Data handling using the Tidyverse

Völundur Hafstað, Deborah Figueiredo Nacer de Oliveira

Contents

Introduction

In this section of the course we will be exploring many of the core functions of the tidyverse. The tidyverse is a collection of packages that are designed for data science. They are designed to work very well with each other and together form an ecosystem that is ideal for analyzing most forms of data. Each of the functions we will cover is relatively simple, but when used together are capable of very fast, efficient and readable data analysis. We will be working with sample data sets that we have provided in the course material.

This file is an R Markdown Notebook. It is a very convenient file format when you are writing lots of text alongside your R code, for example when writing reports. In an R markdown file you can insert code "chunks" and run them individually. They will appear in the document as gray text boxes. The output of that code is then output directly underneath the code chunk.

We start by loading in the tidyverse suite of packages using the `library()` function:

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.7     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Libraries are generally loaded at the very beginning of the file, so that you and other people can see at a glance what packages you are using. In this section of the course we will only load the tidyverse library, which is a collection of several packages in one.

Reading in data

We have provided several files containing data in the course material. Make sure to set your working directory to the folder you have downloaded these files to. In my case they are located in a folder called "r_course", in a sub-folder called "data".

```
setwd("/home/v/projects/r_course/")
```

Now R knows where to look for any files that you want to read in. To actually open them in R we will use `read_tsv()`. Since we have already set a working directory, we do not need to give R the full path to the file, just the relative path from the working directory.

The first data set we will work with is a list of all domestic flights departing from three New York airports in 2013. When we read in data, we want to assign it to a variable name that makes it obvious what we are working with. In this case a simple name like "flights" will do nicely.

```
flights <- read_tsv("data/nycflights13_flights.txt")
```

```
## Rows: 336776 Columns: 19
## -- Column specification --------------------------------------------------------
## Delimiter: "\t"
## chr   (4): carrier, tailnum, origin, dest
## dbl  (14): year, month, day, dep_time, sched_dep_time, dep_delay, arr_time, ...
## dttm  (1): time_hour
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

When we read in data with `read_tsv()`, it gives some useful information about the file we have just read in, although this can be disabled. Here we can see that the file we read in has around 336,000 rows of data and 19 columns. `read_tsv()` also tries to guess the type of each column, i.e. whether it is a character, numeric, logical etc.

To get a better idea of what the data looks like we can use the functions `head()` and `tail()` to see the first and last rows of the data.

```
head(flights)
```

```
## # A tibble: 6 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <dbl> <dbl> <dbl>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## 4  2013     1     1      544            545        -1     1004           1022
## 5  2013     1     1      554            600        -6      812            837
## 6  2013     1     1      554            558        -4      740            728
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <dbl>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

```
tail(flights)
```

```
## # A tibble: 6 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <dbl> <dbl> <dbl>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
## 1  2013     9    30       NA           1842        NA       NA           2019
## 2  2013     9    30       NA           1455        NA       NA           1634
## 3  2013     9    30       NA           2200        NA       NA           2312
## 4  2013     9    30       NA           1210        NA       NA           1330
## 5  2013     9    30       NA           1159        NA       NA           1344
## 6  2013     9    30       NA            840        NA       NA           1020
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <dbl>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Another great function to quickly see what you are working with is `glimpse()`.

```
glimpse(flights)
```

```
## Rows: 336,776
## Columns: 19
## $ year           <dbl> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2~
## $ month          <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ day            <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ dep_time       <dbl> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ~
## $ sched_dep_time <dbl> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ~
## $ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -1~
## $ arr_time       <dbl> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,~
## $ sched_arr_time <dbl> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,~
## $ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -1~
## $ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
## $ flight         <dbl> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
## $ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
## $ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",~
## $ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",~
## $ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
## $ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
## $ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6~
## $ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
## $ time_hour      <dttm> 2013-01-01 10:00:00, 2013-01-01 10:00:00, 2013-01-01 1~
```

`glimpse()` shows you every column in the data frame, which can be convenient if you are working with many columns of data.

Data wrangling

One of the most important aspects of working with large data sets is to be able to easily find the exact data that you need.

For example, let's say that we are only interested in how far the average flight from NYC is. The flights data frame contains much more information than that, but the only column of data we are interested in is "distance".

We might also only be interested in flights departing from JFK airport. The flights data contains information from three different airports in an approximately even ratio, so in this case about two-thirds of the rows are not relevant to us.

This section will show you three simple functions that you can use to extract just the data that you need.

filter()

`filter()` is a function that subsets rows, keeping only those that we are interested in. It is similar `which()` that was covered in the first part of the course, but the syntax for `filter()` is quite a bit simpler. We can tell the function what rows we are interested in by giving it a "condition" to filter by. For example to filter flights that originate from JFK we would use:

```
filter(flights, origin == 'JFK')
```

```
## # A tibble: 111,279 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <dbl> <dbl> <dbl>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
```

```
## 1 2013 1 1 542 540 2 923 850
## 2 2013 1 1 544 545 -1 1004 1022
## 3 2013 1 1 557 600 -3 838 846
## 4 2013 1 1 558 600 -2 849 851
## 5 2013 1 1 558 600 -2 853 856
## 6 2013 1 1 558 600 -2 924 917
## 7 2013 1 1 559 559 0 702 706
## 8 2013 1 1 606 610 -4 837 845
## 9 2013 1 1 611 600 11 945 931
## 10 2013 1 1 613 610 3 925 921
## # ... with 111,269 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <dbl>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Here we supplied the `filter()` function with two things. The first is the data frame "flights". This tells `filter()` on what data frame we want to apply the filter. The next is the filter condition: `origin == 'JFK'`. "origin" is the name of the column in the data frame that we are using to filter the data, and 'JFK' is what needs to be contained in that particular cell in order to be kept. Note that when working with strings you need to wrap it inside single or double quotes.´ The double equal sign "==" is a comparison operator and will return either true or false. Note that this is **not** the same as "=", which can be used to assign variables. One of the most common sources of errors when you are beginning to learn R is to write "=" when you meant to write "=="!. For more information on conditionals see the R_building_blocks.pdf file.

