Exploring data #1

Tidyverse

The “tidyverse”

You will also see code that uses functions like `read.csv` and `read.table` to read in flat files. These are from base R. The `readr` functions are very similar, but have some more sensible defaults, including in determining column classes.

Compared to the `read.table` family of functions, the `read_*` functions:

- Work better with large datasets: faster, includes progress bar
- Have more sensible defaults (e.g., characters default to characters, not factors)

The “tidyverse”

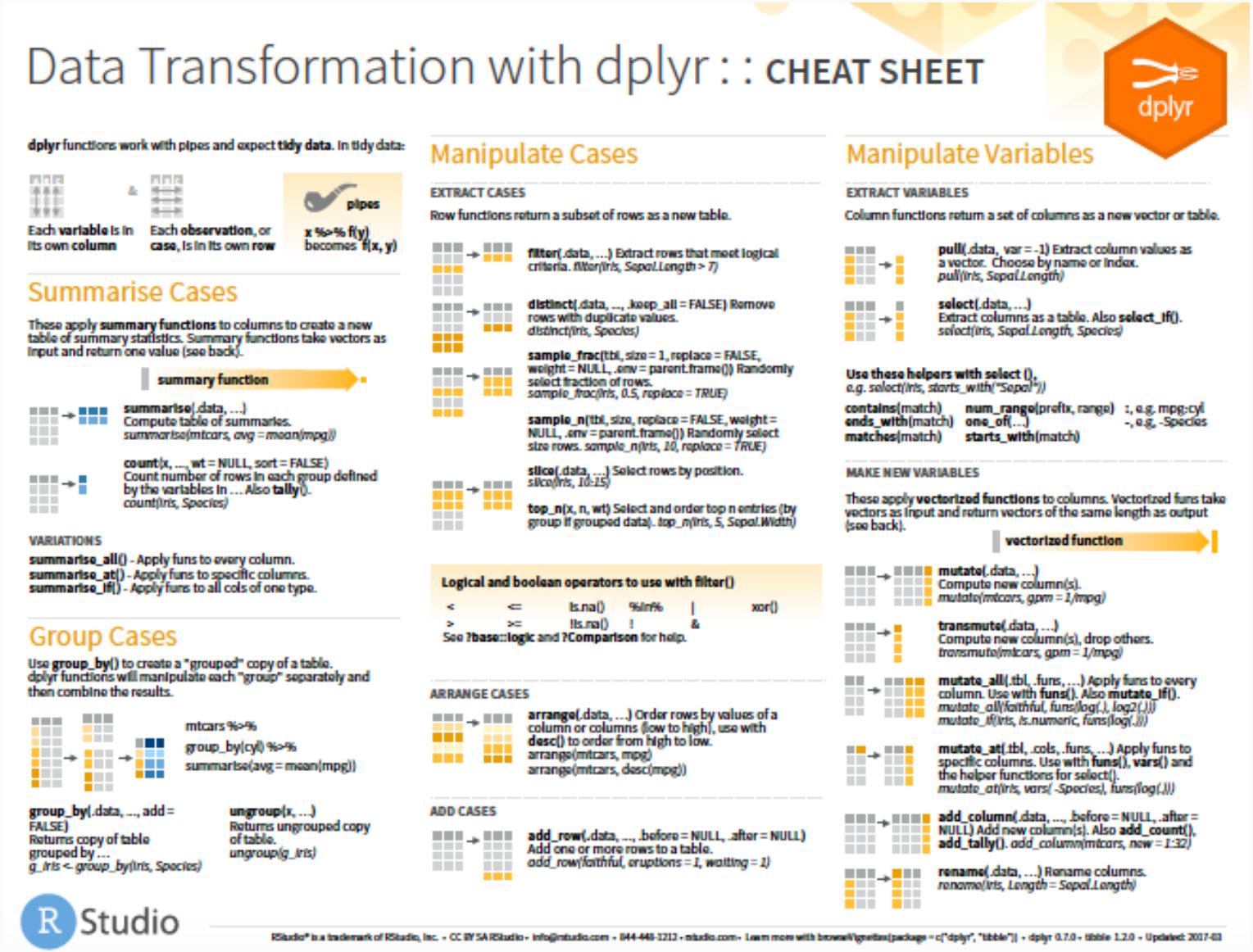
The `readr` package is part of the “tidyverse”– a collection of recent and developing packages for R, many written by Hadley Wickham.

The “tidyverse”

"A giant among data nerds"

https://priceonomics.com/hadley-wickham-the-man-who-revolutionized-r/

Cheatsheets



Data Transformation with dplyr : : CHEAT SHEET

dplyr

dplyr functions work with pipes and expect **tidy data**. In tidy data:

Each **variable** is in its own **column** & Each **observation**, or **case**, is in its own **row**

pipes
x %>% f(y) becomes f(x, y)

Summarise Cases

These apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).

summary function

summarise(.data, ...)
Compute table of summaries.
summarise(mtcars, avg = mean(mpg))

count(x, ..., wt = NULL, sort = FALSE)
Count number of rows in each group defined by the variables in ... Also **tally**().
count(iris, Species)

VARIATIONS

summarise_all() - Apply funs to every column.
summarise_at() - Apply funs to specific columns.
summarise_if() - Apply funs to all cols of one type.

Group Cases

Use **group_by**() to create a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.

```
mtcars %>%
group_by(cyl) %>%
summarise(avg = mean(mpg))
```

group_by(.data, ..., add = FALSE)
Returns copy of table grouped by ...
g_iris <- group_by(iris, Species)

ungroup(x, ...)
Returns ungrouped copy of table.
ungroup(g_iris)

Manipulate Cases

EXTRACT CASES

Row functions return a subset of rows as a new table.

filter(.data, ...) Extract rows that meet logical criteria. *filter(iris, Sepal.Length > 7)*

distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values.
distinct(iris, Species)

sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select fraction of rows.
sample_frac(iris, 0.5, replace = TRUE)

sample_n(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select size rows. *sample_n(iris, 10, replace = TRUE)*

slice(.data, ...) Select rows by position.
slice(iris, 10:15)

top_n(x, n, wt) Select and order top n entries (by group if grouped data). *top_n(iris, 5, Sepal.Width)*

Logical and boolean operators to use with filter()

<	<=	is.na()	%in%	|	xor()
>	>=	!is.na()	!	&	

See **?base::logic** and **?Comparison** for help.

ARRANGE CASES

arrange(.data, ...) Order rows by values of a column or columns (low to high), use with **desc**() to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

ADD CASES

add_row(.data, ..., .before = NULL, .after = NULL) Add one or more rows to a table.
add_row(faithful, eruptions = 1, waiting = 1)

Manipulate Variables

EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.

pull(.data, var = -1) Extract column values as a vector. Choose by name or index.
pull(iris, Sepal.Length)

select(.data, ...)
Extract columns as a table. Also **select_if**().
select(iris, Sepal.Length, Species)

Use these helpers with select (),
e.g. select(iris, starts_with("Sepal"))

contains(match)	**num_range**(prefix, range)	:, e.g. mpg:cyl
ends_with(match)	**one_of**(...)	-, e.g. -Species
matches(match)	**starts_with**(match)	

MAKE NEW VARIABLES

These apply **vectorized functions** to columns. Vectorized funs take vectors as input and return vectors of the same length as output (see back).

vectorized function

mutate(.data, ...)
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

transmute(.data, ...)
Compute new column(s), drop others.
transmute(mtcars, gpm = 1/mpg)

mutate_all(.tbl, .funs, ...) Apply funs to every column. Use with **funs**(). Also **mutate_if**().
mutate_all(faithful, funs(log(.), log2(.)))
mutate_if(iris, is.numeric, funs(log(.)))

mutate_at(.tbl, .cols, .funs, ...) Apply funs to specific columns. Use with **funs**(), **vars**() and the helper functions for select().
mutate_at(iris, vars(-Species), funs(log(.)))

add_column(.data, ..., .before = NULL, .after = NULL) Add new column(s). Also **add_count**(), **add_tally**(). *add_column(mtcars, new = 1:32)*

rename(.data, ...) Rename columns.
rename(iris, Length = Sepal.Length)

RStudio

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more with browseVignettes(package = c("dplyr", "tibble")) • dplyr 0.7.0 • tibble 1.2.0 • Updated: 2017-03

Cheatsheets

RStudio has several very helpful **cheatsheets**. These are one-page sheets (front and back) that cover many of the main functions for a certain topic or task in R. These cheatsheets cover a lot of the main "tidyverse" functions.

