



# *An Introduction to R: Part I<sup>1</sup>*

Mark Andrews

Psychology Department, Nottingham Trent University

 @xmjandrews

 mark.andrews@ntu.ac.uk

 <https://github.com/mark-andrews/u-herts-r-workshop>

---

<sup>1</sup>These slides are not intended to be self-contained and comprehensive, but just aim to provide some of the workshop's content. Elaborations and explanations will be provided in the workshop itself.

## *What is R and why should you care*

- ▶ R is a program for doing statistics and data analysis.
- ▶ R's advantages or selling points relative to other programs (e.g, SPSS, SAS, Stata, Minitab, Python, Matlab, Maple, Mathematica, Tableau, Excel, SQL, and many others) come down to three inter-related factors:
  - ▶ It is immensely powerful.
  - ▶ It is open-source.
  - ▶ It is very and increasingly widely used.

## *R: A power tool for data analysis*

The range and depth of statistical analyses and general data analyses that can be accomplished with R is immense.

- ▶ Built into R are virtually the entire repertoire of widely known and used statistical methods.
- ▶ Also built in to R is an extensive graphics library.
- ▶ R has a vast set of add-on or contributed packages. There are presently 14531 additional contributed packages.
- ▶ R is a programming language that is specialized to efficiently manipulate and perform calculations on data.
- ▶ The R programming language itself can be extended by interfacing with other programming languages like C, C++, Fortran, Python, and high performance computing or big data tools like Hadoop, Spark, SQL.

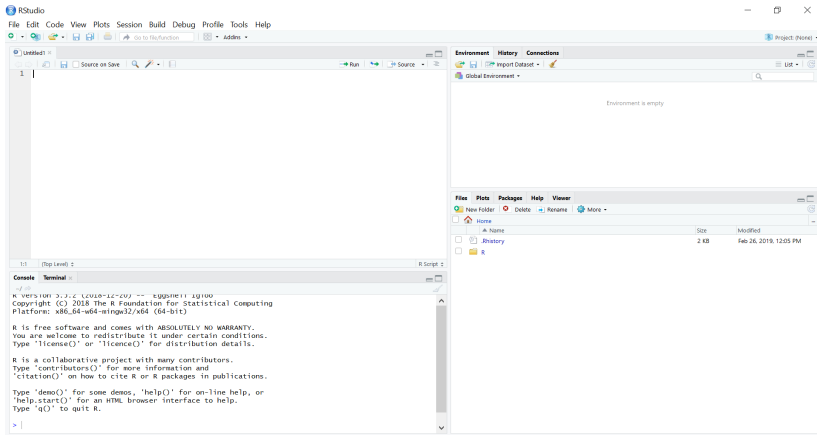
## *R: Open source software*

- ▶ R is free and open source software, distributed according to the GNU public license.
- ▶ Likewise, virtually all of contributed R packages are likewise free and open source.
- ▶ In practical terms, this means that is freely available for everyone to use, now and forever, on more or less any device they choose.
- ▶ Open source software always has the potential to *go viral* and develop a large self-sustaining community of user/developers. This has arguably happened with R.

## *R: Popularity and widespread use*

- ▶ When it comes to the computational implementation of modern statistical methods, R is the de facto standard. For example, the Journal of Statistical Software is overwhelmingly dominated by programs written in R.
- ▶ R is also currently very highly ranked according to many rankings of widely used programming languages of any kind. It ranked in the top 10 or top 20 most widely used programming languages.
- ▶ R is ranked as one of the top five most popular data science programs in jobs for data scientists, and in multiple surveys of data scientists, it is often ranked as the first or second mostly widely used data science tool.

# A guided tour of RStudio



## Introducing R commands

- A useful way to think about R, and not an inaccurate one either, is that it is simply a calculator.

```
> 2 + 2 # addition
#> [1] 4
> 3 - 5 # subtraction
#> [1] -2
> 3 * 2 # multiplication
#> [1] 6
> 4 / 3 # division
#> [1] 1.333333
> (2 + 2) ^ (3 / 3.5) # exponents and brackets
#> [1] 3.281341
```

## Equality/inequality operations

- Testing for the equality or inequality of pairs of numbers, already starts to go beyond the usual capabilities of handheld calculator.

```
> 12 == (6 * 2)           # test for equality
#> [1] TRUE
> (3 * 4) != (18 - 7)     # test for inequality
#> [1] TRUE
> 3 < 10                  # less than
#> [1] TRUE
> (2 * 5) <= 10           # less than or equal
#> [1] TRUE
```



## Logical values and logical operations

- ▶ In the previous step, the results are returned as either TRUE or FALSE. These are logical or *Boolean* values.
- ▶ Just as we can represent numbers and operations on numbers, so too can we have two logical values, TRUE and FALSE (always written in all uppercase), and Boolean operations (*and*, *or*, and *not*) on logical values.

```
> TRUE & FALSE # logical and
#> [1] FALSE
> TRUE | TRUE  # logical or
#> [1] TRUE
> !TRUE        # logical not
#> [1] FALSE
> (TRUE | !TRUE) & !FALSE
#> [1] TRUE
```

## Variables and assignment

- If we type the following at the command prompt and then press Enter, the result is displayed but not stored.

```
> (12/3.5)^2 + (1/2.5)^3 + (1 + 2 + 3)^0.33  
#> [1] 13.6254
```

- We can, however, assign the value of the above calculation to a variable named x.

```
> x <- (12/3.5)^2 + (1/2.5)^3 + (1 + 2 + 3)^0.33
```

- Now, we can use x as is it were a number.

```
> x  
#> [1] 13.6254  
> x ^ 2  
#> [1] 185.6516  
> x * 3.6  
#> [1] 49.05145
```

## *Assignment rules*

- ▶ In general, the assignment rule is

```
name <- expression
```

The expression is any R code that returns some value.

- ▶ The name must consist of letters, numbers, dots, and underscores.

```
x123    # acceptable
```

```
.x
```

```
x_y_z
```

```
xXx_123
```

- ▶ It must begin with a letter or a dot that is not followed by a number.

```
_x      # not acceptable
```

```
.2x
```

```
x-y-z
```

- ▶ The recommendation is to use names that are meaningful, relatively short, without dots (using `_` instead for punctuation), and primarily consisting of lowercase characters.

# Vectors

- ▶ Vectors are one dimensional sequences of values.
- ▶ For example, if we want to create a vector of the first 6 primes numbers, we could do the following.

```
> primes <- c(2, 3, 5, 7, 11, 13)
```

- ▶ We can now perform operations (arithmetic, logical, etc) on the primes vector.

```
> primes + 1
#> [1] 3 4 6 8 12 14
> primes / 2
#> [1] 1.0 1.5 2.5 3.5 5.5 6.5
> primes == 3
#> [1] FALSE TRUE FALSE FALSE FALSE FALSE
> primes >= 7
#> [1] FALSE FALSE FALSE TRUE TRUE TRUE
```

## Indexing vectors

- ▶ For any vector, we can refer to individual elements using indexing operations.

```
> primes[1]
#> [1] 2
> primes[5]
#> [1] 11
```

- ▶ If we want to refer to sets of elements, rather than just individual elements, we can use vectors (made with the `c()` function) inside the indexing square brackets.

```
> primes[c(3, 5, 2)]
#> [1] 5 11 3
```

- ▶ If we use a negative valued index, we can refer to or all elements *except* one.

```
> primes[-1]
#> [1] 3 5 7 11 13
> primes[-2]
#> [1] 2 5 7 11 13
```

## Vector types

- ▶ A vector be a sequence of numbers, logical values, or characters.

```
> nation <- c('ireland', 'england', 'scotland', 'wales')
```

