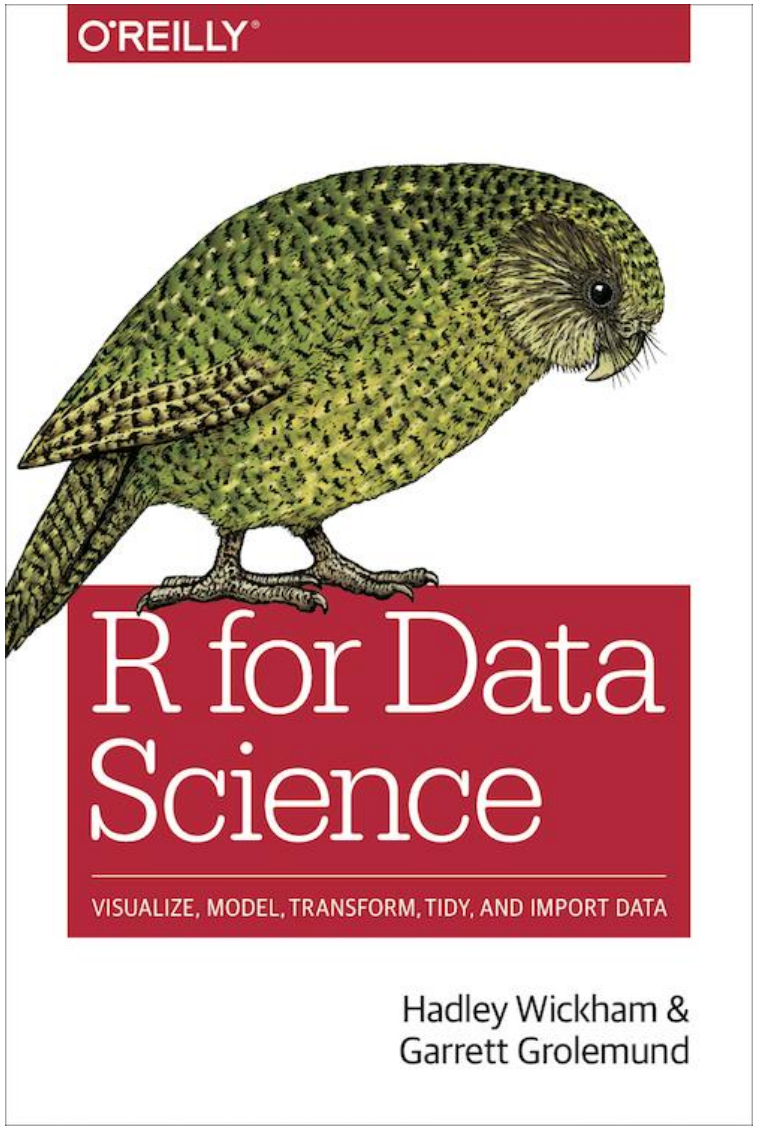
UiT The Arctic University of Norway

FSK-2053 Data science & bioinformatics for fisheries and aquaculture

Practice 1 – Data Wrangling

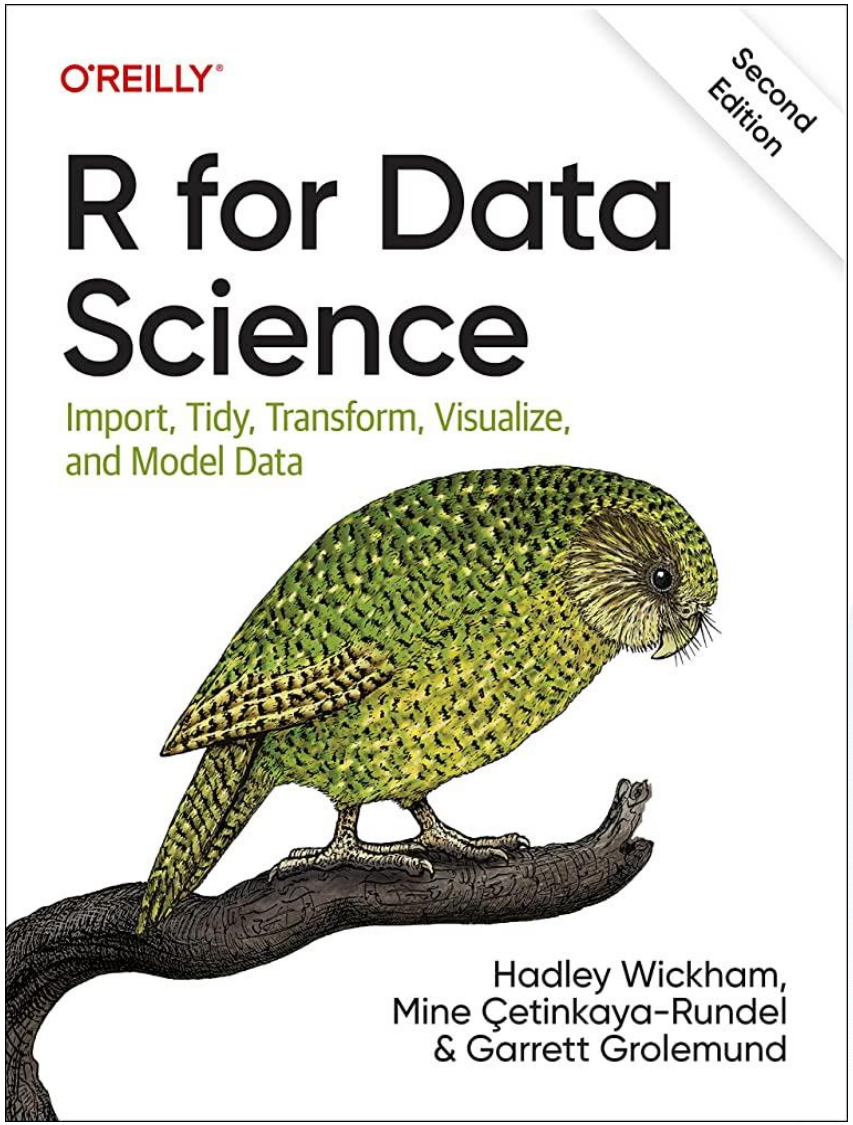
Daniel Kumazawa Morais
daniel.morais@uit.no

https://en.wikipedia.org/wiki/Wrangler_(profession)#/media/File:07350u_The_horse_wrangler.tif



https://r4ds.had.co.nz/index.html
First edition - 2017

https://r4ds.had.co.nz/index.html
First edition - 2017



https://r4ds.hadley.nz/
Second edition - mid 2023

Learning objectives: Practice 1.1

2.1 Loading datasets with readr

- Be able to import data from CSV and Excel files, both local and remote (URL)

- Understand the differences between a tibble and a data.frame

2.2. Tidying-up datasets with tidyr

- Understand the concept of tidy data

- Be able to convert between long and wide formats using pivot functions

  `pivot_longer()`

  `pivot_wider()`

- Be able to use the 4 basic tidyr *verbs*

  `gather()`

  `spread()`

  `separate()`

  `unite()`

- Be able to chain functions using pipes ”%>%”

Learning objectives: Practice 1.2

2.3 Working with different types of data classes and data structures