You can access these directly from RStudio. Go to "Help" -> "Cheatsheets" and select the cheatsheet on the topic of interest.

You can find even more of these cheatsheets at https://www.rstudio.com/resources/cheatsheets/.

Data from R packages

Data from R packages

So far we've covered three ways to get data into R:

1. From flat files (either on your computer or online)
2. From files like SAS and Excel
3. From R objects (i.e., using `load()`)

Many R packages come with their own data, which is very easy to load and use.

Data from R packages

For example, the `faraway` package has a dataset called `worldcup` that you'll use today. To load it, use the `data()` function once you've loaded the package with the name of the dataset as its argument:

```
library(faraway)
data("worldcup")
```

Data from R packages

Unlike most data objects you'll work with, the data that comes with an R package will often have its own help file. You can access this using the ? operator:

```
?worldcup
```

Data from R packages

To find out all the datasets that are available in the packages you currently have loaded, run `data()` without an option inside the parentheses:

```
data()
```

To find out all of the datasets available within a certain package, run `data` with the argument `package`:

```
data(package = "faraway")
```

As a note, you can similarly use `library()`, without the name of a package, to list all of the packages you have installed that you could call with `library()`:

```
library()
```

`nepali` example data

For the example plots, I'll use a dataset in the `faraway` package called `nepali`. This gives data from a study of the health of a group of Nepalese children.

```
library(faraway)
data(nepali)
```

I'll be using functions from `dplyr` and `ggplot2` during the course, so I'll load those:

```
library(dplyr)
library(ggplot2)
```

`nepali` example data

For the `nepali` dataset, each observation is a single measurement for a child; there can be multiple observations per child.

I'll limit it to the columns with the child's id, sex, weight, height, and age, and I'll limit to each child's first measurement.

```
nepali <- nepali %>%
  # Limit to certain columns
  select(id, sex, wt, ht, age) %>%
  # Convert id and sex to factors
  mutate(id = factor(id),
         sex = factor(sex, levels = c(1, 2),
                      labels = c("Male", "Female"))) %>%
  # Limit to first obs. per child
  distinct(id, .keep_all = TRUE)
```

`nepali` example data

The first few rows of the data now looks like:

```
nepali %>%
  slice(1:4)
```

```
## # A tibble: 4 x 5
##   id     sex       wt    ht   age
##   <fct>  <fct>  <dbl> <dbl> <int>
## 1 120011 Male   12.8   91.2    41
## 2 120012 Female 14.9  104.     57
## 3 120021 Female  7.70  70.1     8
## 4 120022 Female 12.1   86.4    35
```

Logical vectors

Logical statements

Last week, you learned some about logical statements and how to use them with the `filter` function.

You can use *logical vectors*, created with these statements, for a lot of things. We'll review them and add some more details this week.

Logical vectors

A logical statement outputs a *logical vector*. This logical vector will be the same length as the original vector tested by the logical statement:

```
is_male <- nepali$sex == "Male"
length(nepali$sex)
```

```
## [1] 200
```

```
length(is_male)
```

```
## [1] 200
```

Logical vectors

Each element of the logical vector can only have one of three values (`TRUE`, `FALSE`, `NA`). The logical vector will have the value `TRUE` at any position where the original vector met the logical condition you tested, and `FALSE` anywhere else:

```
head(nepali$sex)
```

```
## [1] Male   Female Female Female Male   Male
## Levels: Male Female
```

```
head(is_male)
```

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

Logical vectors

Because the logical vector is the same length as the vector it's testing, you can add logical vectors to dataframes with `mutate`:

```
nepali <- nepali %>%
  mutate(is_male = sex == "Male") # Add column. Is obs. male?
nepali %>%
  slice(1:3)
```

```
## # A tibble: 3 x 6
##   id     sex       wt    ht   age is_male
##   <fct>  <fct>  <dbl> <dbl> <int> <lgl>
## 1 120011 Male   12.8   91.2    41 TRUE
## 2 120012 Female 14.9  104.     57 FALSE
## 3 120021 Female  7.70  70.1     8 FALSE
```

Logical vectors

As another example, you could add a column that is a logical vector of whether each child's first-measured height is over 100 centimeters:

```
nepali %>%
  mutate(very_tall = ht > 100) %>% # Is height over 100 cm?
  select(id, ht, very_tall) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 3
##   id        ht very_tall
##   <fct>  <dbl> <lgl>
## 1 120011  91.2 FALSE
## 2 120012 104.  TRUE
## 3 120021  70.1 FALSE
```

Logical vectors

You can "flip" a logical vector (i.e., change every `TRUE` to `FALSE` and vice-versa) using the bang operator (`!`):

```
nepali %>%
  mutate(very_tall = ht > 100,
         not_tall = !very_tall) %>%
  select(id, ht, very_tall, not_tall) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 4
##   id        ht very_tall not_tall
##   <fct>  <dbl> <lgl>     <lgl>
## 1 120011  91.2 FALSE     TRUE
## 2 120012 104.  TRUE      FALSE
## 3 120021  70.1 FALSE     TRUE
```

Logical vectors

You can do a few cool things now with this vector. For example, you can use it with the `filter` function to pull out just the rows where `is_male` is `TRUE`:

```
nepali %>%
  filter(is_male) %>%
  select(id, ht, wt, sex) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 4
##   id        ht    wt sex
##   <fct>  <dbl> <dbl> <fct>
## 1 120011  91.2 12.8  Male
## 2 120023  99.4 14.2  Male
## 3 120031  96.4 13.9  Male
## 4 120051  69.5  8.30 Male
## 5 120053  96.0 15.8  Male
```

Logical vectors

Or, with `!`, just the rows where `is_male` is `FALSE`:

```
nepali %>%
  filter(!is_male) %>%
  select(id, ht, wt, sex) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 4
##   id        ht    wt sex
##   <fct>  <dbl> <dbl> <fct>
## 1 120012 104.  14.9  Female
## 2 120021  70.1  7.70 Female
## 3 120022  86.4 12.1  Female
## 4 120052  83.6 11.8  Female
## 5 120061  78.5  8.70 Female
```

Logical vectors

All of the values in a logical vector are saved with a number underneath. Values of `TRUE` are saved as 1 and values of `FALSE` are saved as 0.

You can use `sum()` to get the sum of all values in a vector. Because logical vector values are linked with numerical values of 0 or 1, you can use `sum()` to find out how many males and females are in the dataset:

```
sum(nepali$is_male)
```

```
## [1] 107
```

```
sum(!nepali$is_male)
```

```
## [1] 93
```

In-course exercise

We'll take a break now to start the in-course exercise for this week (Sections 3.6.1 and 3.6.2).

Simple statistics

Simple statistics functions

We've looked at how to subset, arrange, and add to a dataframe. Next we'll look at how to summarizes a dataframe.

We'll start by looking at some simple statistics functions from base R, and then we'll look at how some of those functions can be used with the `summarize` function from the `dplyr` package to quickly get interesting summaries of data.

Simple statistics functions

Here are some simple statistics functions you will likely use often:

Function	Description
`range()`	Range (minimum and maximum) of vector
`min()`, `max()`	Minimum or maximum of vector
`mean()`, `median()`	Mean or median of vector
`table()`	Number of observations per level for a factor vector
`cor()`	Determine correlation(s) between two or more vectors
`summary()`	Summary statistics, depends on class

Simple statistic examples

All of these take, as the main argument, the vector(s) for which you want the statistic. If there are missing values in the vector, you'll need to add an option to say what to do when them (e.g., `na.rm` or `use="complete.obs"`– see help files).

```
mean(nepali$wt, na.rm = TRUE)
```

```
## [1] 10.18432
```

```
range(nepali$ht, na.rm = TRUE)
```

```
## [1]  52.4 104.1
```

```
table(nepali$sex)
```

```
##
##   Male Female
##    107     93
```

Simple statistic examples

The cor function can take two or more vectors. If you give it multiple values, it will give the correlation matrix for all the vectors.

```
cor(nepali$wt, nepali$ht, use = "complete.obs")
```

```
## [1] 0.9571535
```

```
cor(nepali[ , c("wt", "ht", "age")], use = "complete.obs")
```

```
##             wt        ht       age
## wt  1.0000000 0.9571535 0.8931195
## ht  0.9571535 1.0000000 0.9287129
## age 0.8931195 0.9287129 1.0000000
```

`summary()`: A bit of OOP

R supports object-oriented programming. This shows up with `summary()`. R looks to see what type of object it's dealing with, and then uses a method specific to that object type.

```
summary(nepali$wt)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    3.80    7.90   10.10   10.18   12.40   16.70
##    NA's
##      15
```

```
summary(nepali$sex)
```

```
##   Male Female
##    107     93
```

The `summarize` function

Within a "tidy" workflow, you can use the `summarize` function from the `dplyr` package to create summary statistics for a dataframe. This function inputs a dataframe and outputs a dataframe with the specified summary measures.