- ▶ We can index this vector as we did with a vector of numbers.

```
> nation[1]
#> [1] "ireland"
> nation[2:3]
#> [1] "england" "scotland"
> nation == 'ireland'
#> [1] TRUE FALSE FALSE FALSE
```

- ▶ The class function in R will identify the data type of the vector.

```
> class(primes)
#> [1] "numeric"
> class(nation)
#> [1] "character"
> class(nation == 'ireland')
#> [1] "logical"
```

# Data frames

- ▶ Data frames are rectangular data structures; they have certain number of columns, and each column has the same number of rows. Each column is in fact a vector.
- ▶ Usually, data frames are created when read in the contents of a data file, but we can produce them on the command line with the `data.frame()`.

```
> Df <- data.frame(name = c('billy', 'joe', 'bob'),  
+                  age = c(21, 29, 23))  
> Df  
#>   name age  
#> 1 billy 21  
#> 2  joe 29  
#> 3  bob 23
```

## Indexing data frames

- ▶ We can refer to elements of a data frame in different ways.
- ▶ The simplest is to use double indices, one for the rows, one for the columns.

```
> Df[3, 2] # row 3, col 2
#> [1] 23
> Df[c(1, 3), 2] # rows 1 and 3, col 2
#> [1] 21 23
> Df[1,] # row 1, all cols
#>   name age
#> 1  billy 21
> Df[, 2] # all rows, col 2
#> [1] 21 29 23
```



## Indexing data frames (continued)

- ▶ We could also refer to the column by name. To do so, we could use the following \$ notation.

```
> Df$age  
#> [1] 21 29 23
```

- ▶ An alternative syntax that accomplishes the same thing is to use *double* square brackets as follows.

```
> Df[['age']]  
#> [1] 21 29 23
```

- ▶ A *single* square brackets, we would obtain the following.

```
> Df['age']  
#>   age  
#> 1  21  
#> 2  29  
#> 3  23
```

# Functions

- ▶ In functions, we put data in, calculations or done to or using this data, and new data, perhaps just a single value, is then returned.
- ▶ There are probably hundreds of thousands of functions in R.
- ▶ For example,

```
> length(primes)
#> [1] 6
> sum(primes)
#> [1] 41
> mean(primes)
#> [1] 6.833333
> median(primes)
#> [1] 6
> sd(primes)
#> [1] 4.400758
> var(primes)
#> [1] 19.36667
```

## Custom functions

- ▶ R makes it easy to create new functions.

```
> my_mean <- function(x){ sum(x)/length(x) }
```

- ▶ This my\_mean takes a vector as input and divides its sum by the number of elements in it. It then returns this values. The x is a placeholder for whatever variable we input into the function.
- ▶ We would use it just as we would use mean.

```
> my_mean(primes)
#> [1] 6.833333
```

## Writing R scripts

- ▶ Scripts are files where we write R commands, which can be then saved for later use.
- ▶ You can bring up RStudio's script editor with Ctrl+Shift+N, or go to the File/ New File/ R script, or click on the New icon on the left of the taskbar below the menu and choose R script.
- ▶ In a script, you can have as many lines of code as you wish, and there can be as many blank lines as you wish.

```
1 composites <- c(4, 6, 8, 9, 10, 12)
2
3 composites_plus_one <- composites + 1
4
5 composites_minus_one <- composites - 1
```

- ▶ If you place the cursor on line 1, you can then click the Run icon, or press the Ctrl+Enter keys.

## Writing R scripts (continued)

One reason why writings in scripts is very practically valuable, even if you don't wish to save the scripts, is when you are write long and complex commands.

```
1 Df <- data.frame(name = c('jane', 'joe', 'billy'),  
2                   age = c(23, 27, 24),  
3                   sex = c('female', 'male', 'male'),  
4                   occupation = c('tinker', 'tailor', 'spy')  
5 )
```

We can execute this command as if it were on a single line by placing the cursor anywhere on any line and pressing Ctrl+Enter.

## Code comments

- ▶ An almost universal feature of programming language is the option to write *comments* in the code files.
- ▶ A comment allows you write to notes or comments around the code that is then skipped over when the script or the code lines are being executed.
- ▶ In R, anything following the # symbol on any line is treated as a comment.

```
1  # Here is a data frame with four variables.
2  # The variables are name, age, sex, occupation.
3  Df <- data.frame(name = c('jane', 'joe', 'billy'),
4                  # This line is a comment too.
5                  age = c(23, 27, 24), # Another comment.
6                  sex = c('female', 'male', 'male'),
7                  occupation = c('tinker', 'tailor', 'spy')
8  )
```

# Packages

- ▶ There are presently 14531 contributed packages in R.
- ▶ The easiest way to install a package is to click the Install button on the top left of the *Packages* window in the lower right pane.
- ▶ You can also install a package or packages with the `install.packages` command.

```
> install.packages("dplyr")  
> install.packages(c("dplyr", "tidyr", "ggplot2"))
```

- ▶ Having installed a package, it must be loaded to be used. This can be done by clicking the tick box before the package name in the *Packages* window, or use the `library` command.

```
> library("tidyverse")
```

## Reading in data

- ▶ R allows you to import data from a very large variety of data file types, including from other statistics programs like SPSS, Stata, SAS, Minitab, and so on, and common file formats like `.xlsx` and `.csv`.
- ▶ When learning R initially, the easiest way to import data is using the Import Dataset button in the Environment window.
- ▶ If we use the *From Text (readr)*... option, it runs the `read_csv` R command, which we can run ourselves on the command line, or write in a script.

```
> library(readr)
> blp_df <- read_csv("data/blp-trials-short.txt")
```



## Viewing data

- The easiest way to view a data frames is to type its name.

```
> blp_df
#> # A tibble: 1,000 x 7
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl>    <dbl> <dbl>
#> 1      20 N    staud    N      977      511    977
#> 2       9 N    dinbuss  N      565      765    565
#> 3      47 N    snilling N      562      496    562
#> 4     103 N    gancens  N      572      656    572
#> 5      45 W    filled   W      659      981    659
#> 6      73 W    journals W      538     1505    538
#> 7      24 W    apache   W      626      546    626
#> 8      11 W    flake    W      566      717    566
#> 9      32 W    reliefs  W      922     1471    922
#> 10     96 N    sarves   N      555      806    555
#> # ... with 990 more rows
```

## Viewing data (continued)

- ▶ Another option to view a data frame is to `glimpse` it.

```
> glimpse(blp_df)
#> Observations: 1,000
#> Variables: 7
#> $ participant <dbl> 20, 9, 47, 103, 45, 73, 24, 11, 32, 96, 8
#> $ lex         <chr> "N", "N", "N", "N", "W", "W", "W", "W", "
#> $ spell       <chr> "staud", "dinbuss", "snilling", "gancens"
#> $ resp        <chr> "N", "N", "N", "N", "W", "W", "W", "W", "
#> $ rt          <dbl> 977, 565, 562, 572, 659, 538, 626, 566, 9
#> $ prev.rt     <dbl> 511, 765, 496, 656, 981, 1505, 546, 717,
#> $ rt.raw      <dbl> 977, 565, 562, 572, 659, 538, 626, 566, 9
```

