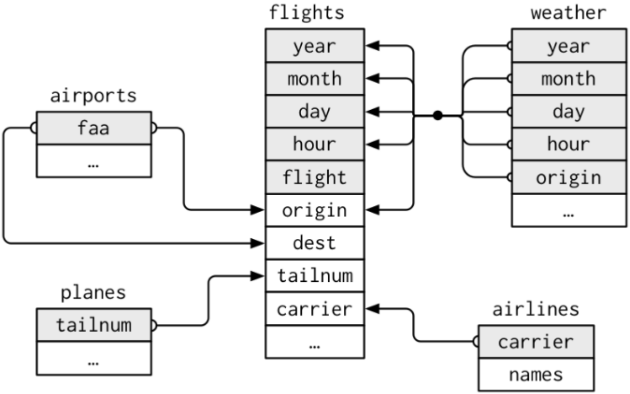
One way to show the relationships between the different tables is with a drawing



`flights` connects to `planes` via a single variable, `tailnum`

nycflights13

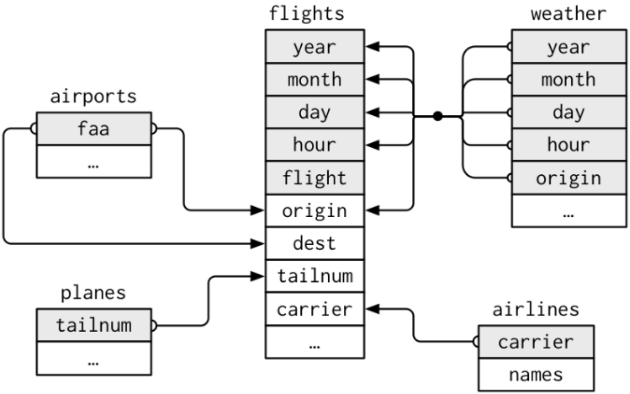
One way to show the relationships between the different tables is with a drawing



`flights` connects to `airlines` through the `carrier` variable

nycflights13

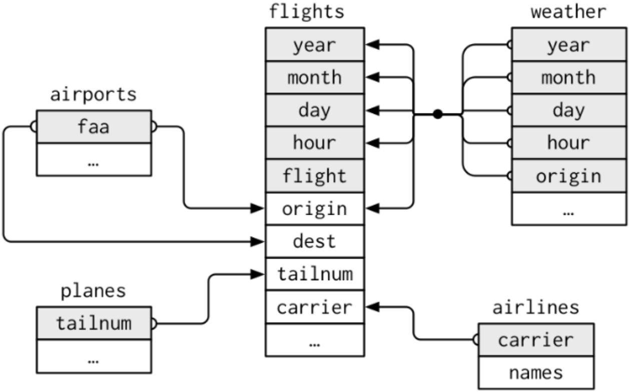
One way to show the relationships between the different tables is with a drawing



`flights` connects to `airports` in two ways: via the `origin` and `dest` variables

nycflights13

One way to show the relationships between the different tables is with a drawing



`flights` connects to `weather` via `origin` (the location), and `year`, `month`, `day` and `hour`

Keys

Keys

- The variables used to *connect* each pair of tables are called keys
- It is a variable (or set of variables) that *uniquely* identifies an observation
- There are two types of keys:

1. **A primary key** uniquely identifies an observation in its own table
   $>$ `planes$tailnum` is a primary key in the "planes" table

2. **A foreign key** uniquely identifies an observation in another table
   $>$ `flights$tailnum` is a foreign key because it appears in the flights table where it matches each flight to a unique plane

A variable can be both a primary key and a foreign key. For example, "origin" is part of the "weather" primary key, and is also a foreign key for the "airport" table.

Keys

To count the primary keys and look for entries where n is greater than one:

```
> planes %>%
+ count( tailnum) %>%
+ filter(n > 1)
# A tibble: 0 x 2
# ... with 2 variables: tailnum <chr>, n <int>
> weather %>%
+   count(year, month, day, hour, origin) %>%
+   filter(n > 1)
# A tibble: 3 x 6
   year month   day  hour origin     n
  <dbl> <dbl> <int> <int> <chr>  <int>
1 2013.   11.     3     1 EWR        2
2 2013.   11.     3     1 JFK        2
3 2013.   11.     3     1 LGA        2
```

Mutating joins

Mutating joins

- It allows to combine variables from two tables
- It matches observations by their keys
- It copies across variables from one table to the other

Mutating joins

- Create a narrow dataset:

```
> flights2 <- flights %>%
+   select(year:day, hour, origin, dest, tailnum, carrier)
> flights2
# A tibble: 336,776 x 8
    year month   day  hour origin dest  tailnum carrier
   <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>
 1  2013     1     1     5 EWR    IAH   N14228  UA
 2  2013     1     1     5 LGA    IAH   N24211  UA
 3  2013     1     1     5 JFK    MIA   N619AA  AA
 4  2013     1     1     5 JFK    BQN   N804JB  B6
 5  2013     1     1     6 LGA    ATL   N668DN  DL
 6  2013     1     1     5 EWR    ORD   N39463  UA
 7  2013     1     1     6 EWR    FLL   N516JB  B6
 8  2013     1     1     6 LGA    IAD   N829AS  EV
 9  2013     1     1     6 JFK    MCO   N593JB  B6
10  2013     1     1     6 LGA    ORD   N3ALAA  AA
# ... with 336,766 more rows
```