The `summarize` function

The basic format for using `summarize` is:

```
## Generic code
summarize(dataframe,
          summary_column_1 = function(existing_columns),
          summary_column_2 = function(existing_columns))
```

The output from `summarize` will be a dataframe with:

- One row (later we will look at using `summarize` within groups of data, and that will result in more rows)
- As many columns as you have defined summaries in the `summarize` function (the generic code above would result in two columns)

The summarize function

As an example, to summarize the nepali dataset to get the mean weight, median height, and minimum and maximum ages of children, you could run:

```
summarize(nepali,
          mean_wt = mean(wt, na.rm = TRUE),
          median_ht = median(ht, na.rm = TRUE),
          youngest = min(age, na.rm = TRUE),
          oldest = max(age, na.rm = TRUE))
```

```
##    mean_wt median_ht youngest oldest
## 1 10.18432        80        0     60
```

Notice that the output is one row (since the summary was on ungrouped data), with four columns (since we defined four summaries in the summarize function).

The summarize function

Because the first input to the summarize function is a dataframe, you can "pipe into" a summarize call. For example, we could have written the code on the previous slide as:

```
nepali %>%
  summarize(mean_wt = mean(wt, na.rm = TRUE),
            median_ht = median(ht, na.rm = TRUE),
            youngest = min(age, na.rm = TRUE),
            oldest = max(age, na.rm = TRUE))
```

As another note, because the output from summarize is also a dataframe, we could also "pipe into" another tidyverse function after running summarize.

The `summarize` function

There are some special functions that you can use with `summarize`:

Function	Description
`n()`	Number of elements in a vector
`n_distinct()`	Number of unique elements in a vector
`first()`	First value in a vector
`last()`	Last value in a vector

The `summarize` function

For example, the following call would give you the total number of observations in the dataset, the number of distinct values of age measured across all children, the ID of the first child included in the dataset, and the weight of the last child included in the dataset:

```
nepali %>%
  summarize(n_children = n(),
            n_distinct_ages = n_distinct(age),
            first_id = first(id),
            last_weight = last(wt))
```

```
##   n_children n_distinct_ages first_id last_weight
## 1        200              58   120011           5
```

Grouping and summarizing

Often, you'll want to get summaries of the data stratified by groups within the data. For example, in the Nepali dataset, you may want to get summaries by sex or by whether the child was short or tall.

To get grouped summaries of a dataframe, you can first use the `group_by` function from the `dplyr` package to "group" the dataset, and then when you run "summarize", it will be applied **by group** to the data.

Your final output from summarize will be a dataframe with:

- As many rows as there were unique groups in the grouping factor(s)
- As many columns as you have defined summaries in the `summarize` function (the generic code above would result in two columns), plus columns for each of the grouping factors

Grouping and summarizing

Without piping, the use of `group_by` and `summarize` looks like this:

```
# Generic code
summarize(group_by(dataframe,
                   grouping_factor_1, grouping_factor_2),
          summary_column_1 = function(existing_columns),
          summary_column_2 = function(existing_columns))
```

You can see that `group_by` is nested within the `summarize` call, because `group_by` must be applied to the dataframe before `summarize` is run if you want to get summaries by group.

Grouping and summarizing

This call tends to look much cleaner if you use piping. With piping, the generic call looks like:

```
# Generic code
dataframe %>%
  group_by(grouping_factor_1, grouping_factor_2) %>%
  summarize(summary_column_1 = function(existing_columns),
            summary_column_2 = function(existing_columns))
```

Grouping and summarizing

For example, in the Nepali dataset, say you want to get summaries by sex. You want to get the total number of children in each group, the mean weight, and the ID of the first child.

You can run:

```
nepali %>%
  group_by(sex) %>%
  summarize(n_children = n(),
            mean_wt = mean(wt, na.rm = TRUE),
            first_id = first(id))
```

```
## # A tibble: 2 x 4
##   sex    n_children mean_wt first_id
##   <fct>       <int>   <dbl> <fct>
## 1 Male          107   10.5  120011
## 2 Female         93    9.82 120012
```

Grouping and summarizing

```
nepali %>%
  group_by(sex) %>%
  summarize(n_children = n(),
            mean_wt = mean(wt, na.rm = TRUE),
            first_id = first(id))
```

```
## # A tibble: 2 x 4
##   sex    n_children mean_wt first_id
##   <fct>       <int>   <dbl> <fct>
## 1 Male          107   10.5  120011
## 2 Female         93    9.82 120012
```

Notice that the output is a dataframe with two rows (since there were two groups in the grouping factor) and four columns (one for the grouping factor, plus one for each of the summaries defined in the `summarize` function).

Grouping and summarizing

You can group by more than one variable. For example, to get summaries within groups divided by both sex and whether the child is tall ($>$ 100 cm) or not, you could run:

```
nepali %>%
  mutate(tall = ht > 100) %>%
  filter(!is.na(tall)) %>%
  group_by(sex, tall) %>%
  summarize(n_children = n(),
            mean_wt = mean(wt, na.rm = TRUE))
```

```
## # A tibble: 4 x 4
## # Groups:   sex [?]
##   sex    tall  n_children mean_wt
##   <fct>  <lgl>      <int>   <dbl>
## 1 Male   FALSE         94   10.2
## 2 Male   TRUE           5   15.2
## 3 Female FALSE         80    9.43
## 4 Female TRUE           6   15.1
```

In-course exercise

We'll take a break now to continue the in-course exercise for this week (Section 3.6.3).

More reading / practice

If you would like more reading and practice on what we've covered so far on transforming data, see chapter 5 of the "R for Data Science" book suggested at the start of the course.

As a reminder, that is available at:

http://r4ds.had.co.nz

Plots

Plots to explore data

Plots can be invaluable in exploring your data.

Today, we will focus on **useful**, rather than **attractive** graphs, since we are focusing on exploring rather than presenting data.

Next lecture, we will talk more about customization, to help you make more attractive plots that would go into final reports.

`ggplot` conventions

Here, we'll be using functions from the `ggplot2` library, so you'll need to install that package:

```
library(ggplot2)
```

The basic steps behind creating a plot with `ggplot2` are:

1. Create an object of the `ggplot` class, typically specifying the **data** to be shown in the plot;
2. Add on (using +) one or more **geoms**, specifying the **aesthetics** for each; and
3. Add on (using +) other elements to create and customize the plot (e.g., add layers to customize scales or themes or to add facets).

Note: To avoid errors, end lines with +, don't start lines with it.

Plot data

The `ggplot` function requires you to input a dataframe with the data you will plot. All the columns in that dataframe can be mapped to specific aesthetics within the plot.

```
nepali %>%
  slice(1:3)
```

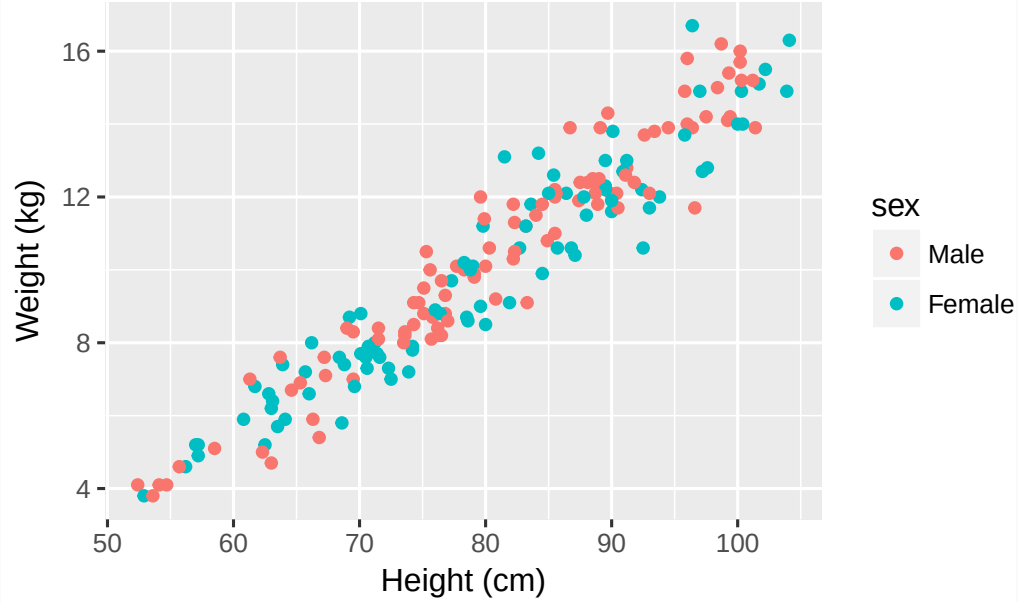
```
## # A tibble: 3 x 6
##   id     sex       wt    ht   age is_male
##   <fct>  <fct>  <dbl> <dbl> <int> <lgl>
## 1 120011 Male   12.8   91.2    41 TRUE
## 2 120012 Female 14.9  104.     57 FALSE
## 3 120021 Female  7.70  70.1     8 FALSE
```