- Understand the use of the basic data classes and data structures in R
- Use vectorized operations

2.4. Data wrangling with dplyr

- Be able to wrangle data by chaining tidyr and dplyr functions
- Be able to use the six main dplyr one-table verbs:
  - `select()`
  - `filter()`
  - `arrange()`
  - `mutate()`
  - `summarise()`
  - `group_by()`
- Be able to use these additional one-table verbs:
  - `rename()`
  - `distinct()`
  - `count()`
  - `slice()`
  - `pull()`

Learning objectives: Practice 1.3

2.5. Working with two or more related databases with dplyr

- Be able to use the four mutating join verbs:

  `left_join()`

  `right_join()`

  `inner_join()`

  `full_join()`

- Be able to use the two filtering join verbs:

  `semi_join()`

  `anti_join()`

- Be able to use the two binding join verbs:

  `bind_rows()`

  `bind_cols()`

- Be able to use the three set operations:

  `intersect()`

  `union()`

  `setdiff()`

Hands-on practice 1

Commands to focus on:

Basic R:

- getwd() / setwd()
- dir.create()
- list.files()
- file.path()
- file.copy()
- file.exists()
- file.remove()
- head()
- tail()
- class()
- str()
- download.file()
- tempfile() / tempdir()
- scan()
- gsub()
- as.character()
- as.numeric()
- as.integer()

readr:

- read_lines()
- read_csv()
- read_csv2()
- read_tsv()

readxl:

- read_excel()

tidyr:

- pivot_longer() / gather()
- pivot_wider() / spread()
- unite()
- separate()
- drop_na()

dplyr: one table:

- select()
- filter()
- arrange()
- mutate()
- summarise()
- group_by()

- rename()
- transmute()
- distinct()
- count()
- slice()
- pull()
- near()

dplyr: two tables:

- left_join()
- right_join()
- inner_join()
- full_join()
- semi_join()
- anti_join()
- union()
- setdiff()
- bind_rows()
- bind_cols()

ggplot2:

- ggplot()

tidyr

The following five key tidyr functions will allow you to convert most tables to a tidy format (longer format) and return from a tidy table to a wider format whenever necessary:

- **`pivot_longer() / gather()`** Takes values from several columns and gathers them into a single column
- **`pivot_wider() / spread()`** Takes values from a column in a tidy table and spreads them into several columns
- **`unite()`** Unites multiple columns into one by pasting values together
- **`separate()`** Separates a character column into several by a matching separating character or by numeric position
- **`drop_na()`** Removes rows containing missing values, frequently created by the previous operations

Help for tidyverse commands can be found in: https://www.tidyverse.org/

Now we will run
Practice 1.1 script in R-Studio!

Data wrangling

Some of the basic questions about the nature of a dataset are the following:

- How big is it?

- How many attributes are there and what are their characteristics (quantitative or categorical)?

- Can we extract a significantly smaller sub-set to meet our needs?

- How much missing data is there?

- Does it have geospatial attributes?

- Does it have temporal attributes?

- How is it distributed? Does it have extreme outliers? For each categorical attribute how many categories (levels) are there?

- If the data is a set of several tables, how are they related? Can it be characterized as a linked network of variables? If so, which variables will serve as indices or nodes?

Data wrangling is the process of converting data from its raw form into a format that can be ingested by data processing tools. Visualization can be a useful part of data wrangling, especially as a tool in appreciating issues relating to distribution of values. Visualizations of two-dimensional distributions can also be valuable.

Data classes and data structures in R

R has **five** types of atomic classes (types of data):

- Boolean (True/False): type **logical**. E.g., `is.logical(T) → TRUE`
- Integer values: type **integer**. E.g., `is.integer(4.5) → FALSE`
- Real or decimal (continuous numeric) values: type **double**. E.g. `> is.double(4.5) → TRUE`
- Complex numbers: type **complex**. E.g., `is.complex(4+0i) → TRUE`
- Strings of alphanumeric characters: type **character**. E.g., `is.character("45") → TRUE`

Note: you can force a number to be of class integer, by adding "L":

```
is.integer(4) → FALSE
is.integer(4L) → TRUE
```

As any object-oriented language, R has many kinds of data structures. These include:

- vectors
- factors
- lists
- matrices
- data frames
- special classes (these include tibbles, formulas, dates, etc.)

You can learn more about the main types of data structures in R and some useful functions to obtain information from them in this link:

https://resbaz.github.io/2014-r-materials/lessons/01-intro_r/data-structures.html

Vectors and factors

A vector in R is just an ordered list of elements of the same class. They commonly appear separated by commas. We can use the c() operator to enumerate the elements of a vector:

```
names <- c("Ragnar", "Bjørn", "Sigurd", "Hvirserk", "Ivar")
ages <- c(45, 32, 17, 25, 19)
heights <- c(1.98, 1.72, 1.67, 1.84, 1.75)
```

Factors are like vectors, but they can only have a fixed and known set of possible values (levels). R uses factors to handle categorical variables.

```
names_fact <- factor(c("Ragnar", "Bjørn", "Sigurd", "Hvirserk", "Ivar")) names_fact

   [1] Ragnar   Bjørn    Sigurd   Hvirserk Ivar
   Levels: Bjørn Hvirserk Ivar Ragnar Sigurd
```