The `filter()` function can be supplied with more than one conditional. For example if we want to find flights between JFK and Miami airport (MIA) that departed between 6 and 8 in the morning, we can string all of those together using the "&" (and) and "|" (or) symbols.

```
filter(flights, origin == "JFK" & dest == "MIA" & dep_time > 600 & dep_time < 800)
```

```
## # A tibble: 574 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <dbl> <dbl> <dbl>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
##  1  2013     1     1      759            800        -1     1057           1127
##  2  2013     1     2      707            715        -8     1022           1045
##  3  2013     1     2      757            800        -3     1058           1127
##  4  2013     1     3      710            715        -5     1042           1045
##  5  2013     1     3      757            800        -3     1109           1124
##  6  2013     1     4      712            715        -3     1022           1045
##  7  2013     1     4      756            800        -4     1109           1124
##  8  2013     1     5      710            715        -5     1024           1045
##  9  2013     1     5      754            800        -6     1101           1124
## 10  2013     1     6      708            715        -7     1057           1045
## # ... with 564 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <dbl>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Exercise 1

1. How many flights departed JFK in February?
2. How many flights departed LGA airport in June and July combined?

select()

The `select()` function is similar to `filter()`, but it subsets columns instead of rows.

```
select(flights, tailnum, air_time, distance)
```

```
## # A tibble: 336,776 x 3
##    tailnum air_time distance
##    <chr>      <dbl>    <dbl>
##  1 N14228       227     1400
##  2 N24211       227     1416
##  3 N619AA       160     1089
##  4 N804JB       183     1576
##  5 N668DN       116      762
##  6 N39463       150      719
##  7 N516JB       158     1065
##  8 N829AS        53      229
##  9 N593JB       140      944
## 10 N3ALAA       138      733
## # ... with 336,766 more rows
```

The first argument we supply to `select()` is the name of the data frame, all subsequent arguments are the names of the columns we want to keep. The order of the columns in the output will be the order you supply them to the function.

`select()` is quite flexible in how you specify the column names. You can give it the column number and/or the column name, and the column names do not have to be in quotes:

```
select(flights, 1:3, 7, origin, "dep_delay", arr_delay)
```

```
## # A tibble: 336,776 x 7
##     year month   day arr_time origin dep_delay arr_delay
##    <dbl> <dbl> <dbl>    <dbl> <chr>      <dbl>     <dbl>
##  1  2013     1     1      830 EWR            2        11
##  2  2013     1     1      850 LGA            4        20
##  3  2013     1     1      923 JFK            2        33
##  4  2013     1     1     1004 JFK           -1       -18
##  5  2013     1     1      812 LGA           -6       -25
##  6  2013     1     1      740 EWR           -4        12
##  7  2013     1     1      913 EWR           -5        19
##  8  2013     1     1      709 LGA           -3       -14
##  9  2013     1     1      838 JFK           -3        -8
## 10  2013     1     1      753 LGA           -2         8
## # ... with 336,766 more rows
```

Note that this is quite ugly code, especially the mix of quoted and unquoted column names!

You can also select columns that are in a character vector:

```
a <- c("origin", "dest")
select(flights, a)
```

```
## Note: Using an external vector in selections is ambiguous.
## i Use ‘all_of(a)‘ instead of ‘a‘ to silence this message.
## i See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.
## This message is displayed once per session.
```

```
## # A tibble: 336,776 x 2
##    origin dest
##    <chr>  <chr>
##  1 EWR    IAH
##  2 LGA    IAH
##  3 JFK    MIA
##  4 JFK    BQN
##  5 LGA    ATL
##  6 EWR    ORD
##  7 EWR    FLL
##  8 LGA    IAD
##  9 JFK    MCO
## 10 LGA    ORD
## # ... with 336,766 more rows
```

Be careful with this as you will get an error if there are strings in the vector that are not column names in the data frame.

mutate()

Often what we are interested in is not explicitly given in our data but can be calculated using values from several columns. We can create new columns in our data frame using the `mutate()` function. For example, let's say that we are interested in how fast an airplane was going. We can calculate the speed using information from the "distance" and "air_time" columns and store it in a new column "speed":

```
mutate(flights, speed = distance / air_time)
```

```
## # A tibble: 336,776 x 20
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <dbl> <dbl> <dbl>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 12 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <dbl>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
## #   speed <dbl>
```

The resulting data frame is now 20 columns instead of 19, with the new column "speed" positioned last.

Code styles and the pipe

The true power of the tidyverse functions comes when we string them together to perform complex analyses using many relatively simple steps.

There are many ways to write code. Everyone has their own preferred way to write and structure code in a way that makes sense to them. It is worth keeping in mind that your code should be readable by other people, and when you start working on complicated projects with other people this becomes absolutely essential.

The "programmer" way - if you have used python or a similar programming language this might feel natural:

```
x <- filter(flights, origin == "JFK")
x <- select(x, tailnum, air_time, distance)
x <- mutate(x, speed = distance / air_time)
x
```

Here we assigned the new data frame and its intermediate steps into the variable "x". In each line of code we overwrite x by applying a new function to it, modifying it in some way. This is a good way to write your code as it makes it fairly readable for other people. A good rule of thumb is to only do "one thing" in each line of code (in this case `filter()`, `select()`, `mutate()`). The downside of this style is it does get a bit redundant always having to type `x <- ...`.

R doesn't care if your code is readable or not as long as it executes without errors. If you wanted to you could get exactly the same output using a single line of code using the "unreadable" way:

```
y <-
  mutate(select(filter(flights, origin == "JFK"), tailnum, air_time, distance),
    speed = distance / air_time) # complicated and unreadable code!
y
```

To R this is exactly the same block of code. In fact it is probably a bit more efficient since it does not have to store the intermediate steps! However this is completely unreadable for other people and you should avoid writing code like this as much as you can. In cases where you are forced to write out code like this it is important to comment the code so that people know at a glance what it is you have done.

We will be using a bit of a different way to write our code, using something known as the pipe. The pipe is written `%>%` (keyboard shortcut ctrl+shift+m or command+shift+m) and is read as *and then*. It replaces the first argument of a function on its right with whatever output comes from the code on the left. For example

```
flights %>%                # take the "flights" data frame, and then
  filter(origin == "JFK")  # apply the filter() function
```

is the same thing as writing:

```
filter(flights, origin == "JFK")
```

You would read the code as taking the flights data *and then* applying the `filter()` function. Notice that when we use the pipe we do not have to type in "flights" inside `filter()` anymore, only the filter conditions. This makes it very easy to read other people's code, as it is executed in the same order as we read it.

The beauty of the pipe is that we can string together multiple functions like this, to make a chain of pipes that is as long as we need. So the "tidyverse" way using pipes looks like:

```
z <- flights %>%                                       # take the flight data, and then
  filter(origin == "JFK") %>%                          # filter that data, and then
  select(year, month, day, tailnum, air_time, distance) %>%  # select the columns we need, and then
  mutate(speed = distance / air_time)                  # mutate to create a new column
z
```

```
## # A tibble: 111,279 x 7
##     year month   day tailnum air_time distance speed
##    <dbl> <dbl> <dbl> <chr>      <dbl>    <dbl> <dbl>
##  1  2013     1     1 N619AA       160     1089  6.81
##  2  2013     1     1 N804JB       183     1576  8.61
##  3  2013     1     1 N593JB       140      944  6.74
##  4  2013     1     1 N793JB       149     1028  6.90
##  5  2013     1     1 N657JB       158     1005  6.36
##  6  2013     1     1 N29129       345     2475  7.17
##  7  2013     1     1 N708JB        44      187  4.25
##  8  2013     1     1 N3739P       128      760  5.94
##  9  2013     1     1 N532UA       366     2586  7.07
## 10  2013     1     1 N635JB       175     1074  6.14
## # ... with 111,269 more rows
```

In the end this looks a lot like the "programmer" style of writing code, but without the redundancy of assigning the intermediate steps or specifying what data frame it is that we want to use to each function.

Exercise 2

Look at the data set "diamonds" that comes built-in to the ggplot2 package (part of the tidyverse suite):

1. Find the diamond with the highest price/volume ratio - how many carats is it? Try to write your code using pipes! Hint: use the arrange() function to sort the data. Check out the documentation by typing `?arrange()`.

```
diamonds
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

Summarising data

In this section we will cover a couple of functions that are great to use to generate summary statistics - these are often the first steps when it comes to analyzing any type of numerical data.

One of the simplest ways of summarising data is by counting the number of observations based on some criteria. Let's say we want to find out how many flights have departed from New York each month. There are several ways to do this:

```r
flights %>%
  group_by(month) %>%
  summarise(n = n())
```

```
## # A tibble: 12 x 2
##    month     n
##    <dbl> <int>
##  1     1 27004
##  2     2 24951
##  3     3 28834
##  4     4 28330
##  5     5 28796
##  6     6 28243
##  7     7 29425
##  8     8 29327
##  9     9 27574
## 10    10 28889
## 11    11 27268
## 12    12 28135
```

```r
flights %>%
  group_by(month) %>%
  tally()
```

```
## # A tibble: 12 x 2
##    month     n
##    <dbl> <int>
##  1     1 27004
##  2     2 24951
##  3     3 28834
##  4     4 28330
##  5     5 28796
##  6     6 28243
##  7     7 29425
##  8     8 29327
##  9     9 27574
## 10    10 28889
## 11    11 27268
## 12    12 28135
```

```r
flights %>%
  count(month)
```

```
## # A tibble: 12 x 2
##    month     n
##    <dbl> <int>
##  1     1 27004
##  2     2 24951
##  3     3 28834
##  4     4 28330
##  5     5 28796
##  6     6 28243
```

```
##  7      7 29425
##  8      8 29327
##  9      9 27574
## 10     10 28889
## 11     11 27268
## 12     12 28135
```

Personally I like to use either the first or last options here. The `summarise()` function is very powerful and can generate more summary statistics for you, such as mean, median, min, max and standard deviation. For an overview of what summary functions can be used inside `summarise()` please see the dplyr cheat sheet.