## Summarizing data with summary

- An easy way to summarize a data frame is with summary.

```
> summary(blp_df)
```

```
#>   participant          lex          spell          r
#>   Min.      : 1.00   Length:1000   Length:1000   Length:1000
#>   1st Qu.: 20.00   Class :character Class :character Class :character
#>   Median : 47.00   Mode  :character Mode  :character Mode  :character
#>   Mean     : 49.46
#>   3rd Qu.: 75.00
#>   Max.     :105.00
#>
#>          rt          prev.rt          rt.raw
#>   Min.      : 361.0   Min.      : 0.0   Min.      : 335.0
#>   1st Qu.: 514.0   1st Qu.: 510.0   1st Qu.: 518.0
#>   Median : 588.0   Median : 594.0   Median : 605.0
#>   Mean     : 637.8   Mean     : 660.1   Mean     : 707.9
#>   3rd Qu.: 708.2   3rd Qu.: 712.0   3rd Qu.: 754.0
#>   Max.     :1750.0   Max.     :2413.0   Max.     :9925.0
#>   NA's     :176     NA's      :3
```

## *Data wrangling tools in R*

- ▶ There are many tools in R for doing data wrangling. Here, we will focus of a core set of inter-related `tidyverse` tools.
- ▶ We'll begin with commands available in the `dplyr` package, particularly its so-called *verbs* such as the following.
  - ▶ `select`
  - ▶ `rename`
  - ▶ `slice`
  - ▶ `filter`
  - ▶ `mutate`
  - ▶ `arrange`
  - ▶ `group_by`
  - ▶ `summarize`
- ▶ All of these tools are can be combined together using the `%>%` pipe operator

## Selecting variables with *select*

- The `dplyr` command `select` allows us to select variables from a data frame we want.

```
> select(blp_df, participant, lex, resp, rt)
#> # A tibble: 1,000 x 4
#>   participant lex    resp    rt
#>   <dbl> <chr> <chr> <dbl>
#> 1      20 N      N      977
#> 2       9 N      N      565
#> 3      47 N      N      562
#> 4     103 N      N      572
#> 5      45 W      W      659
#> 6      73 W      W      538
#> 7      24 W      W      626
#> 8      11 W      W      566
#> 9      32 W      W      922
#> 10     96 N      N      555
#> # ... with 990 more rows
```

## Selecting variables with *select*

- We can select a range of variables by specifying the first and last variables in the range with a `:` between them as follows.

```
> select(blp_df, spell:prev.rt)
#> # A tibble: 1,000 x 4
#>   spell      resp      rt prev.rt
#>   <chr>      <chr> <dbl>   <dbl>
#> 1 staud      N      977     511
#> 2 dinbuss    N      565     765
#> 3 snilling  N      562     496
#> 4 gancens    N      572     656
#> 5 filled     W      659     981
#> 6 journals   W      538    1505
#> 7 apache     W      626     546
#> 8 flake      W      566     717
#> 9 reliefs    W      922    1471
#> 10 sarves    N      555     806
#> # ... with 990 more rows
```

## Selecting variables with *select*

- We can also select a range of variables using indices as in the following example.

```
> select(blp_df, 2:5) # columns 2 to 5
#> # A tibble: 1,000 x 4
#>   lex      spell      resp      rt
#>   <chr> <chr>      <chr> <dbl>
#> 1 N      staud      N      977
#> 2 N      dinbuss    N      565
#> 3 N      snilling   N      562
#> 4 N      gancens    N      572
#> 5 W      filled     W      659
#> 6 W      journals   W      538
#> 7 W      apache     W      626
#> 8 W      flake      W      566
#> 9 W      reliefs    W      922
#> 10 N     sarves     N      555
#> # ... with 990 more rows
```

## Selecting variables with *select*

- We can select variables according to the character or characters that they begin with.

```
> select(blp_df, starts_with('p'))
```

```
#> # A tibble: 1,000 x 2
```

```
#>   participant prev.rt
```

```
#>           <dbl>   <dbl>
```

```
#> 1             20     511
```

```
#> 2              9     765
```

```
#> 3             47     496
```

```
#> 4            103     656
```

```
#> 5             45     981
```

```
#> 6             73    1505
```

```
#> 7             24     546
```

```
#> 8             11     717
```

```
#> 9             32    1471
```

```
#> 10            96     806
```

```
#> # ... with 990 more rows
```



## Selecting variables with *select*

- We can select variables by the characters they end with.

```
> select(blp_df, ends_with('t'))  
#> # A tibble: 1,000 x 3  
#>   participant      rt prev.rt  
#>   <dbl> <dbl>   <dbl>  
#> 1         20   977     511  
#> 2          9   565     765  
#> 3         47   562     496  
#> 4        103   572     656  
#> 5         45   659     981  
#> 6         73   538    1505  
#> 7         24   626     546  
#> 8         11   566     717  
#> 9         32   922    1471  
#> 10        96   555     806  
#> # ... with 990 more rows
```

## Selecting variables with *select*

- We can select variables that contain a certain set of characters in any position.

```
> select(blp_df, contains('rt'))  
#> # A tibble: 1,000 x 4  
#>   participant      rt prev.rt rt.raw  
#>   <dbl> <dbl>   <dbl> <dbl>  
#> 1         20    977     511    977  
#> 2          9    565     765    565  
#> 3         47    562     496    562  
#> 4        103    572     656    572  
#> 5         45    659     981    659  
#> 6         73    538    1505    538  
#> 7         24    626     546    626  
#> 8         11    566     717    566  
#> 9         32    922    1471    922  
#> 10        96    555     806    555  
#> # ... with 990 more rows
```

## Selecting variables with *select*

- ▶ We can select variables using *regular expressions*.
- ▶ For example, the regular expression `^rt|rt$` will match the `rt` if it begins or ends a string.

```
> select(blp_df, matches('^rt|rt$'))
```

```
#> # A tibble: 1,000 x 3
```

```
#>       rt prev.rt rt.raw
```

```
#>   <dbl>   <dbl> <dbl>
```

```
#> 1    977     511    977
```

```
#> 2    565     765    565
```

```
#> 3    562     496    562
```

```
#> 4    572     656    572
```

```
#> 5    659     981    659
```

```
#> 6    538    1505    538
```

```
#> 7    626     546    626
```

```
#> 8    566     717    566
```

```
#> 9    922    1471    922
```

```
#> 10   555     806    555
```

```
#> # ... with 990 more rows
```

## Removing variables with *select*

- We can use `select` to *remove* variables as well as select them. To remove a variable, we precede its name with a minus sign.

```
> select(blpl_df, -participant) # remove `participant`  
#> # A tibble: 1,000 x 6  
#>   lex      spell      resp      rt prev.rt rt.raw  
#>   <chr> <chr>      <chr> <dbl>   <dbl>  <dbl>  
#> 1 N      staud      N      977     511    977  
#> 2 N      dinbuss    N      565     765    565  
#> 3 N      snilling   N      562     496    562  
#> 4 N      gancens    N      572     656    572  
#> 5 W      filled     W      659     981    659  
#> 6 W      journals   W      538    1505    538  
#> 7 W      apache     W      626     546    626  
#> 8 W      flake      W      566     717    566  
#> 9 W      reliefs    W      922    1471    922  
#> 10 N     sarves     N      555     806    555  
#> # ... with 990 more rows
```

## Removing variables with *select*

- To remove variables indexed 2 to 6, we would do the following.

```
> select(blp_df, -(2:6))  
#> # A tibble: 1,000 x 2  
#>   participant rt.raw  
#>   <dbl> <dbl>  
#> 1      20    977  
#> 2       9    565  
#> 3      47    562  
#> 4     103    572  
#> 5      45    659  
#> 6      73    538  
#> 7      24    626  
#> 8      11    566  
#> 9      32    922  
#> 10     96    555  
#> # ... with 990 more rows
```