A common beginner's error is trying to change a value in a factor to an undefined level. This will generate an NA value and a warning message:

```
names_fact[2] <- "Ironside“
        Warning message:
        In `[<-.factor`(`*tmp*`, 2, value = "Ironside") : invalid factor level, NA
        generated
names_fact
        [1] Ragnar   <NA>     Sigurd   Hvirserk Ivar
        Levels: Bjørn Hvirserk Ivar Ragnar Sigurd
```

Quantitative and categorical variables

It is important to consider the quantitative or categorical character of a variable. Vectors can be either quantitative or categorical, factors can be only categorical. This distinction is crucial when deciding which statistical models are applicable or which kinds of graphical visualizations will be most suitable. Numerical variables are represented in axes. Categorical variables can be represented by different graphic aesthetic attributes in a graph (colours, shapes, sizes, etc.)

Numerical vectors are always quantitative by nature. However, sometimes they can be categorized (e.g. years can be considered as fixed levels of a factor).

Categorical variables (usually in the form of factors) can be either unordered or ordered by their nature. Ordered factors are sometimes referred to as "semi-quantitative". E.g.:

```
sex <- factor(c("male", "male", "female", "female", "male", "male"))
abundances <- factor(c("rare","occasional","frequent","common","abundant"))
```

By default, the order of the levels will be alphabetical. If we want to use a different order, we have to state that when defining the factor. E.g.:

```
abundances <- factor(c("rare","rare","frequent","abundant","frequent", "rare"),
levels=c("rare","occasional","frequent","common","abundant"), ordered = TRUE)
```

Reordering character vectors is helpful to improve display of axes and legends in the right order. **forcats** package of the Tidyverse provides a suite of tools that solve common problems with factors, including changing the order of levels or the values.

Comparisons in R

Errors due to misunderstanding data classes or to using wrong data structures are a common cause of frustation among R beginners. It is important to get familiar with these concepts, specially when using comparison conditions. Sometimes the result of a particular comparison in R can be unexpected. Some examples are:

```
“45” == 45 → TRUE
“45.0” == 45 → FALSE
(1/49)*49 == 1 → FALSE
49/49 == 1 → TRUE
1/10^200 == 0  → FALSE
1/10^500 == 0  → TRUE
```

We can use function **dplyr::near()** to avoid most of these confusing results when using numerical values. This function uses a practical (customizable) value for the numeric tolerance (minimum gap to consider two values as different). By default this is 1.5 e-8.

```
near(1/10^200, 0)  → TRUE
near((1/49)*49, 1) → TRUE
near(0.0000050, 0.0000051) → FALSE
near(0.000000050, 0.000000051) → TRUE
```

However, it will not work when comparing a character to a numerical value:

```
near(“45”,45)  → ERROR! non-numeric argument to binary operator
near(as.numeric("45.0"),45) → TRUE
```

Comparison operators in R

There are six basic comparison operators in R:

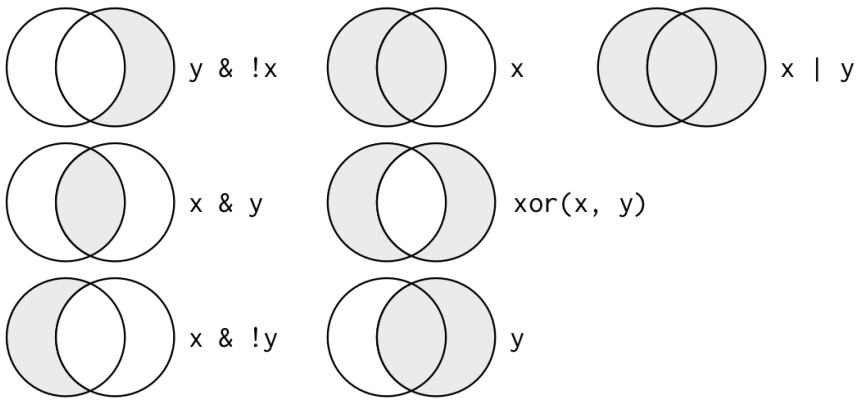
>	(greater than)
>=	(greater than or equal to)
<	(less than)
<=	(less than or equal to)
!=	(not equal)
==	(equal) Note that using a single equal (=) in a comparison is a common beginner's error in R

Note that the first four operators are quantitative and will work only with numerical values or levels of ordered (semiquantitative) factors. They will not work with unordered categorical factors. Note also that logical (FALSE/TRUE) values are sometimes handled as numerical (0/1) values. Thus (TRUE > FALSE) will be TRUE.