Let's make a table that shows the number of outgoing flights with some other summary statistics:

```
flights %>%
  group_by(month) %>%
  summarise(n = n(),
            mean_dep_delay = mean(dep_delay),
            min_dep_delay = min(dep_delay),
            max_dep_delay = max(dep_delay))
```

```
## # A tibble: 12 x 5
##    month     n mean_dep_delay min_dep_delay max_dep_delay
##    <dbl> <int>          <dbl>         <dbl>         <dbl>
##  1     1 27004             NA            NA            NA
##  2     2 24951             NA            NA            NA
##  3     3 28834             NA            NA            NA
##  4     4 28330             NA            NA            NA
##  5     5 28796             NA            NA            NA
##  6     6 28243             NA            NA            NA
##  7     7 29425             NA            NA            NA
##  8     8 29327             NA            NA            NA
##  9     9 27574             NA            NA            NA
## 10    10 28889             NA            NA            NA
## 11    11 27268             NA            NA            NA
## 12    12 28135             NA            NA            NA
```

There is no limit to how many summary statistics you can generate inside a single `summarise()`, as long as they are separated by a comma. Here I have also split each summary into a separate line in the code to improve readability - a good practice!

Note however that there is something strange going on with the new columns - their values are all NA. This means that there is at least one "dep_delay" observation in the original data that has an NA value. We need to remove those first before generating summary statistics. We can do this by either outright removing rows that contain an NA, or converting the NAs to another value such as 0. In this case we will just remove them. We do this with the `drop_na()` function. Many of these summary functions such as `mean()` also accept an optional field "na.rm" which removes the NAs only for that particular calculation (shown in the code).

```
flights %>%
  drop_na(dep_delay) %>%  # remove rows that contain an NA in the dep_delay column
  group_by(month) %>%
  summarise(n = n(),
            mean_dep_delay = mean(dep_delay, na.rm = T), # can also use na.rm = T instead of drop_na()
            min_dep_delay = min(dep_delay),
            max_dep_delay = max(dep_delay))
```

```
## # A tibble: 12 x 5
##    month     n mean_dep_delay min_dep_delay max_dep_delay
##    <dbl> <int>          <dbl>         <dbl>         <dbl>
##  1     1 26483          10.0            -30          1301
##  2     2 23690          10.8            -33           853
##  3     3 27973          13.2            -25           911
##  4     4 27662          13.9            -21           960
##  5     5 28233          13.0            -24           878
##  6     6 27234          20.8            -21          1137
##  7     7 28485          21.7            -22          1005
##  8     8 28841          12.6            -26           520
##  9     9 27122           6.72           -24          1014
## 10    10 28653           6.24           -25           702
## 11    11 27035           5.44           -32           798
## 12    12 27110          16.6            -43           896
```

If you don't supply any arguments to `drop_na()` it will look at the entire row of data and remove it if it sees an NA in any column. Alternatively you can supply it with column names like we have done here.

Exercise 3

1. What is the average flight distance from each airport?
2. Which carrier has the worst departure delays on average?
3. (Advanced) Can you separate the effects of bad airports and bad carriers?

Joins

The concept of joins can be a bit tricky to grasp, but it is very helpful to know how to use them. Often if we are working with complicated data we store information in separate files. For example, observations related to some disease and patient metadata are often stored in separate files. We can then link the information from the two files using one or more "key" columns present in both files, such as a patient ID.

This image shows the dplyr joins and how they work.

Let's say we want to find out how many seats on average an airplane has for each of the three NYC airports. To achieve this we need to read in a second data set that has information on the planes:

```
planes <- read_tsv("data/nycflights13_planes.txt")
```

```
## Rows: 3322 Columns: 9
## -- Column specification --------------------------------------------------------
## Delimiter: "\t"
## chr (5): tailnum, type, manufacturer, model, engine
## dbl (4): year, engines, seats, speed
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Take a quick look at the data:

```
glimpse(planes)
```

```
## Rows: 3,322
## Columns: 9
## $ tailnum      <chr> "N10156", "N102UW", "N103US", "N104UW", "N10575", "N105UW~
## $ year         <dbl> 2004, 1998, 1999, 1999, 2002, 1999, 1999, 1999, 1999, 199~
## $ type         <chr> "Fixed wing multi engine", "Fixed wing multi engine", "Fi~
## $ manufacturer <chr> "EMBRAER", "AIRBUS INDUSTRIE", "AIRBUS INDUSTRIE", "AIRBU~
## $ model        <chr> "EMB-145XR", "A320-214", "A320-214", "A320-214", "EMB-145~
## $ engines      <dbl> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ~
## $ seats        <dbl> 55, 182, 182, 182, 55, 182, 182, 182, 182, 182, 55, 55, 5~
## $ speed        <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
## $ engine       <chr> "Turbo-fan", "Turbo-fan", "Turbo-fan", "Turbo-fan", "Turb~
```

The information that links this and the flights data is the column "tailnum". Let's connect the "origin" column in the flights data to all the data in the planes data, using the "tailnum" column that appears in both data sets.