## Reordering variables with *select* and *everything()*

- We can rearrange variables by selecting some and then using *everything()* for the others.

```
> select(blpl_df, spell, participant, resp, everything())  
#> # A tibble: 1,000 x 7  
#>   spell      participant resp  lex      rt prev.rt rt.raw  
#>   <chr>          <dbl> <chr> <chr> <dbl>    <dbl> <dbl>  
#> 1 staud             20 N      N      977      511    977  
#> 2 dinbuss           9 N      N      565      765    565  
#> 3 snilling          47 N      N      562      496    562  
#> 4 gancens          103 N      N      572      656    572  
#> 5 filled           45 W      W      659      981    659  
#> 6 journals          73 W      W      538     1505    538  
#> 7 apache           24 W      W      626      546    626  
#> 8 flake            11 W      W      566      717    566  
#> 9 reliefs           32 W      W      922     1471    922  
#> 10 sarves           96 N      N      555      806    555  
#> # ... with 990 more rows
```

## *Renaming with rename*

- When we select individual variables with `select`, we can rename them too.

```
> select(blp_df, subject=participant, reaction_time=rt)
#> # A tibble: 1,000 x 2
#>   subject reaction_time
#>   <dbl>         <dbl>
#> 1         20         977
#> 2          9         565
#> 3         47         562
#> 4        103         572
#> 5         45         659
#> 6         73         538
#> 7         24         626
#> 8         11         566
#> 9         32         922
#> 10        96         555
#> # ... with 990 more rows
```

- But this just return the selected variables.

## *Renaming with rename*

- If we want to rename some variables, and get a data frame with all variables, including the renamed ones, we should use `rename`.

```
> rename(blp_df, subject=participant, reaction_time=rt)
#> # A tibble: 1,000 x 7
#>   subject lex spell resp reaction_time prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr>          <dbl>   <dbl> <dbl>
#> 1      20 N  staud      N           977     511   977
#> 2       9 N  dinbuss    N           565     765   565
#> 3      47 N  snilling  N           562     496   562
#> 4     103 N  gancens   N           572     656   572
#> 5      45 W  filled    W           659     981   659
#> 6      73 W  journals  W           538    1505   538
#> 7      24 W  apache    W           626     546   626
#> 8      11 W  flake     W           566     717   566
#> 9      32 W  reliefs   W           922    1471   922
#> 10     96 N  sarves    N           555     806   555
#> # ... with 990 more rows
```



## Selecting observations with *slice*

- To select rows 10, 20, 50, 100, 500:

```
> slice(blp_df, c(10, 20, 50, 100, 500))  
#> # A tibble: 5 x 7  
#>   participant lex    spell resp    rt prev.rt rt.raw  
#>       <dbl> <chr> <chr> <chr> <dbl>   <dbl> <dbl>  
#> 1         96 N    sarves N    555     806   555  
#> 2         46 W    mirage W    778     571   778  
#> 3         72 N    gright N    430     675   430  
#> 4          3 W    gleam  W    361     370   361  
#> 5         92 W    coaxes W    699     990   699
```

## Selecting observations with *slice*

- To select rows 10 to 100 inclusive:

```
> slice(blp_df, 10:100)
```

```
#> # A tibble: 91 x 7
```

#>	participant	lex	spell	resp	rt	prev.rt	rt.raw
#>		<dbl>	<chr>	<chr>	<chr>	<dbl>	<dbl>
#> 1	96	N	sarves	N	555	806	555
#> 2	82	W	deceits	W	657	728	657
#> 3	37	W	nothings	N	NA	552	712
#> 4	52	N	chuespies	N	427	539	427
#> 5	96	N	mowny	N	1352	1020	1352
#> 6	96	N	cranned	N	907	573	907
#> 7	89	N	flud	N	742	834	742
#> 8	3	N	bromble	N	523	502	523
#> 9	7	N	trubbles	N	782	458	782
#> 10	35	N	playfound	N	643	663	643

```
#> # ... with 81 more rows
```

## Selecting observations with *slice*

A useful dplyr function that can be used in `slice` and elsewhere is `n()`, which gives the number of observations in the data frame.

```
> slice(blp_df, 600:n())  
#> # A tibble: 401 x 7  
#>   participant lex spell resp rt prev.rt rt.raw  
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>  
#> 1 16 W earthworms W 767 659 767  
#> 2 50 W markers W 664 852 664  
#> 3 35 N spoton N 522 721 522  
#> 4 88 W tawny N NA 535 856  
#> 5 51 N gember N 562 598 562  
#> 6 63 W classed W 706 429 706  
#> 7 63 N clallers N 401 495 401  
#> 8 8 W pauper W 734 1126 734  
#> 9 2 W badges W 485 498 485  
#> 10 97 N foarded N 802 464 802  
#> # ... with 391 more rows
```

## Dropping observations with *slice*

- We can drop the first 10 observations as follows.

```
> slice(blp_df, -(1:10))  
#> # A tibble: 990 x 7  
#>   participant lex      spell      resp      rt prev.rt rt.raw  
#>   <dbl> <chr> <chr>      <chr> <dbl>   <dbl> <dbl>  
#> 1      82 W    deceits    W      657     728    657  
#> 2      37 W    nothings  N      NA     552    712  
#> 3      52 N    chuespies N     427     539    427  
#> 4      96 N    mowny     N    1352    1020   1352  
#> 5      96 N    cranned   N     907     573    907  
#> 6      89 N    flud      N     742     834    742  
#> 7       3 N    bromble   N     523     502    523  
#> 8       7 N    trubbles  N     782     458    782  
#> 9      35 N    playfound N     643     663    643  
#> 10     46 W    mirage    W     778     571    778  
#> # ... with 980 more rows
```

## Selecting observations with *filter*

- ▶ The `filter` command is a powerful means to filter (i.e., to filter in, not filter out) observations according to their values.
- ▶ For example, we can select all the observations where the `lex` variable is N.

```
> filter(blp_df, lex == 'N')
```

```
#> # A tibble: 502 x 7
```

```
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl>    <dbl> <dbl>
#> 1      20 N    staud    N     977     511    977
#> 2       9 N    dinbuss  N     565     765    565
#> 3      47 N    snilling N     562     496    562
#> 4     103 N    gancens  N     572     656    572
#> 5      96 N    sarves   N     555     806    555
#> 6      52 N    chuespies N     427     539    427
#> 7      96 N    mowny    N    1352    1020   1352
#> 8      96 N    cranned  N     907     573    907
#> 9      89 N    flud     N     742     834    742
#> 10      3 N    bromble  N     523     502    523
#> # ... with 492 more rows
```

## Selecting observations with *filter*

- We can filter by multiple conditions by listing each one with commas between them.

```
> filter(blp_df, lex == 'N', resp == 'W')
#> # A tibble: 35 x 7
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl>    <dbl> <dbl>
#> 1         73 N    bunding  W      NA      978    1279
#> 2         63 N    gallays  W      NA      589     923
#> 3         50 N    droper   W      NA      741     573
#> 4          6 N    flooder  W      NA      524     557
#> 5         73 N    khantum  W      NA      623    1355
#> 6         81 N    seaped   W      NA      765     691
#> 7         43 N    gafers   W      NA      556     812
#> 8        101 N    winchers W      NA      632     852
#> 9         81 N    flaged   W      NA      674     609
#> 10        11 N    frocker  W      NA      653     665
#> # ... with 25 more rows
```