Conversely, identity comparisons ==, !=, near(), identical() will work with all classes of values.

Comparisons can be combined to form complex filtering conditions, using the logical (also known as Boolean) operators:

&	(AND operator, intersection)
|	(OR operator, union)
!	(NOT operator, complement)
xor	(Exclusive-OR operator)



Vectorized operations in R (1)

One of the most powerful features of R is the ability to operate with vectors. Typically, an unit operation will be applied equally to all elements of a vector. And an operation between two (or more) vectors will be automatically done for each element of the vectors in an ordered way. E.g.:

```
x <- c(2,6,8,12)
y <- c(2,5,10,20)

x * 3   Result:  6 18 24 36
x + y   Result:  4 11 18 32
x * y   Result: 4 30 80 240
```

This is also true for any class of vectors. E.g. with a character vector:

```
x <- c("Ivar", "Bjørn", "Hvitserk")
paste(x, "Ragnarson")

Result:  "Ivar Ragnarson"    "Bjørn Ragnarson"    "Hvitserk Ragnarson"
```

Vectorized operations are crucial when working with big datasets in R. Existing variables (vectors) can be modified or new variables can be generated from existing ones, by using relatively simple operations and understandable syntax.

In many other programming languages, such operations would need to use complex loops or iterations.

Vectorized operations in R (2)

Many functions and operations can be applied to whole datasets or matrices. E.g.

```
char_dataset <- as.character(dataset)
```

This includes comparisons. When a comparison is applied between an atomic value and a dataset, each element of the dataset will be compared to the atomic value, and the result will be a structure of booleans of the same dimensions than the dataset.

```
X <- data.frame(
  id = c( 1,   2,   1,   2,   1,   2),
  dv = c("A", "B", "C", "D", "A", "B"),
  pt = c( 1,   1,   1,   4,   2,   1)
)

> X==1

        id    dv    pt
[1,]  TRUE FALSE  TRUE
[2,] FALSE FALSE  TRUE
[3,]  TRUE FALSE  TRUE
[4,] FALSE FALSE FALSE
[5,]  TRUE FALSE FALSE
[6,] FALSE FALSE  TRUE
```

Operators && (global AND) and II (global OR) can be useful for comparisons involving matrices:

```
X==1 | X==2
        id    dv    pt
[1,] TRUE FALSE  TRUE
[2,] TRUE FALSE  TRUE
[3,] TRUE FALSE  TRUE
[4,] TRUE FALSE FALSE
[5,] TRUE FALSE  TRUE
[6,] TRUE FALSE  TRUE

X==1 || X==2
TRUE
```

```
X==1 & X==2
        id    dv    pt
[1,] FALSE FALSE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
[5,] FALSE FALSE FALSE
[6,] FALSE FALSE FALSE

X==1 && X==2
FALSE
```

dplyr

The following six key dplyr functions will allow you to solve the vast majority of your data manipulation challenges on a single dataset:

- **filter()** Pick observations (rows) by their values .

- **arrange()** Reorder the rows by their values.

- **select()** Pick variables by their names.

- **mutate()** Create new variables with functions of existing variables.

- **summarise()** Collapse many values down to a single summary.

All of these can be used in conjunction with **group_by()** which changes the scope of each function from operating on the entire dataset to operating on it group-by-group.

These six functions provide the verbs for a language of data manipulation.