```
flights %>%
  select(origin, tailnum) %>%
  left_join(planes, by = "tailnum")
```

```
## # A tibble: 336,776 x 10
##    origin tailnum  year type       manufacturer model engines seats speed engine
##    <chr>  <chr>   <dbl> <chr>      <chr>        <chr>   <dbl> <dbl> <dbl> <chr>
##  1 EWR    N14228   1999 Fixed win~ BOEING       737-~       2   149    NA Turbo~
##  2 LGA    N24211   1998 Fixed win~ BOEING       737-~       2   149    NA Turbo~
##  3 JFK    N619AA   1990 Fixed win~ BOEING       757-~       2   178    NA Turbo~
##  4 JFK    N804JB   2012 Fixed win~ AIRBUS       A320~       2   200    NA Turbo~
##  5 LGA    N668DN   1991 Fixed win~ BOEING       757-~       2   178    NA Turbo~
##  6 EWR    N39463   2012 Fixed win~ BOEING       737-~       2   191    NA Turbo~
##  7 EWR    N516JB   2000 Fixed win~ AIRBUS INDU~ A320~       2   200    NA Turbo~
##  8 LGA    N829AS   1998 Fixed win~ CANADAIR     CL-6~       2    55    NA Turbo~
##  9 JFK    N593JB   2004 Fixed win~ AIRBUS       A320~       2   200    NA Turbo~
## 10 LGA    N3ALAA     NA <NA>       <NA>         <NA>       NA    NA    NA <NA>
## # ... with 336,766 more rows
```

Here we used a so-called "left join". Notice that the resulting data frame has the same number of rows as the original flights data frame. This is probably the most common type of join when using pipes, as we often have a "main" data frame (in this case it is flights) that we want to add some additional info columns to that are stored elsewhere (in this case - planes) without deleting or duplicating information in the main data.

There are some pitfalls when it comes to joining tables. If the key column contains duplicated values, e.g. if the "tailnum" column in the planes data had two rows with an identical tail number, those rows would be duplicated when using joins. This will mess up any downstream summary statistics you would perform on the data. See the example below.

An example of how joins can duplicate rows and skew summary statistics

Let's create two data frames to join. They will be similar to the flights data but much smaller. The first one will be tree rows, two columns: flights 1-3 with tail numbers A1-A3.

```
a <- tibble(flight = c(1,2,3), tailnum = c("A1","A2","A3"))
```

The second will be information on when the planes were built. Notice that someone made a mistake when creating this data, and has accidentally copied the A1 information twice!

```
b <- tibble(tailnum = c("A1", "A1", "A3", "A4"), year = c(2000, 2000, 2020, 2022))
```

Now look what happens when we join these tables:

```
c <- a %>%
  left_join(b, by = "tailnum")
c
```

```
## # A tibble: 4 x 3
##   flight tailnum  year
##    <dbl> <chr>   <dbl>
## 1      1 A1       2000
## 2      1 A1       2000
## 3      2 A2         NA
## 4      3 A3       2020
```

Since we have no information on tail number A2 in the b data frame, that value is listed as NA in the joined table. Also notice that there is no row containing A4 because we used a left join. If we wanted to calculate the average age of the planes in our data, and we had not noticed this mistake, we would get a wrong value out of it:

```
c %>%
  drop_na() %>%
  mutate(age = 2022 - year) %>%
  summarise(mean_age = mean(age))
```

```
## # A tibble: 1 x 1
##   mean_age
##      <dbl>
## 1     15.3
```

In this example it is obvious what is happening, but when working with large data sets with millions of observations, this can give you a real headache!

Exercise 4

1. Try joining together the dataframes `a` and `b` like in the example above but replacing `left_join()` with `right_join()`, `inner_join()`, `full_join()` and `anti_join()`. Can you predict what the resulting outputs will look like?
2. How would you join two tables when the key column has different names in each of the data frames?
3. What are the full names of the three NYC airports? Use the "faa" column in nycflights13_airports.txt and joins to find out!
4. What is the most common destination airport for each of the three NYC airports?
5. Which carrier is the only one that flies to Ted Stevens Anchorage Intl? Make sure to show the full name found in the file nycflights13_airlines.txt!

Plotting

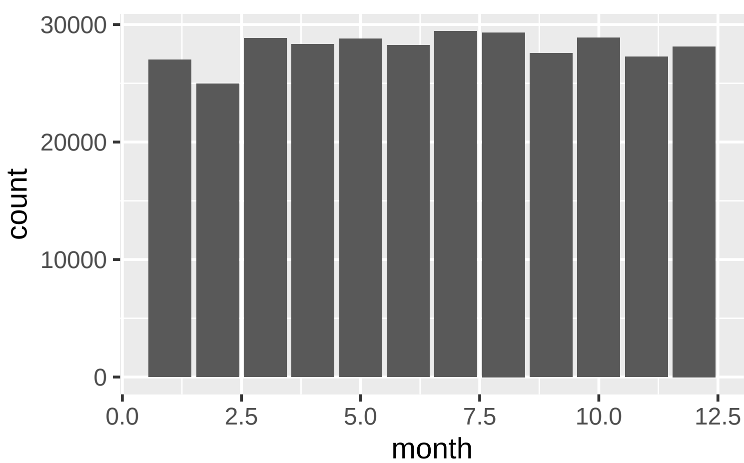
We will dedicate a full day for plotting later in the week, but it is very convenient to summarise data visually, so in this section we will quickly introduce the basics of plotting.

The R package ggplot2 is considered to be one of the most powerful and versatile data-plotting tools out there. It can be quite complicated and the syntax will seem strange at first. However it works brilliantly with the pipe operator and the other packages of the tidyverse. For it to work properly the data needs to be in the correct format (long-format data, we will cover this later).

Basic plots made with ggplot2

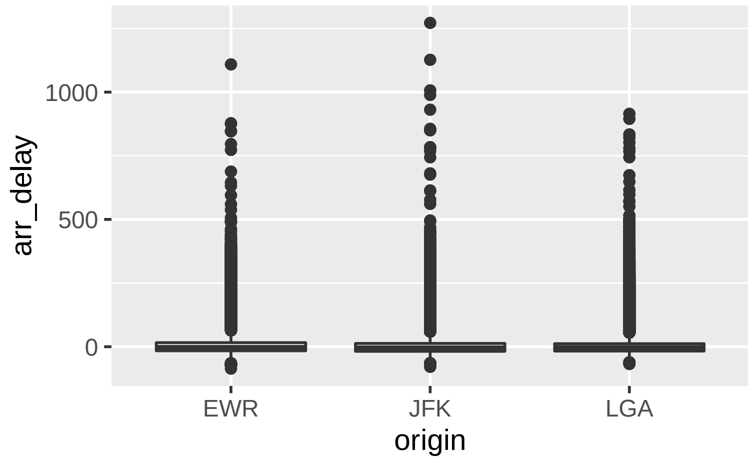
How many flights departed NYC each month?