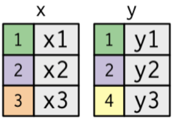
Mutating joins

- Add the full airline name to the flights2 data.
- Combine the `airlines` and `flights2` data frames with `left_join()`

```
> flights2 %>%
+   select(-origin, -dest)  %>%
+   left_join(airlines, by = "carrier")
# A tibble: 336,776 x 7
    year month   day  hour tailnum carrier name
   <int> <int> <int> <dbl> <chr>   <chr>   <chr>
 1  2013     1     1     5 N14228  UA      United Air Lines Inc.
 2  2013     1     1     5 N24211  UA      United Air Lines Inc.
 3  2013     1     1     5 N619AA  AA      American Airlines Inc.
 4  2013     1     1     5 N804JB  B6      JetBlue Airways
 5  2013     1     1     6 N668DN  DL      Delta Air Lines Inc.
 6  2013     1     1     5 N39463  UA      United Air Lines Inc.
 7  2013     1     1     6 N516JB  B6      JetBlue Airways
 8  2013     1     1     6 N829AS  EV      ExpressJet Airlines Inc.
 9  2013     1     1     6 N593JB  B6      JetBlue Airways
10  2013     1     1     6 N3ALAA  AA      American Airlines Inc.
# ... with 336,766 more rows
```

- The result of joining airlines to flights2 is an additional variable: `name`.

Understanding joins

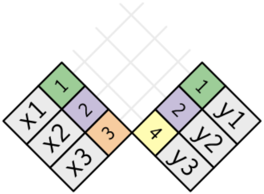



```
> x <- tribble(
+   ~key, ~val_x,
+   1, "x1",
+   2, "x2",
+   3, "x3"
+ )
> x
# A tibble: 3 x 2
    key val_x
  <dbl> <chr>
1     1 x1
2     2 x2
3     3 x3
```

```
> y <- tribble(
+   ~key, ~val_y,
+   1, "y1",
+   2, "y2",
+   4, "y3"
+ )
> y
# A tibble: 3 x 2
    key val_y
  <dbl> <chr>
1     1 y1
2     2 y2
3     4 y3
```

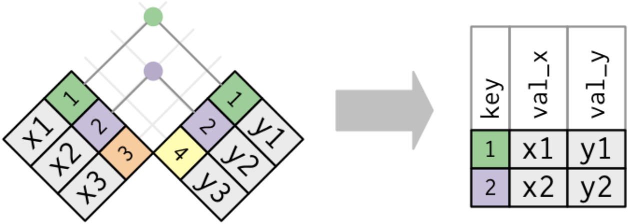
Understanding joins

- The coloured column represents the "key" variable
- The grey column represents the "value" of column that is carried along for the ride
- A join is a way of connecting each row in `x` to zero, one, or more rows in `y`



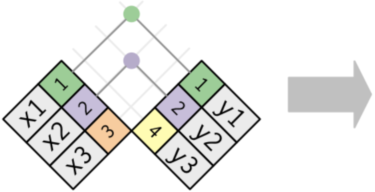
Understanding joins

Notice that we switched the order of the `key` and `value` columns in x to emphasise that joins match based on the **key**.



Inner join

An inner join matches pairs of observations whenever their keys are equal.



key	val_x	val_y
1	x1	y1
2	x2	y2

The output is a new data frame that contains:

- the key,
- the x values,
- the y values.

Inner join

We use by to indicate which variable is the key:

```
> x %>%
+   inner_join(y, by = "key")
# A tibble: 2 x 3
    key val_x val_y
  <dbl> <chr> <chr>
1     1 x1    y1
2     2 x2    y2
```

An important property of an inner join is that **unmatched** rows are **not included** in the result.

Outer joins

outer join

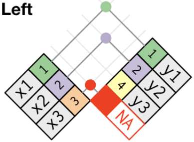
It keeps observations that appear in at least one of the tables.

1. A left join keeps all observations in `x`
2. A right join keeps all observations in `y`
3. A full join keeps all observations in `x` and `y`

These joins work by adding an additional “virtual” observation to each table.

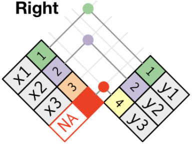
Outer joins

Left



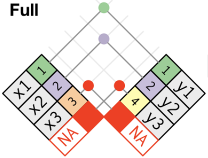
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

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

Other implementations

dplyr	merge
`inner_join(x, y)`	`merge(x, y)`
`left_join(x, y)`	`merge(x, y, all.x = TRUE)`
`right_join(x, y)`	`merge(x, y, all.y = TRUE)`,
`full_join(x, y)`	`merge(x, y, all.x = TRUE, all.y = TRUE)`

References

Hadley Wickham (2014)
Tidy Data
Journal of Statistical Software

Hadley Wickham & Garrett Grolemund (2017)
R for data science: import, tidy, transform, visualize, and model data
O'Reilly.

Roger D. Peng (2015)
R Programming for Data Science

Venables W.N. & Smith D. M. (2018)
An introduction to R: Notes on R - A programming for Data Analysis and Graphics
Version 3.5.1.

Emmanuel Paradis (2005)
R for beginners

Link:
Advanced R
https://adv-r.hadley.nz/

The End