Help for tidyverse commands can be found in: https://www.tidyverse.org/

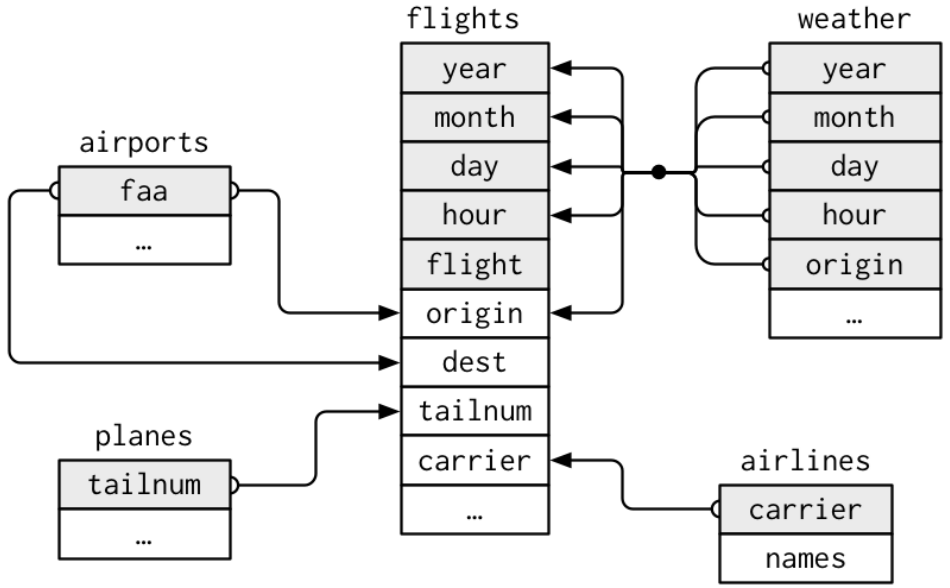
Now we will run
Practice 1.2 script in R-Studio!

Working with several data tables: relational data

It's rare that a data analysis involves only a single table of data. Typically, you have many tables of data and you must combine them to answer the questions that you're interested in. Collectively, multiple tables of data are called relational data because it is the relations, not just the individual datasets, that are important.

One example is a database of occurrences of flights (one table with one row for each flight), which is related to four other different tables: airports, planes, weather, and airlines.

Relations between the tables are those **variables** whose information can be found in two or more tables. These can be represented as arrows:



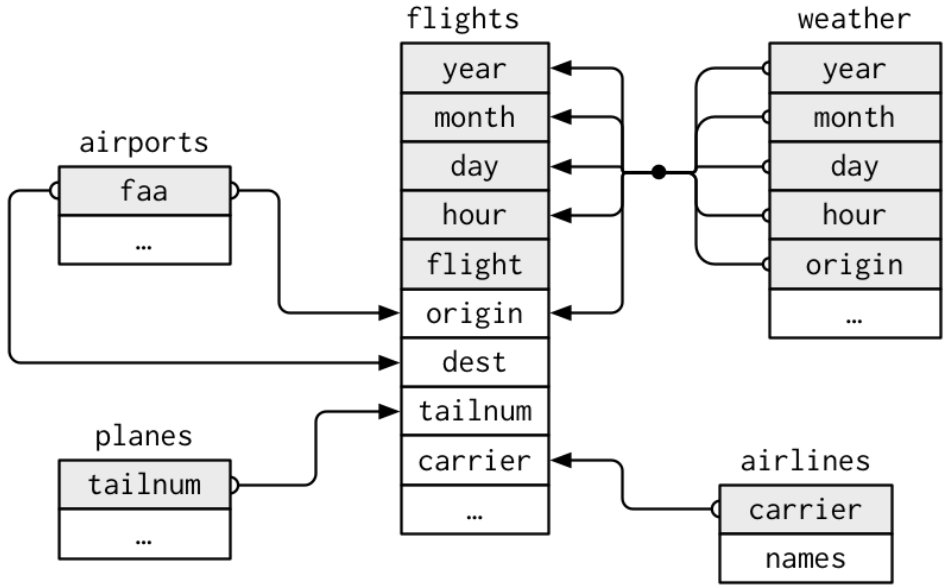
More info in: https://r4ds.had.co.nz/relational-data.html

Working with several data tables: relational data

It's rare that a data analysis involves only a single table of data. Typically, you have many tables of data and you must combine them to answer the questions that you're interested in. Collectively, multiple tables of data are called relational data because it is the relations, not just the individual datasets, that are important.

One example is a database of occurrences of flights (one table with one row for each flight), which is related to four other different tables: airports, planes, weather, and airlines.

Relations between the tables are those **variables** whose information can be found in two or more tables. These can be represented as arrows:



What are variables when we are using tidy data?

More info in: https://r4ds.had.co.nz/relational-data.html

Keys

Variables which can be used to connect a pair of tables are called **keys**.

A key is a **variable** (or set of **variables**) that uniquely identifies observations.