```
flights %>%
  ggplot(aes(x = month)) +  # specify which data column goes on which axis
  geom_bar()                # make a barplot
```



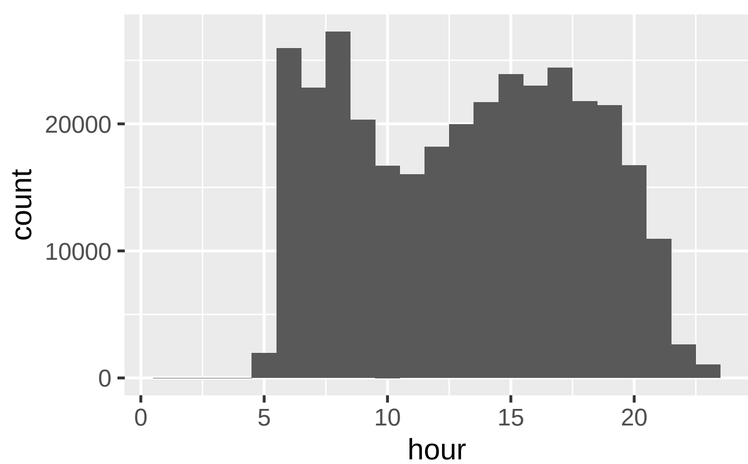
What is the mean arrival delay for each airport?

```
flights %>%
  drop_na(arr_delay) %>%
  ggplot(aes(x = origin, y = arr_delay)) +
  geom_boxplot()
```



At what time of day do flights generally depart?

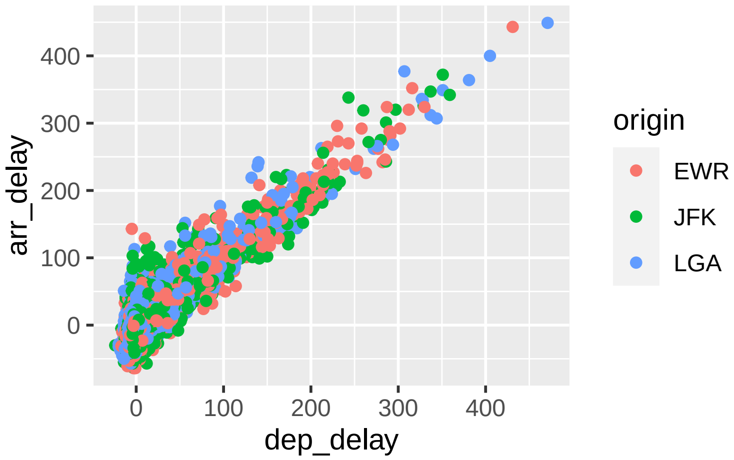
```
flights %>%
  ggplot(aes(x = hour)) +
  geom_histogram(bins = 23)
```



Can we spot a relation between departure delay and arrival delay? Note that here we are making a dot plot with one dot for each row of data - over 300,000 dots. Depending on the speed of your computer it might take a while to plot. If you feel like it takes too long to render you can filter random rows using the handy `slice_sample()` function.

```
flights %>%
  slice_sample(n = 10000) %>%
  ggplot(aes(x = dep_delay, y = arr_delay, col = origin)) +
  geom_point()
```

```
## Warning: Removed 284 rows containing missing values (geom_point).
```



As part 3 of this course covers plotting much more extensively, we will not have any plotting exercises in this chapter. Feel free to explore and try out plotting on your own!

Pivoting

Take a look at the following tables:

```
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
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

```
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
```

```
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
##   <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

```
table4b
```

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

They all store the same information, but in different ways. To work effectively with data you need to be able to transform it into a format that is efficient to work with. What format is best to use depends on what data you have, your research question, and what tools you are using to answer that question.

The tidyverse packages are made to work with "tidy" data - hence the name tidyverse. There are three criteria which make a data set tidy:

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

Exercise 5

1. Take another look at the table1 - 4 data frames. Which one of them do you think meets all the requirements to be "tidy"?

Let's have a look at another data set:

```
world_bank_pop %>%
  head()
```

```
## # A tibble: 6 x 20
##   country indicator      `2000`   `2001` `2002` `2003`  `2004`  `2005`   `2006`
##   <chr>   <chr>           <dbl>    <dbl>  <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
## 1 ABW     SP.URB.TOTL   42444      4.30e4 4.37e4 4.42e4 4.47e+4 4.49e+4  4.49e+4
## 2 ABW     SP.URB.GROW       1.18   1.41e0 1.43e0 1.31e0 9.51e-1 4.91e-1 -1.78e-2
## 3 ABW     SP.POP.TOTL   90853      9.29e4 9.50e4 9.70e4 9.87e+4 1.00e+5  1.01e+5
## 4 ABW     SP.POP.GROW       2.06   2.23e0 2.23e0 2.11e0 1.76e+0 1.30e+0  7.98e-1
## 5 AFG     SP.URB.TOTL 4436299      4.65e6 4.89e6 5.16e6 5.43e+6 5.69e+6  5.93e+6
## 6 AFG     SP.URB.GROW       3.91   4.66e0 5.13e0 5.23e0 5.12e+0 4.77e+0  4.12e+0
## # ... with 11 more variables: `2007` <dbl>, `2008` <dbl>, `2009` <dbl>,
## #   `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>,
## #   `2015` <dbl>, `2016` <dbl>, `2017` <dbl>
```

This data frame contains information about population numbers and growth for some countries over a span of several years. Is this data tidy?

If we want to take a look at how the population of a country changes over the years then this format is not optimal. ggplot2 and the other tidyverse packages work best when the data is in "long" format. In this case it would mean changing the data frame so that we replace columns 2000-2017 with a single column "year". This is done with the `pivot_longer()` function. Let's say we are only interested in the population growth for Sweden and Denmark:

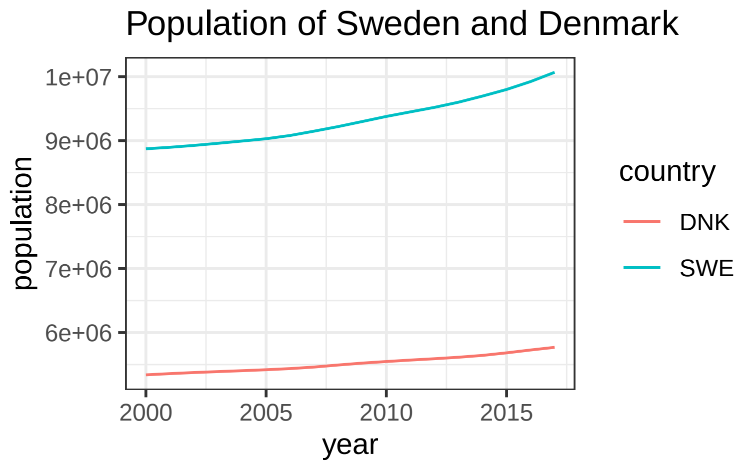
```r
swe_den_population <- world_bank_pop %>%
  filter(indicator == "SP.POP.TOTL") %>%      # filter to keep only total population information
  pivot_longer(cols = -c(country, indicator), # pivot all columns in the data except country and indica
               names_to = "year",             # specify the name of the key column
               values_to = "population") %>%  # specify the name of the value column
  filter(country %in% c("SWE", "DNK")) %>%    # filter to keep only Sweden and Denmark
  mutate(year = as.numeric(year))             # tell R that the year column contains numbers, not strin

swe_den_population
```

```
## # A tibble: 36 x 4
##    country indicator    year population
##    <chr>   <chr>       <dbl>      <dbl>
##  1 DNK     SP.POP.TOTL  2000    5339616
##  2 DNK     SP.POP.TOTL  2001    5358783
##  3 DNK     SP.POP.TOTL  2002    5375931
##  4 DNK     SP.POP.TOTL  2003    5390574
##  5 DNK     SP.POP.TOTL  2004    5404523
##  6 DNK     SP.POP.TOTL  2005    5419432
##  7 DNK     SP.POP.TOTL  2006    5437272
##  8 DNK     SP.POP.TOTL  2007    5461438
##  9 DNK     SP.POP.TOTL  2008    5493621
## 10 DNK     SP.POP.TOTL  2009    5523095
## # ... with 26 more rows
```

Now the data is in a form that is convenient to plot:

```r
swe_den_population %>%
  ggplot(aes(x = year, y = population, col = country)) +  # plot the data, color by country
  geom_line() + # specify that we want a line plot
  theme_bw() +  # make it a bit more pretty to look at
  labs(title = "Population of Sweden and Denmark") # add a title for the plot
```



Exercise 6

1. Try using the `pivot_wider()` function to transform table2 so that it looks exactly like table1. Look at the documentation for `pivot_wider()` to see the correct syntax. Documentation can be accessed by adding a "?" to the beginning of a function:

```
?pivot_wider()
```

Note: `pivot_wider()` has many optional arguments, the only ones you need to use for this exercise are "names_from" and "values_from".

Working with strings

Complicated data often comes with string columns such as identifiers, names, and categorical values. Tidyverse comes with powerful tools to work with strings, most are found in the aptly named stringr package. We will not focus so much on strings for now, but a couple of useful string-related functions are highlighted below.

For this section we will be using data on artwork found in the Tate Art Museum, artwork.csv. Note that this file, unlike the others we have worked with, is comma separated. To read in this data we use the `read_csv()` function.