For example, if we input the `nepali` dataframe, we would be able to create a scatterplot that shows each child's initial height on the x-axis, weight on the y-axis, and sex by the color of the point.

Plot aesthetics

Aesthetics are plotting elements that can show certain elements of the data.

For example, you may want to create a scatterplot where color shows gender, x-position shows height, and y-position shows weight.



Plot aesthetics

In the previous graph, the mapped aesthetics are color, x, and y. In the ggplot code, all of these aesthetic mappings will be specified within an aes call, which will be nested in another call in the ggplot pipeline.

Aesthetic	ggplot abbreviation	nepali column
x-axis position	x =	ht
y-axis position	y =	wt
color	color =	sex

This is how these mappings will be specified in an aes call:

```
# Note: This code should not be run by itself.
# It will eventually be nested in a ggplot call.
aes(x = ht, y = wt, color = sex)
```

Plot aesthetics

Here are some common plot aesthetics you might want to specify:

Code	Description
`x`	Position on x-axis
`y`	Position on y-axis
`shape`	Shape
`color`	Color of border of elements
`fill`	Color of inside of elements
`size`	Size
`alpha`	Transparency (1: opaque; 0: transparent)
`linetype`	Type of line (e.g., solid, dashed)

Geoms

You will add **geoms** that create the actual geometric objects on the plot. For example, a scatterplot has "points" geoms, since each observation is displayed as a point.

There are `geom_*` functions that can be used to add a variety of geoms. The function to add a "points" geom is `geom_point`.

We just covered three plotting elements:

- Data
- Aesthetics
- Geoms

These are three elements that you will almost always specify when using `ggplot`, and they are sufficient to create a number of basic plots.

Creating a ggplot object

You can create a scatterplot using `ggplot` using the following code format:

```
## Generic code
ggplot(data = dataframe) +
  geom_point(mapping = aes(x = column_1, y = column_2,
                           color = column_3))
```

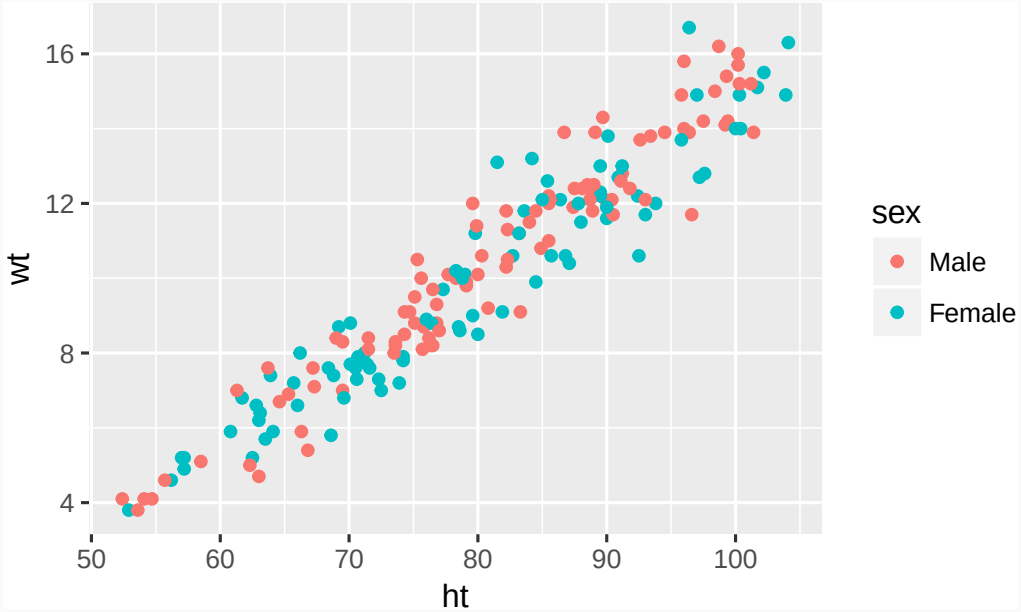
Notice that:

1. The `ggplot` call specifies the **dataframe** with the data you want to plot
2. A **geom** is added using the appropriate `geom_*` function for a scatterplot (`geom_point`).
3. The mappings between columns in the dataframe and **aesthetics** of the geom is specified within an `aes` call in the `mapping` argument of the `geom_*` function call.
4. The `aes` call includes mappings to two aesthetics that are required from the `geom_point` geom (`x` and `y`) and one that is optional (`color`).

Creating a ggplot object

Let's put these ideas together to write the code to create a scatterplot for our example data:

```
ggplot(data = nepali) +
geom_point(mapping = aes(x = ht, y = wt, color = sex))
```



Adding geoms

There are a number of different `geom_*` functions you can use to add geoms to a plot. They are divided between geoms that directly map the data to an aesthetic and those that show some summary or statistic of the data.

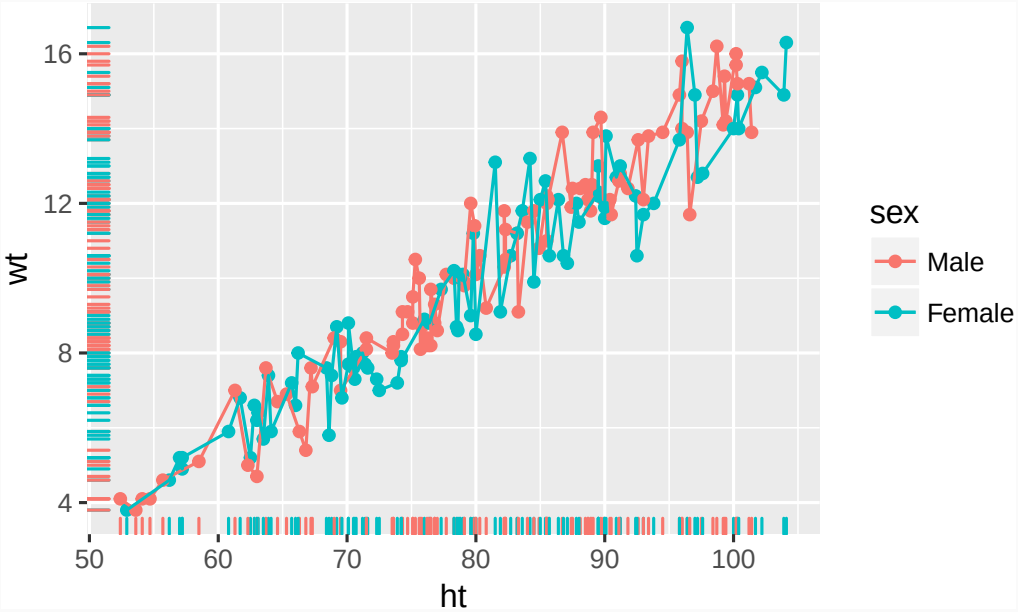
Some of the most common direct-mapping geoms are:

Geom(s)	Description
`geom_point`	Points in 2-D (e.g. scatterplot)
`geom_line`, `geom_path`	Connect observations with a line
`geom_abline`	A line with a certain intercept and slope
`geom_hline`, `geom_vline`	A horizontal or vertical line
`geom_rug`	A rug plot
`geom_label`, `geom_text`	Text labels

Creating a ggplot object

You can add several geoms to the same plot as layers:

```
ggplot(data = nepali) +
geom_point(mapping = aes(x = ht, y = wt, color = sex)) +
geom_line(mapping = aes(x = ht, y = wt, color = sex)) +
geom_rug(mapping = aes(x = ht, y = wt, color = sex))
```



Creating a ggplot object

You may have noticed that all of these geoms use the same aesthetic mappings (height to x-axis position, weight to y-axis position, and sex to color). To save time, you can specify the aesthetic mappings in the first `ggplot` call. These mappings will then be the default for any of the added geoms.

```
ggplot(data = nepali,
       mapping = aes(x = ht, y = wt, color = sex)) +
  geom_point() +
  geom_line() +
  geom_rug()
```

Creating a ggplot object

Because the first argument of the `ggplot` call is a dataframe, you can also "pipe into" a `ggplot` call:

```
nepali %>%
  ggplot(aes(x = ht, y = wt, color = sex)) +
  geom_point() +
  geom_line() +
  geom_rug()
```

In-course exercise

We'll take a break now to continue the in-course exercise for this week (Section 3.6.4).