In simple cases, a single **variable** is sufficient to identify an observation. For example, each plane is uniquely identified by its *tailnum*. In other cases, multiple variables may be needed. For example, to identify an observation in weather you need five variables: year, month, day, hour, and origin.

There are two types of keys:

- A **primary key uniquely identifies an observation in its own table**. For example, *planes$tailnum* is a primary key because it uniquely identifies each plane in the planes table.

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

Once you've identified the primary keys in your tables, it's good practice to verify that they do indeed uniquely identify each observation. One way to do that is to `count()` the primary keys and look for entries where *n* is greater than one.

If a table lacks a unique primary key, it's useful to add one with mutate() and row_number(). That makes it easier to match observations if you've done some filtering and want to check back in with the original data. This is called a **surrogate key**.

Relations and operations with two tables

A **primary key** and the corresponding **foreign key** in another table form a **relation**.

Relations are typically one-to-many. For example, each flight has one plane, but each plane has many flights. In other data, you'll occasionally see a 1-to-1 relationship. You can think of this as a special case of 1-to-many.

You can model many-to-many relations with a many-to-1 relation plus a 1-to-many relation. For example, there's a many-to-many relationship between airlines and airports: each airline flies to many airports; each airport hosts many airlines.

To work with relational data, you need functions that work with pairs of tables. There are three families of *verbs* designed to work with relational data:

- **Mutating joins**, which add new **variables** to one data frame from matching observations in another.

- **Filtering joins**, which filter observations from one data frame based on whether they match an observation in the other table.

- **Set operations**, which treat observations as if they were set elements.

Mutating joins

An **inner join** keeps observations that appear in both tables. The output of an inner join is a new data frame that contains the key, the x values, and the y values.

After an inner join, unmatched rows are not included in the result. This means that generally inner joins are usually not appropriate for use in analysis because it's too easy to lose observations.

An **outer join** keeps observations that appear in at least one of the tables. There are three types of outer joins:

A **left join** keeps all observations in x.

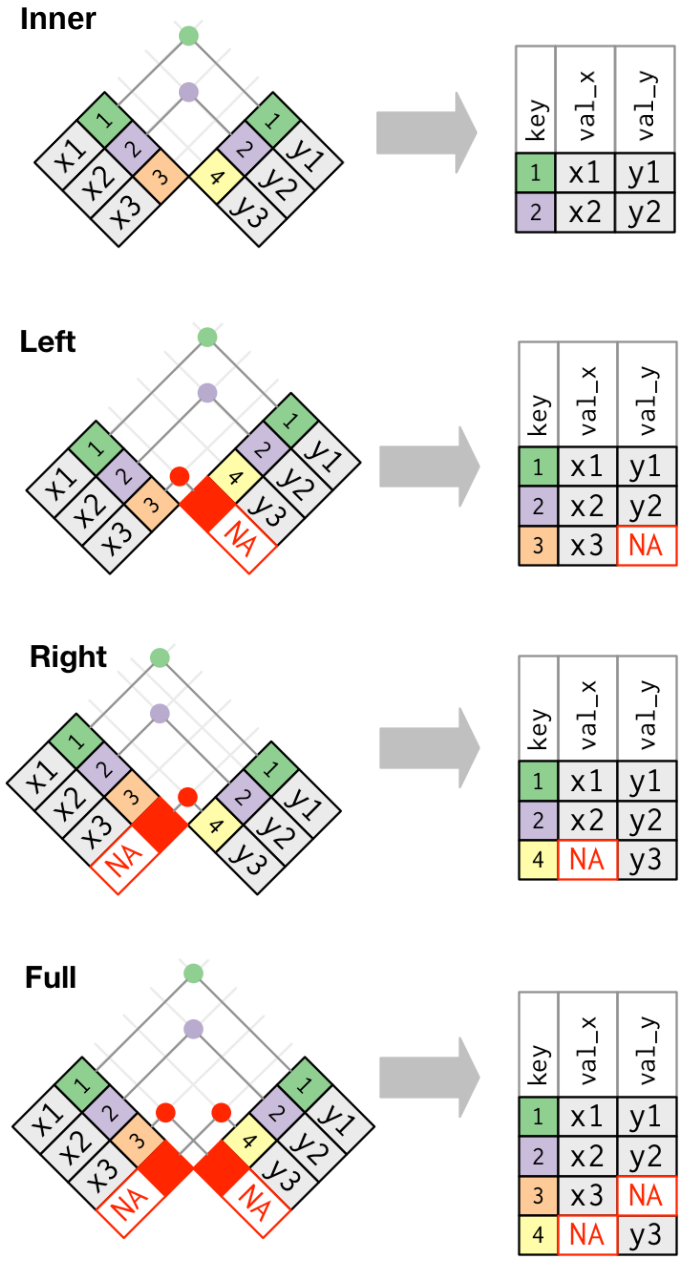
A **right join** keeps all observations in y.