## Selecting observations with *filter*

- ▶ We can make a *disjunction* of conditions for filtering using the logical-or symbol `|`.
- ▶ For example, here we select observations where `rt.raw` is less than 500 *or* greater than 1000.

```
> filter(blp_df, rt.raw < 500 | rt.raw > 1000)
```

```
#> # A tibble: 296 x 7
```

```
#>   participant lex    spell      resp      rt prev.rt rt.raw
#>   <dbl> <chr> <chr>      <chr> <dbl>   <dbl> <dbl>
#> 1      52 N    chuespies  N      427     539    427
#> 2      96 N    mowny      N     1352    1020   1352
#> 3      28 W    stelae     N      NA     678    497
#> 4      85 W    forewarned N      NA     525    350
#> 5      24 W    owl       W     470     535    470
#> 6      97 W    soda        W     436     447    436
#> 7      81 N    fugate      N     425     403    425
#> 8     105 N    pamps       N      NA     884   1494
#> 9      27 W    outgrowth   N      NA     633   1014
#> 10     82 W    kitty       W     431     476    431
#> # ... with 286 more rows
```

## Selecting observations with `filter`

- here we filter observations where values of `rt.raw` is *in* the set on integers 500 to 510.

```
> filter(blp df, rt.raw %in% 500:510)
```

```
#> # A tibble: 26 x 7
```

[illegible]



## Selecting observations with *filter*

In general, we may filter the observations by creating complex Boolean conditionals

```
> filter(blp_df,  
+       lex == 'W',  
+       str_length(spell) < 5 & (resp != lex | rt.raw > 900))
```

```
#> # A tibble: 14 x 7
```

```
#>   participant lex    spell resp    rt prev.rt rt.raw  
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>  
#> 1      21 W    bosk  N    NA    608    1532  
#> 2      68 W    wily  N    NA    723    636  
#> 3      30 W    sew   N    NA    473    524  
#> 4      34 W    jibs  N    NA    781    756  
#> 5      85 W    rote  N    NA    505    458  
#> 6      13 W    ooofs N    NA    560    654  
#> 7      72 W    awed  N    NA    1203   1801  
#> 8      14 W    yids  N    NA    625    620  
#> 9      68 W    oho   N    NA    633    630  
#> 10     103 W    carl  N    NA    1046   1042  
#> 11      46 W    brae  N    NA    644    720
```

## Changing variables and values using *mutate*

- ▶ The `mutate` command is a very powerful tool in the `dplyr` toolbox.
- ▶ As an example, we can create a new variable `acc` that takes the value of `TRUE` whenever `lex` and `resp` have the same value as follows.

```
> mutate(blp_df, acc = lex == resp)
```

```
#> # A tibble: 1,000 x 8
```

```
#>   participant lex    spell    resp    rt prev.rt rt.raw acc
#>   <dbl> <chr> <chr>    <chr> <dbl>    <dbl>    <dbl> <lgl>
#> 1         20 N    staud    N     977      511     977 TRUE
#> 2          9 N   dinbuss  N     565      765     565 TRUE
#> 3         47 N   snilling N     562      496     562 TRUE
#> 4        103 N   gancens  N     572      656     572 TRUE
#> 5         45 W    filled  W     659      981     659 TRUE
#> 6         73 W   journals W     538     1505     538 TRUE
#> 7         24 W    apache  W     626      546     626 TRUE
#> 8         11 W    flake    W     566      717     566 TRUE
#> 9         32 W   reliefs  W     922     1471     922 TRUE
#> 10        96 N    sarves  N     555      806     555 TRUE
#> # ... with 990 more rows
```

## Changing variables and values using *mutate*

- ▶ We can also create multiple new variable at the same time as in the following example.

```
> mutate(blp_df,  
+       acc = lex == resp,  
+       fast = rt.raw < mean(rt.raw, na.rm=TRUE))  
#> # A tibble: 1,000 x 9  
#>   participant lex    spell    resp    rt prev.rt rt.raw acc  
#>   <dbl> <chr> <chr>    <chr> <dbl>    <dbl> <dbl> <lgl>  
#> 1         20 N    staud    N     977      511    977 TRUE  
#> 2          9 N   dinbuss  N     565      765    565 TRUE  
#> 3         47 N   snilling N     562      496    562 TRUE  
#> 4        103 N   gancens  N     572      656    572 TRUE  
#> 5         45 W   filled    W     659      981    659 TRUE  
#> 6         73 W   journals W     538     1505    538 TRUE  
#> 7         24 W   apache    W     626      546    626 TRUE  
#> 8         11 W   flake     W     566      717    566 TRUE  
#> 9         32 W   reliefs   W     922     1471    922 TRUE  
#> 10        96 N   sarves    N     555      806    555 TRUE  
#> # ... with 990 more rows
```

## Sorting with arrange

- ▶ Sorting observations in a data frame is easily accomplished with `arrange`.

```
> arrange(blp_df, participant, spell) # sort by `participant`, t
#> # A tibble: 1,000 x 7
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl>    <dbl> <dbl>
#> 1         1 W    abyss    W      629      683      629
#> 2         1 N    baises    N      524      574      524
#> 3         1 W    carport    W      779      605      779
#> 4         1 N    cellies    N      792      652      792
#> 5         1 W    chafing    W      601      720      601
#> 6         1 N    dametails N      694      635      694
#> 7         1 N    footer    N      789      566      789
#> 8         1 W    gantries    W      644      581      644
#> 9         1 N    hogtush    N      679      568      679
#> 10        1 N    lisedess    N      679      619      679
#> # ... with 990 more rows
```

## Sorting with arrange

- We can sort by the reverse order of any variable by using the desc.

```
> arrange(blp_df, participant, desc(spell))
#> # A tibble: 1,000 x 7
#>   participant lex spell resp rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1         1 N wintes N 545 629 545
#> 2         1 N treeps N 607 610 607
#> 3         1 W squashes W 494 491 494
#> 4         1 N sinkhicks N 536 519 536
#> 5         1 W shafting W 553 571 553
#> 6         1 W month W 500 498 500
#> 7         1 N lisedess N 679 619 679
#> 8         1 N hogtush N 679 568 679
#> 9         1 W gantries W 644 581 644
#> 10        1 N footer N 789 566 789
#> # ... with 990 more rows
```

## Summarizing with *summarize*

- The `summarize` (or `summarise`) function allows us to calculate summary statistics.

```
> summarize(blp_df,
+           mean_rt = mean(rt, na.rm = T),
+           median_rt = median(rt, na.rm = T),
+           sd_rt.raw = sd(rt.raw, na.rm = T)
+ )
#> # A tibble: 1 x 3
#>   mean_rt median_rt sd_rt.raw
#>   <dbl>     <dbl>     <dbl>
#> 1   638.       588       474.
```

## Summarizing with *summarize* and *group\_by*

- Combined with *group\_by*, *summarize* allows us to calculate summary statistics by group

```
> summarize(group_by(blp_df, lex),  
+           mean_rt = mean(rt, na.rm = T),  
+           median_rt = median(rt, na.rm = T),  
+           sd_rt.raw = sd(rt.raw, na.rm = T)  
+ )  
#> # A tibble: 2 x 4  
#>   lex   mean_rt median_rt sd_rt.raw  
#>   <chr>   <dbl>     <dbl>    <dbl>  
#> 1 N      638.       585      307.  
#> 2 W      637.       588      597.
```

## The %>% operator

- ▶ The %>% operator in R is known as the *pipe*.
- ▶ In RStudio, the keyboard shortcut Ctrl+Shift+M types %>%.
- ▶ To understand pipes, let us begin with a simple example of nested functions.

```
> primes <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

```
> sum(primes) # The sum of `primes`  
#> [1] 129
```

```
> sqrt(sum(primes)) # The sqrt of the sum  
#> [1] 11.35782
```

```
> log(sqrt(sum(primes))) # The log of the sqrt of the sum  
#> [1] 2.429906
```



## The %>% operator

- ▶ The %>% allows us to rewrite nested functions as *pipelines*.
- ▶ This

```
> log(sqrt(sum(primes))) # The log of the sqrt of the sum  
#> [1] 2.429906
```

is equivalent to this

```
> primes %>% sum() %>% sqrt() %>% log()  
#> [1] 2.429906
```

## Using %>% with dplyr

- ▶ When used with dplyr verbs, the %>% provides a mini-language for data wrangling.

