Exploring data #1

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:

```r
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:

```r
data()
```

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

```r
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()`:

```r
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.

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

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

```r
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.

```r
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
##     <fctr> <fctr> <dbl> <dbl> <int>
## 1 120011   Male  12.8  91.2    41
## 2 120012 Female  14.9 103.9    57
## 3 120021 Female   7.7  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:

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

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

```r
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
##    <fctr> <fctr> <dbl> <dbl> <int>   <lgl>
## 1 120011   Male  12.8  91.2    41    TRUE
## 2 120012 Female  14.9 103.9    57   FALSE
## 3 120021 Female   7.7  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
##    <fctr> <dbl>     <lgl>
## 1 120011  91.2     FALSE
## 2 120012 103.9      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
##   <fctr> <dbl>     <lgl>    <lgl>
## 1 120011  91.2     FALSE     TRUE
## 2 120012 103.9      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
##    <fctr> <dbl> <dbl> <fctr>
## 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   8.3   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
##     <fctr> <dbl> <dbl> <fctr>
## 1 120012 103.9  14.9 Female
## 2 120021  70.1   7.7 Female
## 3 120022  86.4  12.1 Female
## 4 120052  83.6  11.8 Female
## 5 120061  78.5   8.7 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:

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

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

```r
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:

```r
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
##   <fctr>      <int>     <dbl>   <fctr>
## 1   Male        107 10.497980   120011
## 2 Female         93  9.823256   120012
```

Grouping and summarizing

```r
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
##   <fctr>      <int>     <dbl>   <fctr>
## 1   Male        107 10.497980   120011
## 2 Female         93  9.823256   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
##   <fctr> <lgl>      <int>    <dbl>
## 1   Male FALSE         94 10.24787
## 2   Male  TRUE          5 15.20000
## 3 Female FALSE         80  9.42625
## 4 Female  TRUE          6 15.11667
```

In-course exercise

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

Plots

Plots to explore data

Plots can be invaluable in exploring your data.

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

Next week, 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:

```r
library(ggplot2)
```

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

1. Create an object of the `ggplot` class, typically specifying the **data** and some or all of the **aesthetics**;
2. Add on one or more **geoms** and other elements to create and customize the plot, using `+`.

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

Creating a ggplot object

The first step in plotting using `ggplot2` is to create a ggplot object.

Use the following conventions to do this:

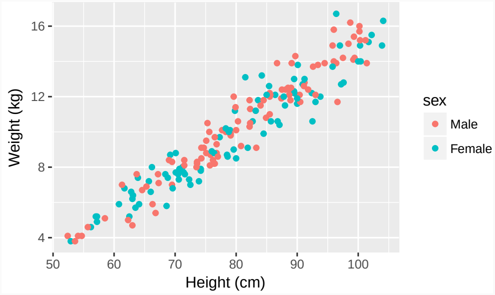
```r
## Generic code
ggplot(dataframe, aes(x = column_1, y = column_2))
## or
object <- ggplot(dataframe, aes(x = column_1, y = column_2))
```

Notice that you first specify the **dataframe** with the data you want to plot and then you might specify either **mappings** or constant values for some or all of the aesthetics (`aes`).

Plot aesthetics

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

For example, color might show gender, x-position might show height, and y-position might show weight.



In this graph, the mapped aesthetics are color, x, and y.

Note: Any of these aesthetics could also be given a constant value, instead of being mapped to an element of the data. For example, all the points

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)

Plot aesthetics

Which aesthetics you must specify depend on which geoms (more on those in a second) you're 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.

Adding geoms

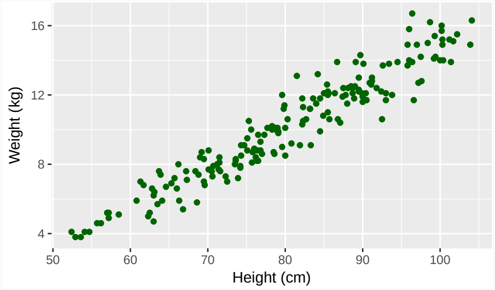
The second step in plotting using `ggplot2` is to add one or more `geoms` to create the plot. You can add these with + to the ggplot object you created.

Some of the most common geoms are:

Plot type	ggplot2 function
Histogram (1 numeric variable)	`geom_histogram`
Scatterplot (2 numeric variables)	`geom_point`
Boxplot (1 numeric variable, possibly 1 factor variable)	`geom_boxplot`
Line graph (2 numeric variables)	`geom_line`

Constant aesthetics

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

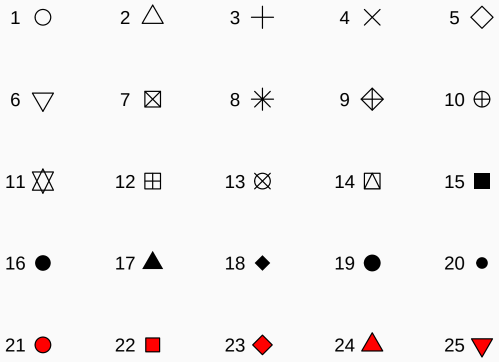


In this case, you'll define that aesthetic when you add the geom, outside of an `aes` statement.

Constant aesthetics

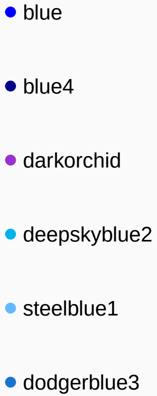
In R, you can specify point shape with a number.