Plot aesthetics

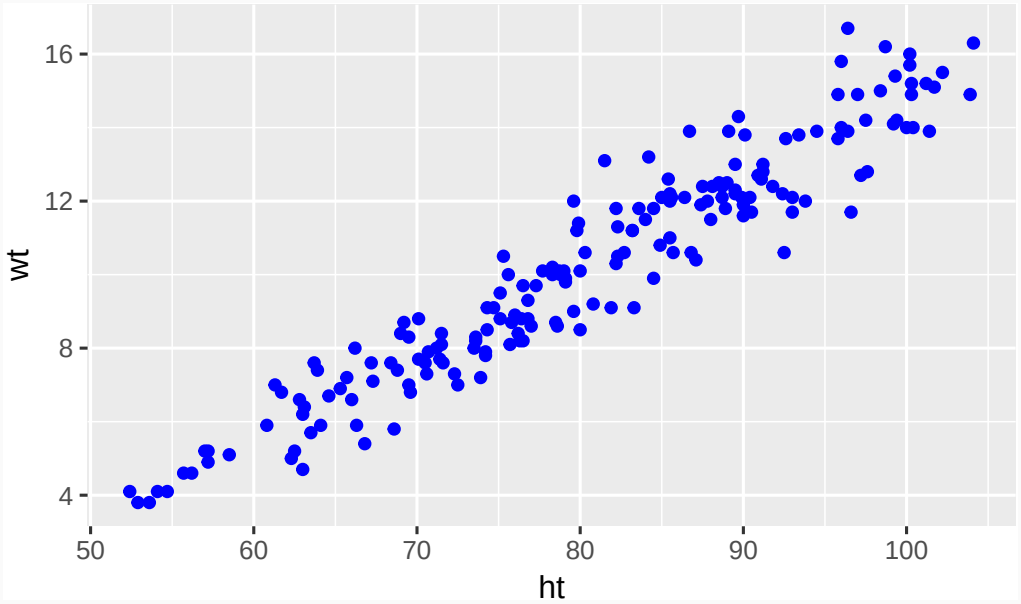
Which aesthetics you must specify in the `aes` call depend on which geom you are adding to the plot.

You can find out the aesthetics you can use for a geom in the "Aesthetics" section of the geom's help file (e.g., `?geom_point`).

Required aesthetics are in bold in this section of the help file and optional ones are not.

Constant aesthetics

Instead of mapping an aesthetic to an element of your data, you can use a constant value for the aesthetic. For example, you may want to make all the points blue, rather than having color map to gender:

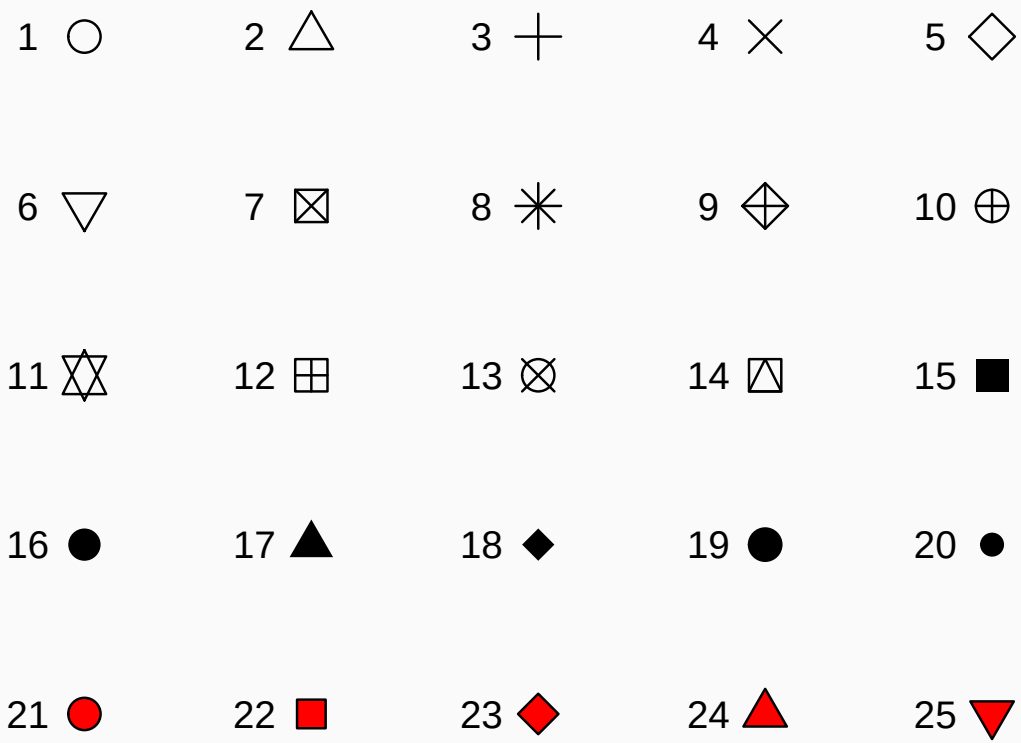


In this case, you can define that aesthetic as a constant for the geom, outside of an `aes` statement.

Constant aesthetics

For example, you may want to change the shape of the points in a scatterplot from their default shape, but not map them to a particular element of the data.

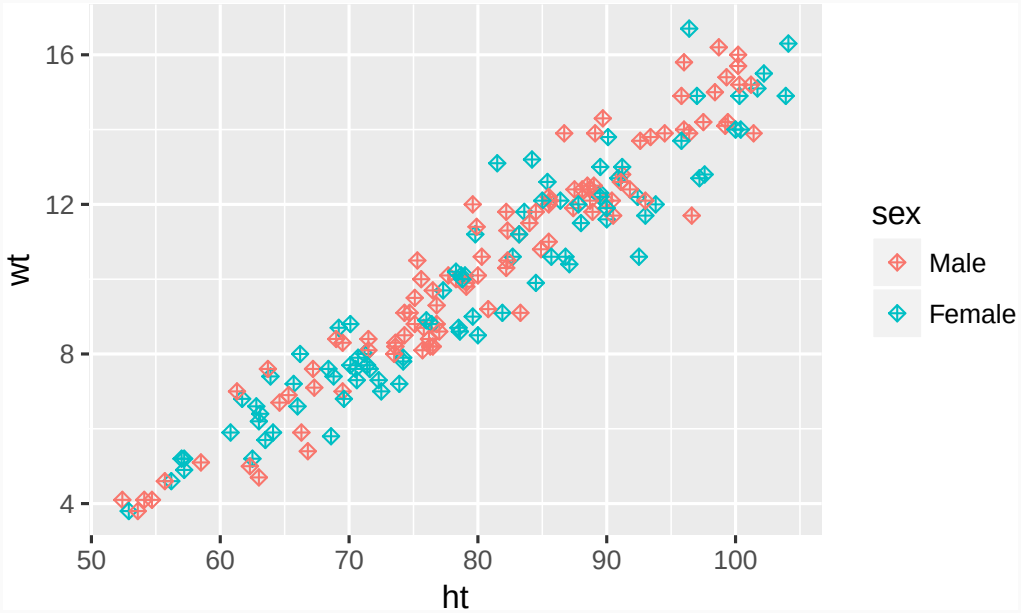
In R, you can specify point shape with a number. Here are the shapes that correspond to the numbers 1 to 25:



Constant aesthetics

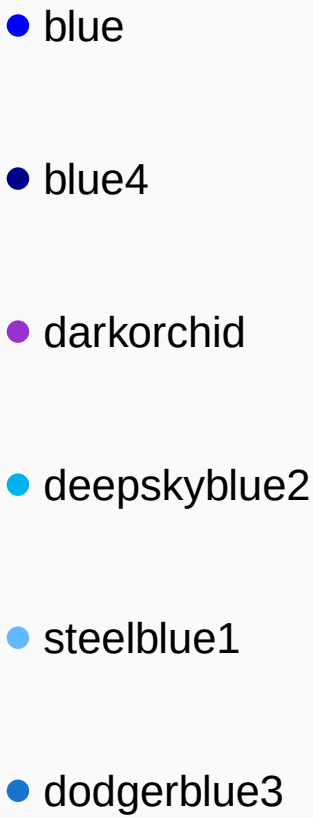
Here is an example of mapping point shape to a constant value other than the default:

```
ggplot(data = nepali) +
  geom_point(mapping = aes(x = ht, y = wt, color = sex),
             shape = 9)
```



Constant aesthetics

R has character names for different colors. For example:

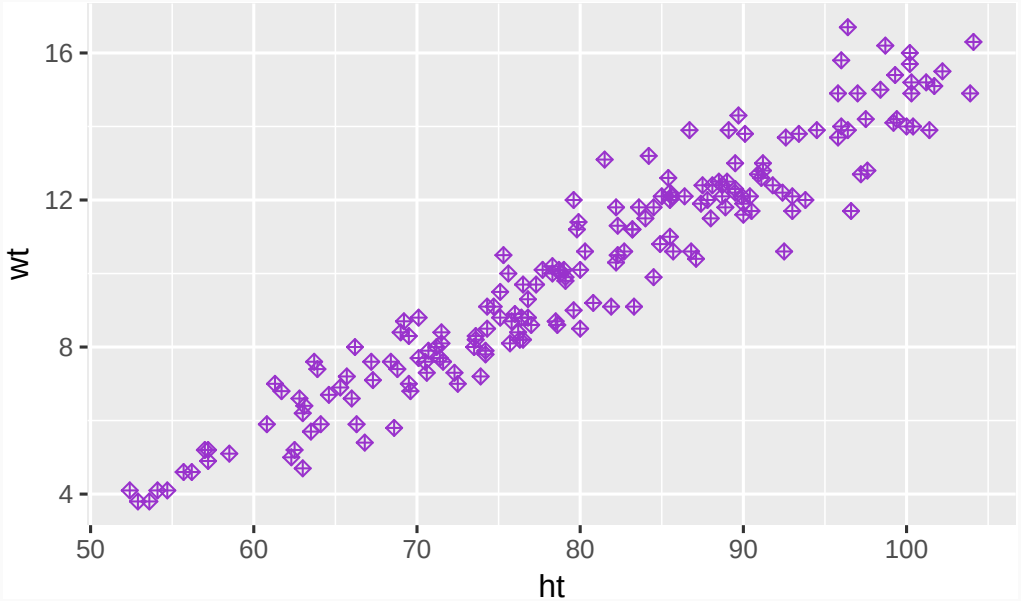


Google "R colors" and search the images to find links to listings of different R colors.

Constant aesthetics

Here is an example of mapping point shape and color to constant values other than the defaults:

```
ggplot(data = nepali) +
  geom_point(mapping = aes(x = ht, y = wt),
             shape = 9,
             color = "darkorchid")
```



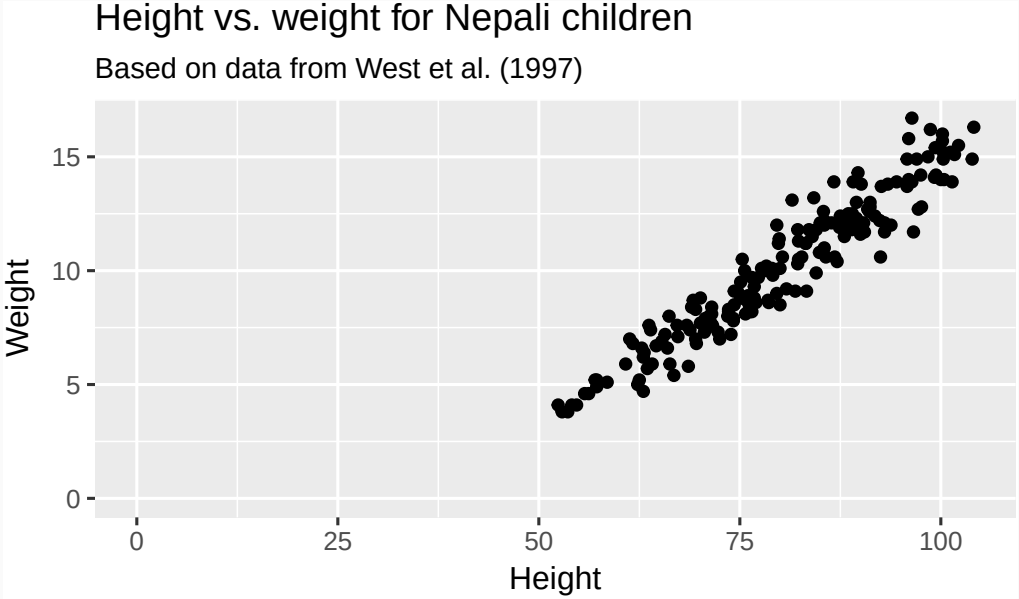
Useful plot additions

There are also a number of elements that you can add onto a `ggplot` object using +. A few very frequently used ones are:

Element	Description
`ggtitle`	Plot title
`xlab`, `ylab`, `labs`	x- and y-axis labels
`xlim`, `ylim`	Limits of x- and y-axis
`expand_limits`	Include a value in a range

Useful plot additions

```
ggplot(data = nepali) +
  geom_point(mapping = aes(x = ht, y = wt)) +
  labs(x = "Height", y = "Weight") +
  ggtitle("Height vs. weight for Nepali children",
          subtitle = "Based on data from West et al. (1997)") +
  expand_limits(x = 0, y = 0)
```



In-course exercise

We'll take a break now to continue the in-course exercise for this week (Section 3.6.5).

Adding geoms

There are a number of different `geom_*` functions you can use to add geoms to a plot. They are divided between geoms that directly map the data to an aesthetic and those that show some summary or statistic of the data.

Some of the most common "statistical" geoms are:

Geom(s)	Description
`geom_histogram`	Show distribution in 1-D
`geom_hex`, `geom_density`	Show distribution in 2-D
`geom_col`, `geom_bar`	Create bar charts
`geom_boxplot`, `geom_dotplot`	Create boxplots and related plots
`geom_smooth`	Add a fitted line to a scatterplot

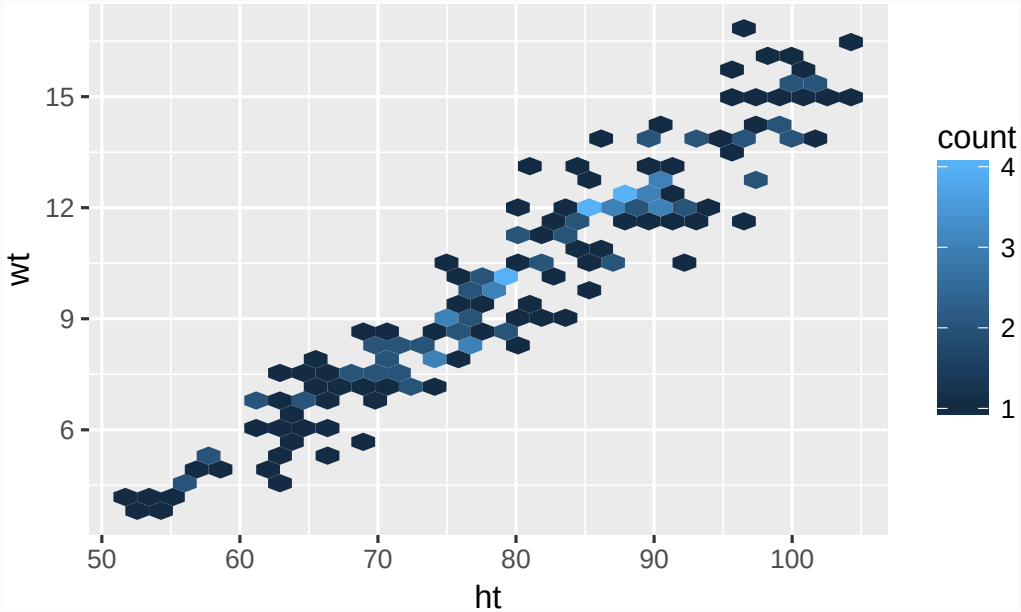
Adding geoms

These "statistical" geoms all input the original data and perform some calculations on that data to determine how to plot the final geom. Often, this calculation involves some kind of summarization.