```
> b1p_df %>%  
+   mutate(accuracy = resp == lex,  
+         stimulus = recode(lex, 'W'='word', 'N'='nonword'))  
+   %>%  
+   select(stimulus, item=spell, accuracy, speed=rt.raw) %>%  
+   arrange(speed) %>%  
+   head(5)  
#> # A tibble: 5 x 4  
#>   stimulus item      accuracy speed  
#>   <chr>    <chr>    <lgl>    <dbl>  
#> 1 word    hopeless FALSE     335  
#> 2 word    tutored   FALSE     348  
#> 3 word    forewarned FALSE     350  
#> 4 word    gleam     TRUE     361  
#> 5 word    looks     TRUE     365
```

## Combining data frames with *bind*

- A *bind* operation is a simple operation that either vertically stack data frames that share common variables, or horizontally stack data frames that have the same number of observations.

To illustrate, we will create three small data frames.

```
> Df_1 <- tibble(x = c(1, 2, 3),  
+               y = c(2, 7, 1),  
+               z = c(0, 2, 7))  
>  
> Df_2 <- tibble(y = c(5, 7),  
+               z = c(6, 7),  
+               x = c(1, 2))  
>  
> Df_3 <- tibble(a = c(5, 6, 1),  
+               b = c('a', 'b', 'c'),  
+               c = c(T, T, F))
```

## Combining data frames with *bind*

- ▶ The Df\_1 and Df\_2 data frames share common variable names. They can be vertically stacked using a `bind_rows` operation.

```
> bind_rows(Df_1, Df_2)
#> # A tibble: 5 x 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     0
#> 2     2     7     2
#> 3     3     1     7
#> 4     1     5     6
#> 5     2     7     7
```

## Combining data frames with *bind*

- The Df\_1 and Df\_3 data frames have the same number of observations and so can be stacked side by side with a `bind_cols` operation.

```
> bind_cols(Df_1, Df_3)
#> # A tibble: 3 x 6
#>       x     y     z     a b     c
#>   <dbl> <dbl> <dbl> <dbl> <chr> <lgl>
#> 1     1     2     0     5 a    TRUE
#> 2     2     7     2     6 b    TRUE
#> 3     3     1     7     1 c    FALSE
```

## Combining data frames with *join*

- ▶ A *join* operation is a common operation in relational databases using SQL.
- ▶ The `blp_df` has a variable `spell` that gives the identity of the stimulus. In a separate file, `blp-stimuli.csv` file, we have three additional variables for these stimuli.

```
> stimuli <- read_csv('data/blp_stimuli.csv')
> stimuli
#> # A tibble: 55,865 x 4
#>   spell    old20    bnc    subtlex
#>   <chr>    <dbl> <dbl>    <dbl>
#> 1 a/c      1.95    14      0
#> 2 aas      1.55     9      1
#> 3 aback    1.85   327     15
#> 4 abaft     2      8      2
#> 5 aband    1.95     0      0
#> 6 abase     1.7     6      2
#> 7 abased    1.75     6      0
#> 8 abashed   1.85    57      0
#> 9 abate     1.75    69      5
```

## Combining data frames with join

- We can join these two data frames with `inner_join`.

```
> inner_join(blp_df, stimuli)
#> # A tibble: 1,000 x 10
#>   participant lex    spell    resp    rt prev.rt rt.raw old20
#>   <dbl> <chr> <chr>    <chr> <dbl>    <dbl> <dbl> <dbl>
#> 1      20 N    staud    N    977      511    977  1.85
#> 2       9 N    dinbuss N    565      765    565  2.9
#> 3      47 N    snilli~ N    562      496    562  1.8
#> 4     103 N    gancens N    572      656    572  2.3
#> 5      45 W    filled  W    659      981    659  1.45
#> 6      73 W    journa~ W    538     1505    538  2.7
#> 7      24 W    apache  W    626      546    626  2.45
#> 8      11 W    flake    W    566      717    566  1.5
#> 9      32 W    reliefs W    922     1471    922  2.25
#> 10     96 N    sarves  N    555      806    555  1.65
#> # ... with 990 more rows
```

## Reshaping data with *tidyr*'s *gather* and *spread*

- Consider the following data frame, which is in a *wide* format.

```
> (recall_df <- read_csv('data/repeated_measured_a.csv'))  
#> # A tibble: 5 x 4  
#>   Subject    Neg    Neu    Pos  
#>   <chr>    <dbl> <dbl> <dbl>  
#> 1 Faye      26     12    42  
#> 2 Jason     29      8    35  
#> 3 Jim       32     15    45  
#> 4 Ron       22     10    38  
#> 5 Victor    30     13    40
```



## Reshaping data with *tidy*'s *gather* and *spread*

- We can convert it to a *long* format as follows:

```
> library(tidyr)
> gather(recall_df, key=condition, value=score, Neg:Pos)
#> # A tibble: 15 x 3
#>   Subject condition score
#>   <chr>      <chr>    <dbl>
#> 1 Faye      Neg        26
#> 2 Jason     Neg        29
#> 3 Jim       Neg        32
#> 4 Ron       Neg        22
#> 5 Victor   Neg        30
#> 6 Faye      Neu        12
#> 7 Jason     Neu         8
#> 8 Jim       Neu        15
#> 9 Ron       Neu        10
#> 10 Victor   Neu        13
#> 11 Faye     Pos        42
#> 12 Jason    Pos        35
#> 13 Jim      Pos        45
```

## Reshaping data with *tidyr's* *gather* and *spread*

- The inverse of a *gather* is a *spread*. To do this, we supply the key and value variables, as defined in the *gather* operation.

```
> recall_long_df <- gather(recall_df, key=condition, value=score)
```

```
> spread(recall_long_df, key=condition, value=score)
```

```
#> # A tibble: 5 x 4
```

```
#>   Subject    Neg    Neu    Pos
```

```
#>   <chr>    <dbl> <dbl> <dbl>
```

```
#> 1 Faye      26     12     42
```

```
#> 2 Jason     29      8     35
```

```
#> 3 Jim       32     15     45
```

```
#> 4 Ron       22     10     38
```

```
#> 5 Victor    30     13     40
```

## Plots and data visualization

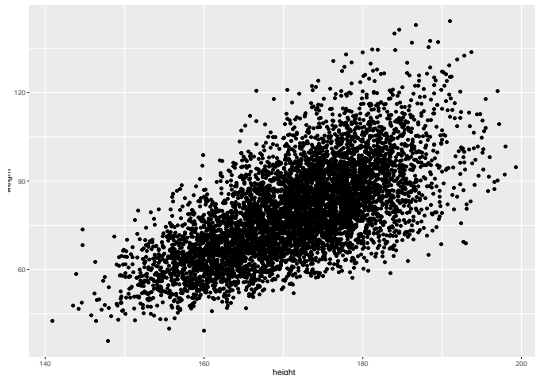
- ▶ The best way to data visualization in R is with ggplot2.