Here are the shapes that correspond to the numbers 1 to 25:



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.

Useful plot additions

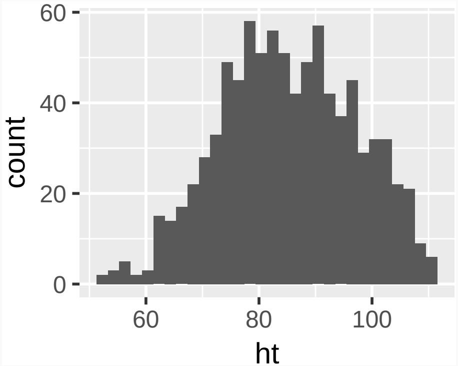
There are also a number of elements that you can add onto a `ggplot` object using `+`. A few very frequently used ones are:

Element	Description
`ggtitle`	Plot title
`xlab`, `ylab`	x- and y-axis labels
`xlim`, `ylim`	Limits of x- and y-axis

Histogram example

For `geom_histogram()`, the main aesthetic is `x`, the (numeric) vector for which you want to create a histogram:

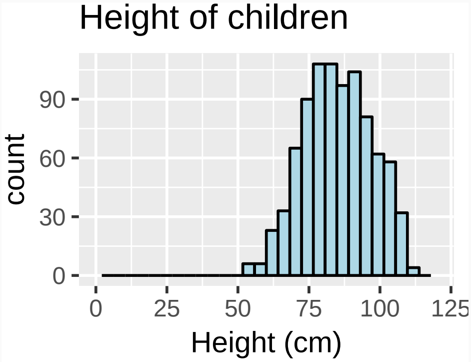
```
ggplot(nepali, aes(x = ht)) +
  geom_histogram()
```



Histogram example

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

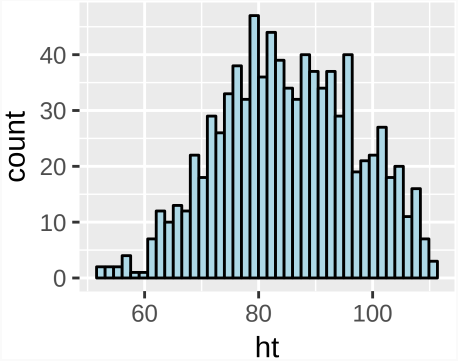
```r
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black") +
  ggtitle("Height of children") +
  xlab("Height (cm)") + xlim(c(0, 120))
```



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:

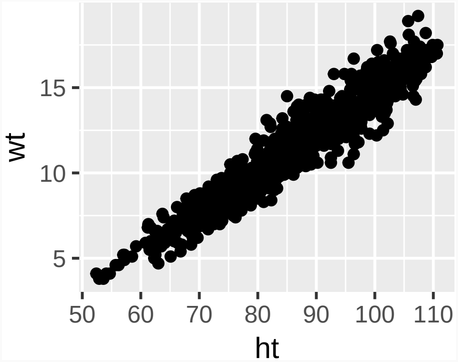
```r
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black",
                 bins = 40)
```



Scatterplot example

You can use the geom_point geom to create a scatterplot. For example, to create a scatterplot of height versus age for the Nepali data:

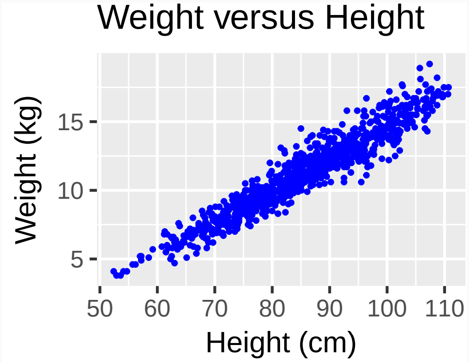
```
ggplot(nepali, aes(x = ht, y = wt)) +
  geom_point()
```



Scatterplot example

Again, you can use some of the options and additions to change the plot appearance:

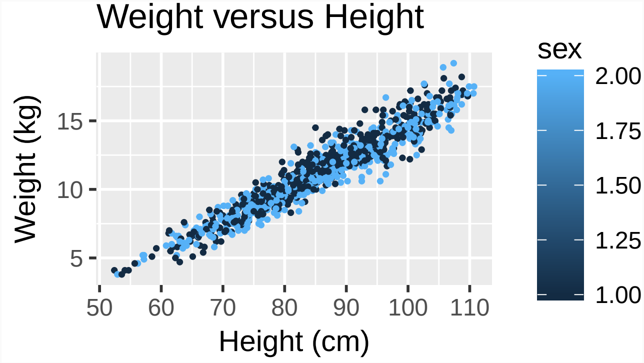
```
ggplot(nepali, aes(x = ht, y = wt)) +
  geom_point(color = "blue", size = 0.5) +
  ggtitle("Weight versus Height") +
  xlab("Height (cm)") + ylab("Weight (kg)")
```



Scatterplot example

You can also try mapping another variable, `sex`, to the `color` aesthetic:

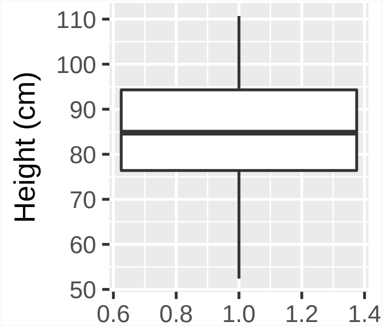
```
ggplot(nepali, aes(x = ht, y = wt, color = sex)) +
  geom_point(size = 0.5) +
  ggtitle("Weight versus Height") +
  xlab("Height (cm)") + ylab("Weight (kg)")
```



Boxplot example

To create a boxplot, use `geom_boxplot`:

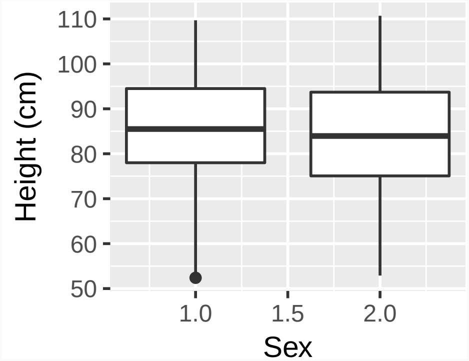
```
ggplot(nepali, aes(x = 1, y = ht)) +
  geom_boxplot() +
  xlab("")+ ylab("Height (cm)")
```



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() +
  xlab("Sex")+ ylab("Height (cm)")
```


Height (cm)
Sex