For example, the geom for a hexagonal 2-D heatmap (`geom_hex`) divides the data into an evenly-sized set of hexagons and then calculates the number of points in each hexagon to provide a 2-D visualization of how the data is distributed.

Adding geoms

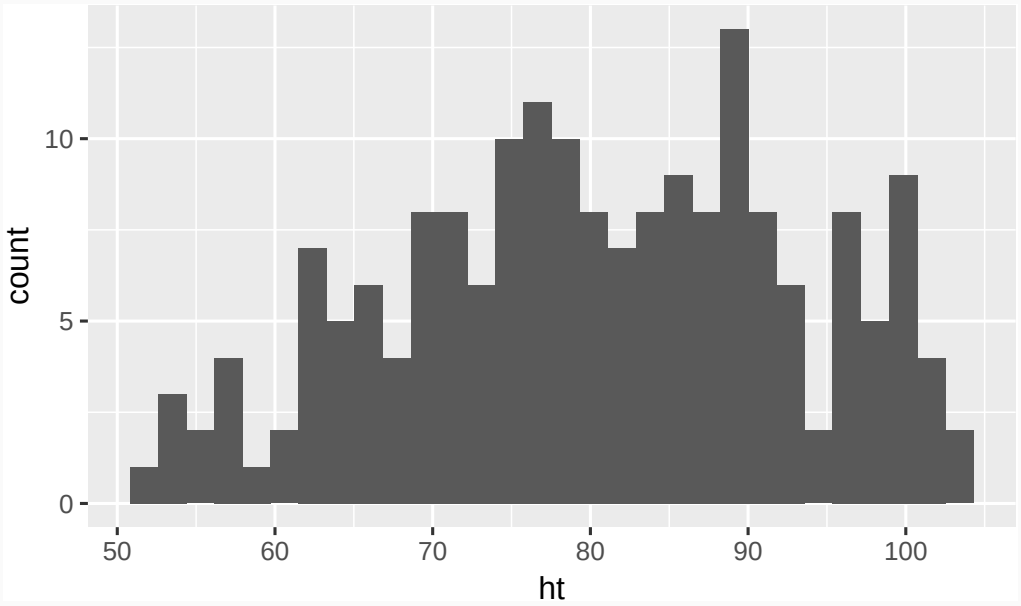
```
ggplot(data = nepali) +
  geom_hex(aes(x = ht, y = wt))
```



Adding geoms

A histogram geom is a similar idea, but only gives the distribution across one variable:

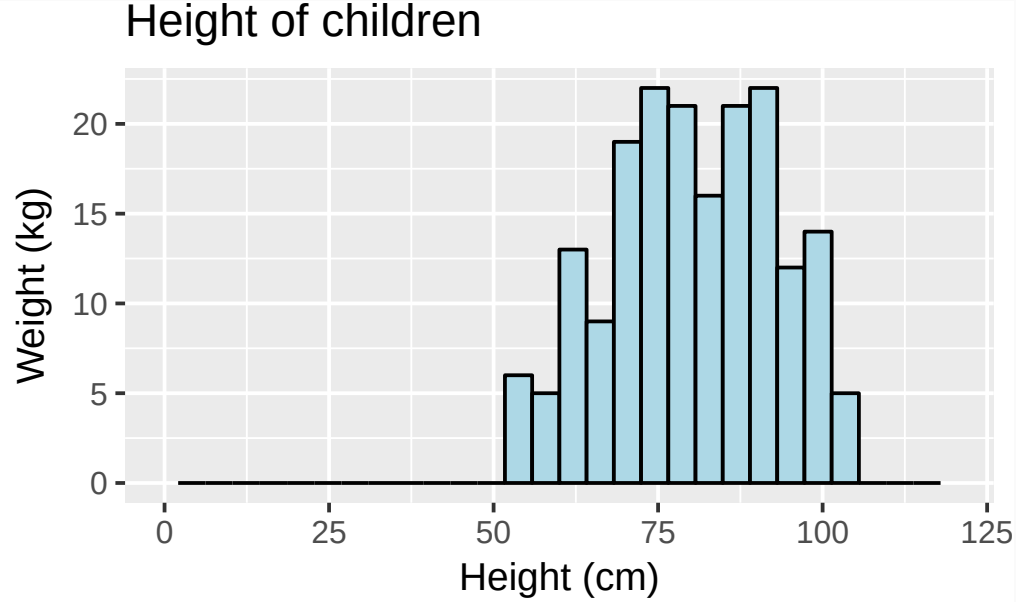
```
ggplot(data = nepali) +
  geom_histogram(aes(x = ht))
```



Histogram example

You can add some elements to the histogram, like `ggtitle`, `labs`, and `xlim`:

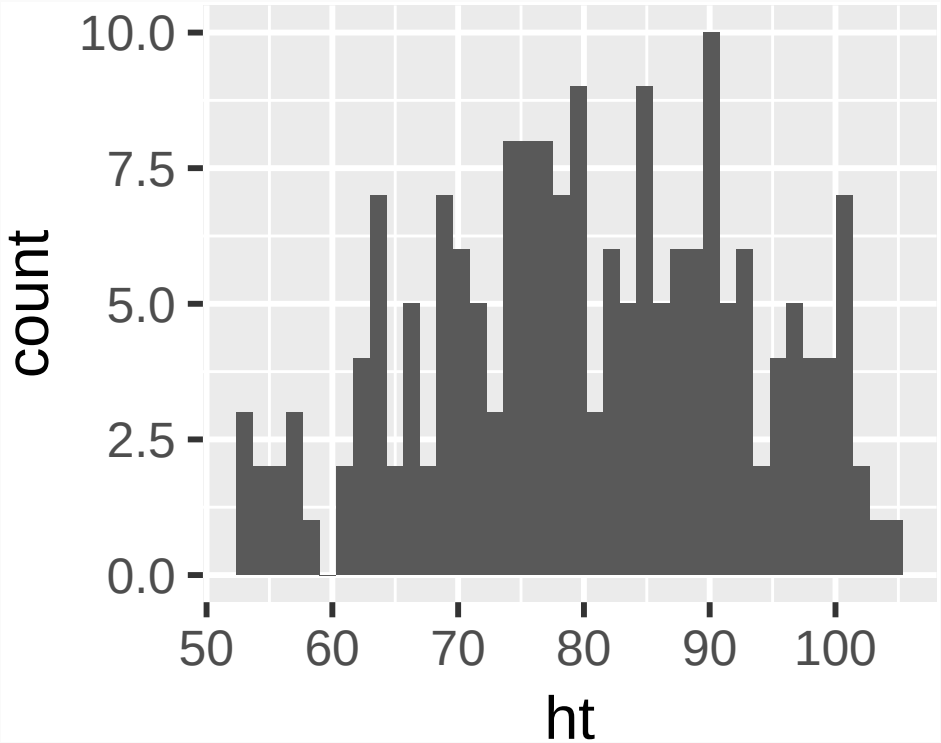
```
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black") +
  ggtitle("Height of children") +
  labs(x = "Height (cm)", y = "Weight (kg)") +
  xlim(c(0, 120))
```



Histogram example

`geom_histogram` also has its own special argument, `bins`. You can use this to change the number of bins that are used to make the histogram:

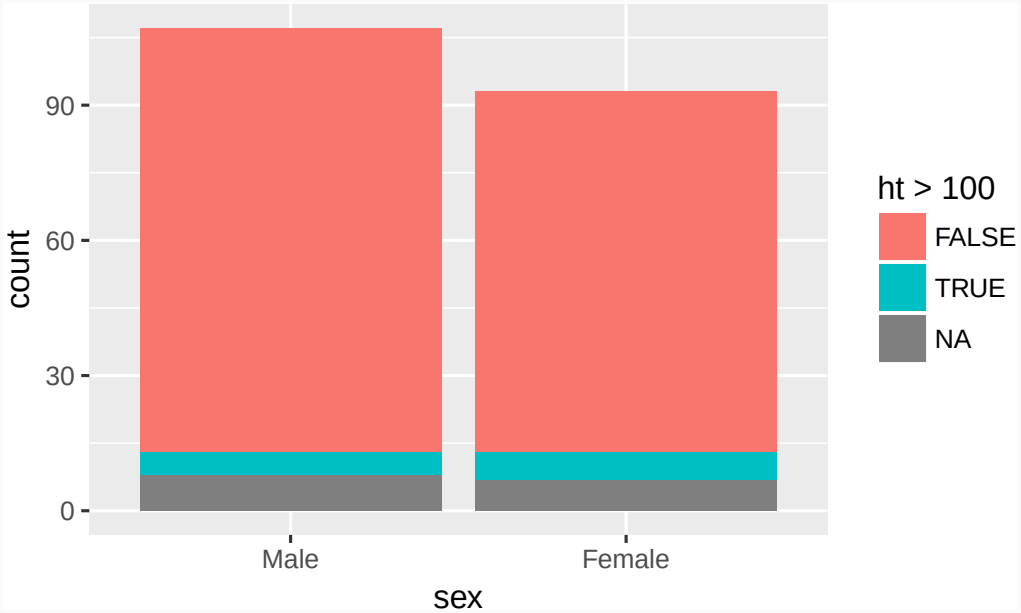
```
ggplot(nepali, aes(x = ht)) +
  geom_histogram(bins = 40)
```



Bar chart

You can use the `geom_bar` geom to create a barchart:

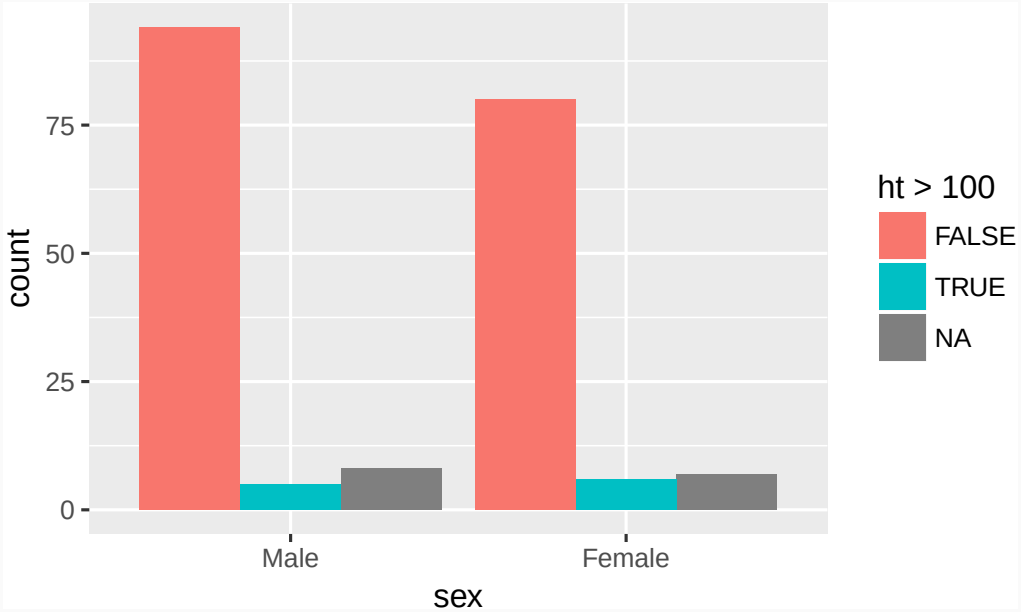
```
ggplot(nepali, aes(x = sex, fill = ht > 100)) +
  geom_bar()
```



Bar chart

With the `geom_bar` geom, you can use the `position` argument to change how the bars for different groups are shown (`"stack"`, `"dodge"`, `"fill"`):

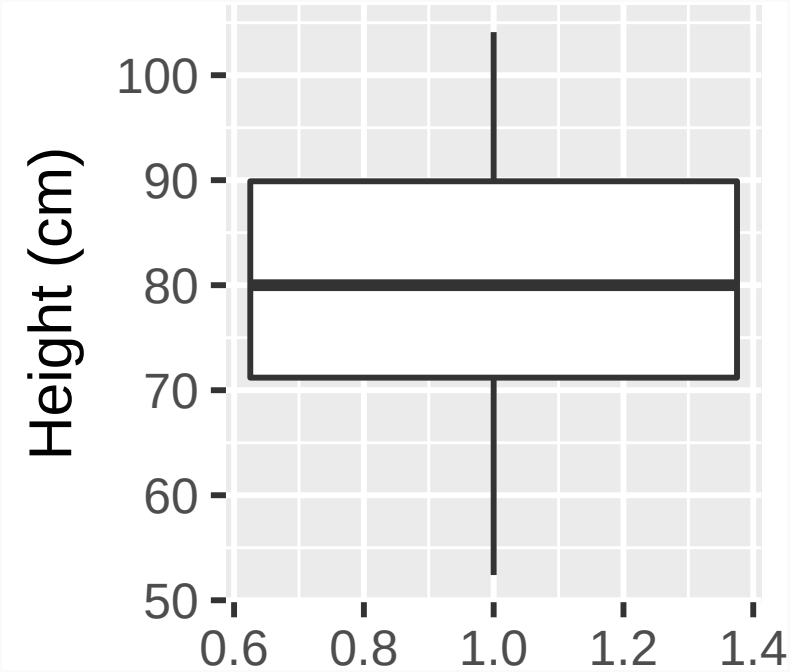
```
ggplot(nepali, aes(x = sex, fill = ht > 100)) +
  geom_bar(position = "dodge")
```



Boxplot example

To create a boxplot, you can use `geom_boxplot`:

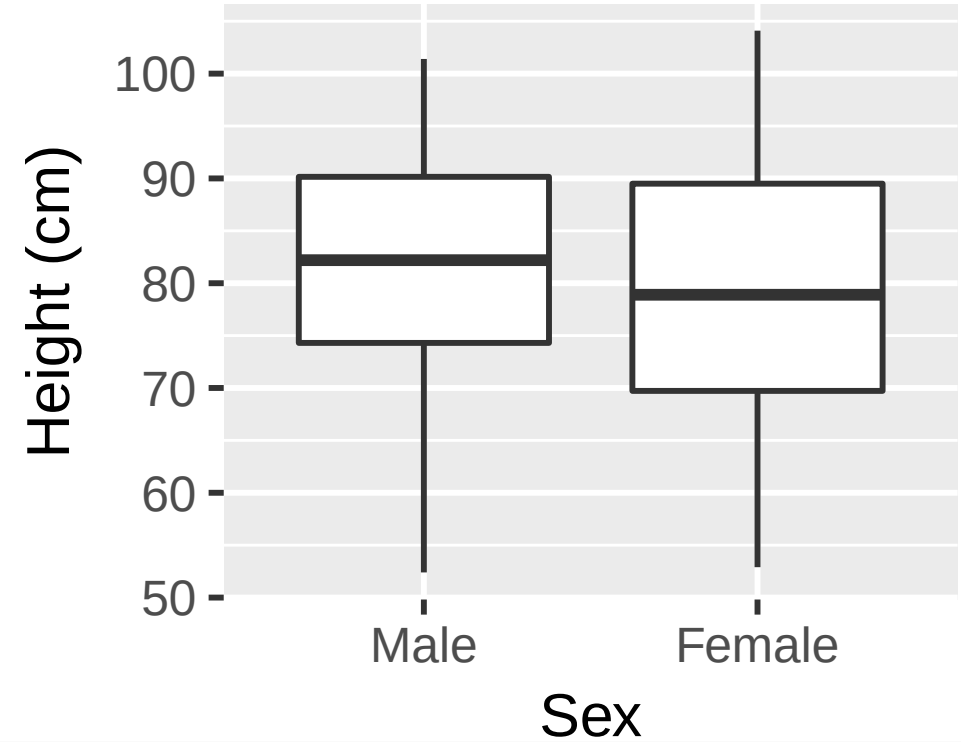
```
ggplot(nepali, aes(x = 1, y = ht)) +
  geom_boxplot() +
  labs(x = "", y = "Height (cm)")
```



Boxplot example

You can also do separate boxplots by a factor. In this case, you'll need to include two aesthetics (`x` and `y`) when you initialize the ggplot object.

```
ggplot(nepali, aes(x = sex, y = ht, group = sex)) +
  geom_boxplot() +
  labs(x = "Sex", y = "Height (cm)")
```



In-course exercise

We'll take a break now to finish the in-course exercise for this week (Section 3.6.6).