```
artwork <- read_csv("data/artwork.csv")
```

```
## Rows: 69201 Columns: 20
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (12): accession_number, artist, artistRole, title, dateText, medium, cre...
## dbl  (7): id, artistId, year, acquisitionYear, width, height, depth
## lgl  (1): thumbnailCopyright
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
artwork %>%
  glimpse()
```

```
## Rows: 69,201
## Columns: 20
## $ id                 <dbl> 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 104~
## $ accession_number   <chr> "A00001", "A00002", "A00003", "A00004", "A00005", "~
## $ artist             <chr> "Blake, Robert", "Blake, Robert", "Blake, Robert", ~
## $ artistRole         <chr> "artist", "artist", "artist", "artist", "artist", "~
## $ artistId           <dbl> 38, 38, 38, 38, 39, 39, 39, 39, 39, 39, 39, 39, 39,~
## $ title              <chr> "A Figure Bowing before a Seated Old Man with his A~
## $ dateText           <chr> "date not known", "date not known", "?c.1785", "dat~
## $ medium             <chr> "Watercolour, ink, chalk and graphite on paper. Ver~
## $ creditLine         <chr> "Presented by Mrs John Richmond 1922", "Presented b~
## $ year               <dbl> NA, NA, 1785, NA, 1826, 1826, 1826, 1826, 1826, 182~
## $ acquisitionYear    <dbl> 1922, 1922, 1922, 1922, 1919, 1919, 1919, 1919, 191~
## $ dimensions         <chr> "support: 394 x 419 mm", "support: 311 x 213 mm", "~
## $ width              <dbl> 394, 311, 343, 318, 243, 240, 242, 246, 241, 243, 2~
## $ height             <dbl> 419, 213, 467, 394, 335, 338, 334, 340, 335, 340, 3~
## $ depth              <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ units              <chr> "mm", "mm", "mm", "mm", "mm", "mm", "mm", "mm", "mm~
## $ inscription        <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ thumbnailCopyright <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ thumbnailUrl       <chr> "http://www.tate.org.uk/art/images/work/A/A00/A0000~
## $ url                <chr> "http://www.tate.org.uk/art/artworks/blake-a-figure~
```

Use `str_sub()` to extract substrings (notice that this is done inside a `mutate()`):

```
artwork %>%
  mutate(title_short = str_sub(title, 1, 10)) %>% # new column that contains characters 1-10 of title
  select(year, artist, title_short)
```

```
## # A tibble: 69,201 x 3
##     year artist         title_short
##    <dbl> <chr>          <chr>
##  1    NA Blake, Robert  A Figure B
##  2    NA Blake, Robert  Two Drawin
##  3  1785 Blake, Robert  The Preach
##  4    NA Blake, Robert  Six Drawin
##  5  1826 Blake, William The Circle
##  6  1826 Blake, William Ciampolo t
##  7  1826 Blake, William The Baffle
##  8  1826 Blake, William The Six-Fo
##  9  1826 Blake, William The Serpen
## 10  1826 Blake, William The Pit of
## # ... with 69,191 more rows
```

Use `str_replace()` to replace parts of strings:

```
artwork %>%
  select(artist) %>%
  mutate(artist = str_replace(artist, "Blake", "John")) %>% # Replaces the first occurrence of "Blake" 
  head()
```

```
## # A tibble: 6 x 1
##   artist
##   <chr>
## 1 John, Robert
## 2 John, Robert
## 3 John, Robert
## 4 John, Robert
## 5 John, William
## 6 John, William
```

`separate()` splits the column into two or more new columns. `paste0()` can merge columns together (although that is not the only thing it can be used for!):

```
artwork %>%
  select(artist) %>%
  head() %>%
  separate(artist, into = c("last_name", "first_name"), sep = ",", remove = F) %>% # keep original colu
  mutate(full_name = paste0(first_name, " ", last_name))
```

```
## # A tibble: 6 x 4
##   artist         last_name first_name full_name
##   <chr>          <chr>     <chr>      <chr>
## 1 Blake, Robert  Blake     " Robert"  " Robert Blake"
## 2 Blake, Robert  Blake     " Robert"  " Robert Blake"
## 3 Blake, Robert  Blake     " Robert"  " Robert Blake"
## 4 Blake, Robert  Blake     " Robert"  " Robert Blake"
## 5 Blake, William Blake     " William" " William Blake"
## 6 Blake, William Blake     " William" " William Blake"
```

The stringr functions also support regular expression (shortened as regex), a way of searching for specific patterns in strings without knowing the exact sequence of characters. For example if you are looking for a subset of patient identifiers that all start with the letter A, followed by 4 digits and ending with the letter S, you would use regular expression to find them. We will not cover regex in this course, but the stringr cheat sheet has an excellent overview of it. Regex is not only used in data science but in many fields of programming, and being able to work with it is a very valuable skill to have.

Other useful functions we have not covered

`distinct()` - gives all unique values in a column

`pull()` - extracts a column into a 2D vector

`arrange()` - orders the data based on a given column, either alphabetically or numerically; can be combined with `desc()`; can take grouping into account through `.by_group = TRUE`

`rename()` - renames a column

`row_number()` - retrieves row number

`slice()` - subsets rows by row number; see also `slice_head()` and `slice_tail()`

`slice_max()` - subsets row(s) with the maximum value of a given column; see also `slice_min()`

`as_tibble()` - converts data into a tibble (the tidyverse version of a data frame)

`rownames_to_column()` - create a new column using the row names; useful for converting between matrices and data frames; see also `column_to_rownames()`