```
> library(ggplot2)
```

- ▶ ggplot2 is package whose main function is ggplot.
- ▶ ggplot is a *layered* plotting system where we map variables to aesthetic properties of a graphic and then add layers.

# Scatterplot

```
> weight_df <- read_csv('data/weight.csv')  
>  
> ggplot(weight_df,  
+       aes(x = height, y = weight)  
+ ) + geom_point()
```



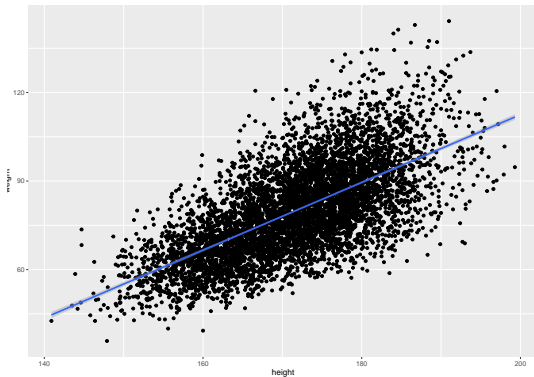
## Scatterplot with *gender* indicated by colour

```
> ggplot(weight_df,  
+         aes(x = height, y = weight, col = gender)  
+ ) + geom_point()
```



## Scatterplot with line of best fit

```
> ggplot(weight_df,  
+         aes(x = height, y = weight))  
+ ) + geom_point() +  
+   stat_smooth(method = 'lm')
```



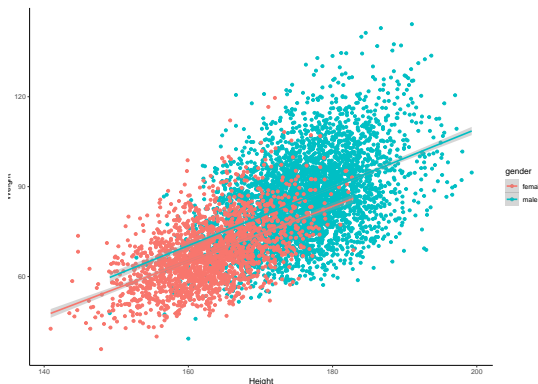
## Scatterplot with line of best fit, for each value of gender

```
> ggplot(weight_df,  
+         aes(x = height, y = weight, col = gender)  
+ ) + geom_point() + stat_smooth(method = 'lm')
```



## Changing style of a plot

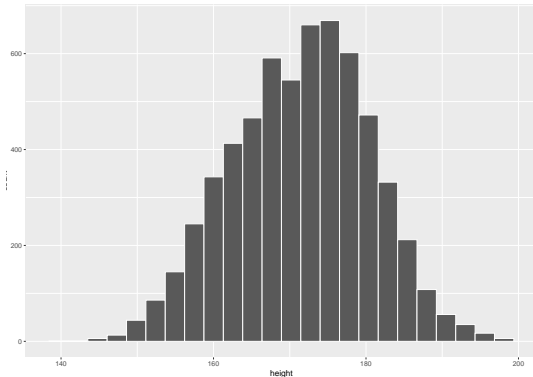
```
> ggplot(weight_df,  
+         aes(x = height, y = weight, col = gender)  
+ ) + geom_point() + stat_smooth(method = 'lm') +  
+   xlab('Height') +  
+   ylab('Weight') +  
+   theme_classic()
```





# Histograms

```
> ggplot(weight_df,  
+         aes(x = height)  
+ ) + geom_histogram(col = 'white', binwidth = 2.54)
```



# Histograms

- A *stacked* histogram

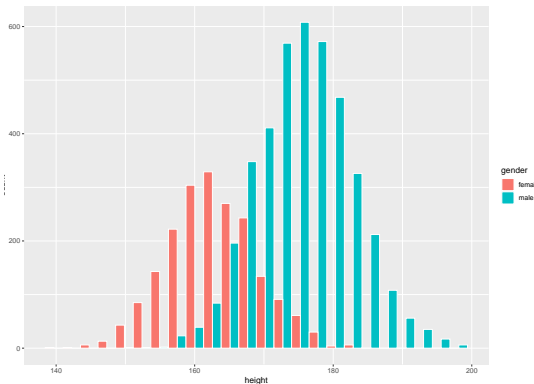
```
> ggplot(weight_df,  
+       aes(x = height, fill = gender)  
+ ) + geom_histogram(col = 'white', binwidth = 2.54)
```



# Histograms

- A *dodged* histogram

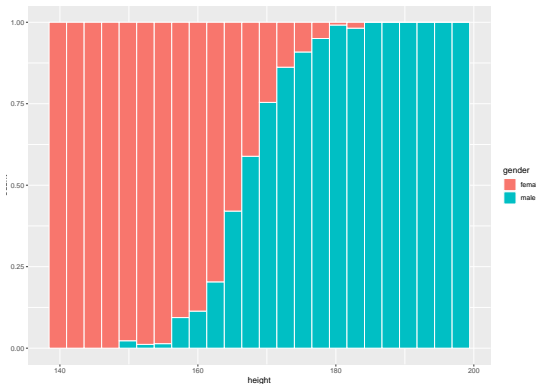
```
> ggplot(weight_df,  
+       aes(x = height, fill = gender)  
+ ) + geom_histogram(col = 'white', binwidth = 2.54,  
+                    position = 'dodge')
```



# Histograms

- A *filled* histogram

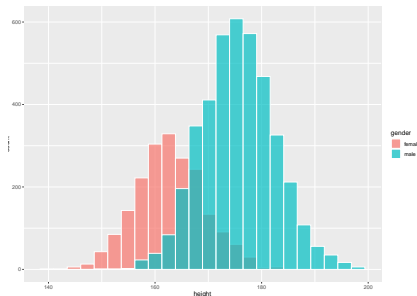
```
> ggplot(weight_df,  
+       aes(x = height, fill = gender)  
+ ) + geom_histogram(col = 'white', binwidth = 2.54,  
+                    position = 'fill')
```



# Histograms

- A *identity* histogram

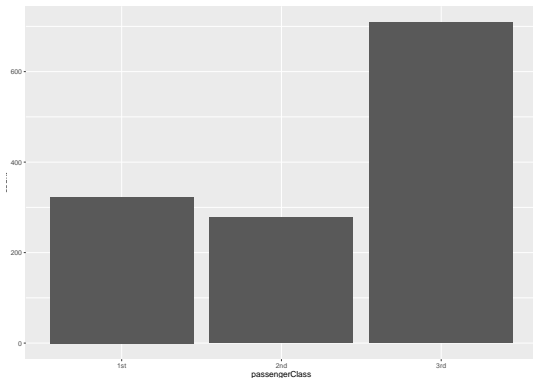
```
> ggplot(weight_df,  
+       aes(x = height, fill = gender)  
+ ) + geom_histogram(col = 'white', binwidth = 2.54,  
+                    position = 'identity', alpha = 0.7)
```



# Barplots

- *Barplots* are used for categorical variables.

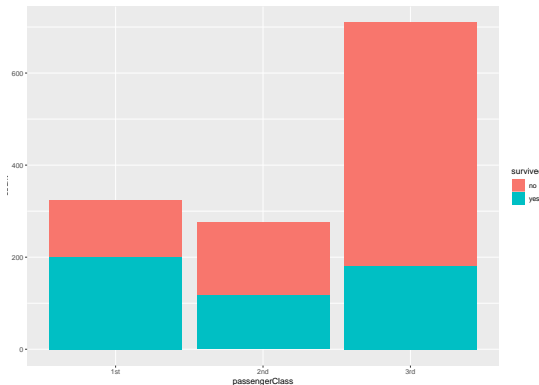
```
> titanic_df <- read_csv('data/TitanicSurvival.csv') %>% select(  
> ggplot(titanic_df,  
+       mapping = aes(x = passengerClass)  
+ ) + geom_bar()
```



# Barplots

- A *stacked* bar plot

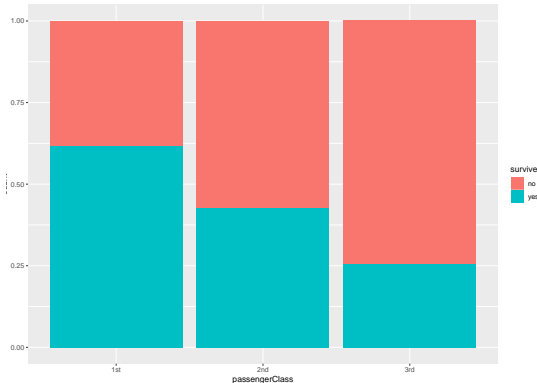
```
> ggplot(titanic_df,  
+       mapping = aes(x = passengerClass, fill=survived)  
+ ) + geom_bar()
```



# Barplots

- A *filled* bar plot