A **full join** keeps all observations in x and y.

After an outer join, unmatched observations will be filled with *NA* values.

The most used join is the left join: because it preserves all observations in the original table, even when there isn't a match in the second table.

The left join should be your default join, when you want to look up additional data from another table.

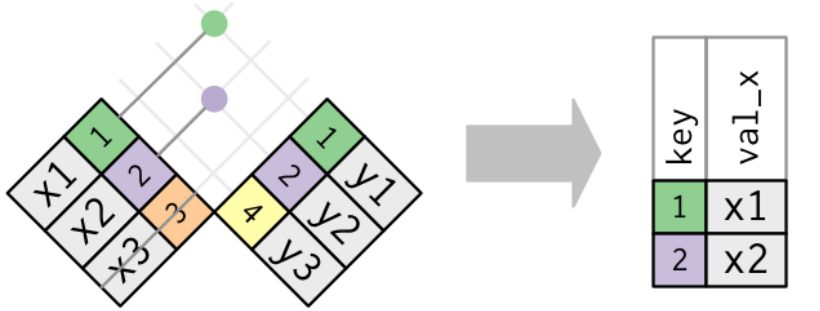


Filtering joins

Filtering joins match observations in the same way as mutating joins, but affect the observations, not the variables. There are two types:

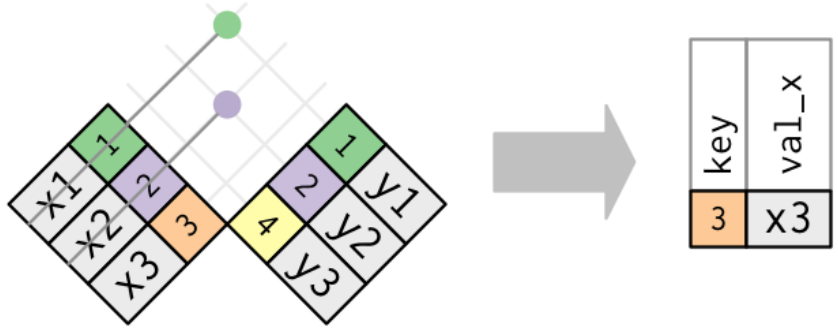
A semi_join keeps all observations in x that have a match in y.

Semi-joins are useful for matching filtered summary tables back to the original rows.



An anti_join drops all observations in x that have a match in y.

Anti-joins are useful for diagnosing join mismatches, to track which observations in a table do not have a match in the second table.



Set operations

Set operations work with complete rows, comparing the values of every variable in the tables. These operations expect the x and y input tables to have exactly the same variables, and treat the observations like sets:

intersect(x,y): returns only observations present in both x and y.

union(x,y): returns unique observations present in x or y.

setdiff(x,y): returns observations present in x, but not in y.

Direct binding of data tables (not recommended)

bind_rows(x,y) will simply bind the observations (rows) of two (or more) databases together into a single table. Columns will be matched by name, and any missing columns will be filled with *NA*. This operation could lead to duplicated rows (if the same observations are present in both tables), which can be removed later by using **distinct()**.
If the tables have the exact same columns, you can use **union(x,y)** to avoid duplicate rows.

The equivalent function for columns, **bind_cols(x,y)** is a dangerous operation which can easily lead to corrupted data. When column-binding, rows are matched by position, so all data frames must have the same number of rows and their observations must be in the same order. It is highly recommended to use some kind of mutating-join instead of bind_cols() for this operation. The only advantage over a left join is when the tables don’t have any key column or IDs to join by and you have to rely solely on their order.

Now we will run
Practice 1.3 script in R-Studio!