```
> ggplot(titanic_df,  
+       mapping = aes(x = passengerClass, fill=survived)  
+ ) + geom_bar(position = 'fill')
```

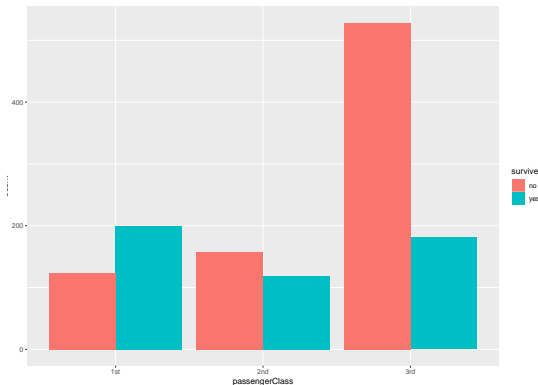




# Barplots

- A *dodged* bar plot

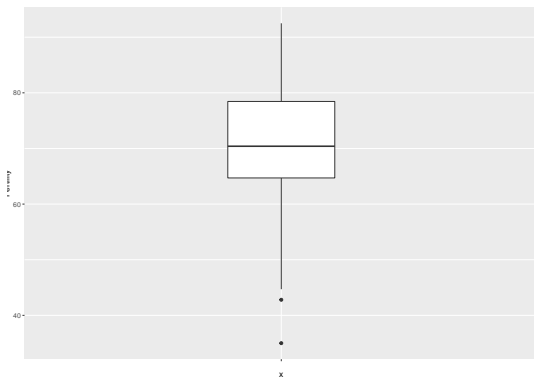
```
> ggplot(titanic_df,  
+       mapping = aes(x = passengerClass, fill=survived)  
+ ) + geom_bar(position = 'dodge')
```



# Boxplots

- We can draw Tukey box plots with `geom_boxplot`.

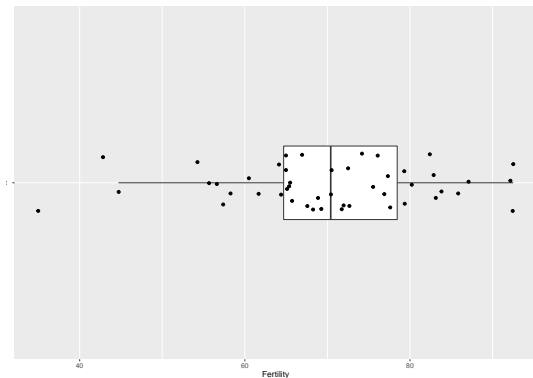
```
> swiss_df <- read_csv('data/swiss.csv')  
> ggplot(swiss_df,  
+       mapping = aes(x = '', y = Fertility)  
+ ) + geom_boxplot(width = 0.25)
```



# Boxplots

- We can *jitter* etc.

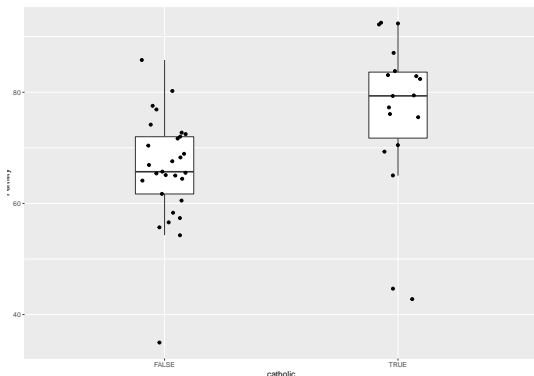
```
> ggplot(swiss_df,  
+       mapping = aes(x = '', y = Fertility)  
+ ) + geom_boxplot(width = 0.25, outlier.shape = NA) +  
+   geom_jitter(width = 0.1) +  
+   coord_flip()
```



# Boxplots

- We can have more than one boxplot.

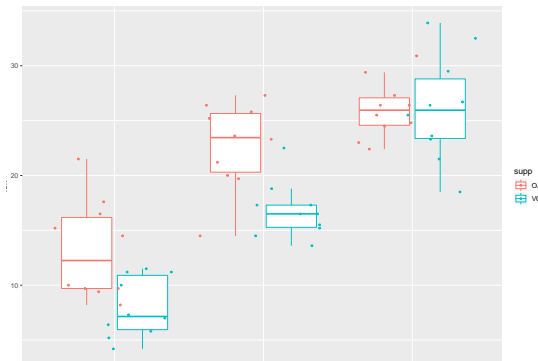
```
> ggplot(swiss_df,
+       mapping = aes(x = catholic, y = Fertility, )
+ ) + geom_boxplot(width = 0.25, outlier.shape = NA) +
+       geom_jitter(width = 0.1)
```



# Boxplots

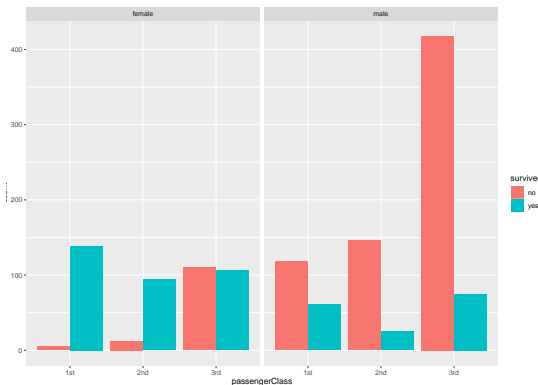
- We can group by multiple variables at once.

```
> toothgrowth_df <- read_csv('data/toothgrowth.csv')
> toothgrowth_df %>%
+   ggplot(mapping = aes(x = dose, y = len, colour = supp))
+     ) + geom_boxplot(outlier.shape = NA, varwidth = T) +
+       geom_jitter(position = position_jitterdodge(0.5),
+         size = 0.75) +
+       scale_x_discrete(limits=c('low', 'medium', 'high'))
```



## Facet plots

```
> ggplot(titanic_df,  
+       mapping = aes(x = passengerClass, fill = survived)  
+ ) + geom_bar(position = 'dodge') +  
+   facet_wrap(~sex)
```



## Facet plots

```
> sleepstudy_df <- read_csv("data/sleepstudy.csv",  
+                             col_types = cols(Subject = col_character()),  
+ )  
>  
> ggplot(sleepstudy_df,  
+         mapping = aes(x = Days, y = Reaction, colour = Subject))  
+ ) + geom_point() +  
+   geom_smooth(method = 'lm', se = F) +  
+   facet_wrap(~Subject) +  
+   theme(legend.position = 'none')
```

