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:

```
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()
```

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:

```
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:

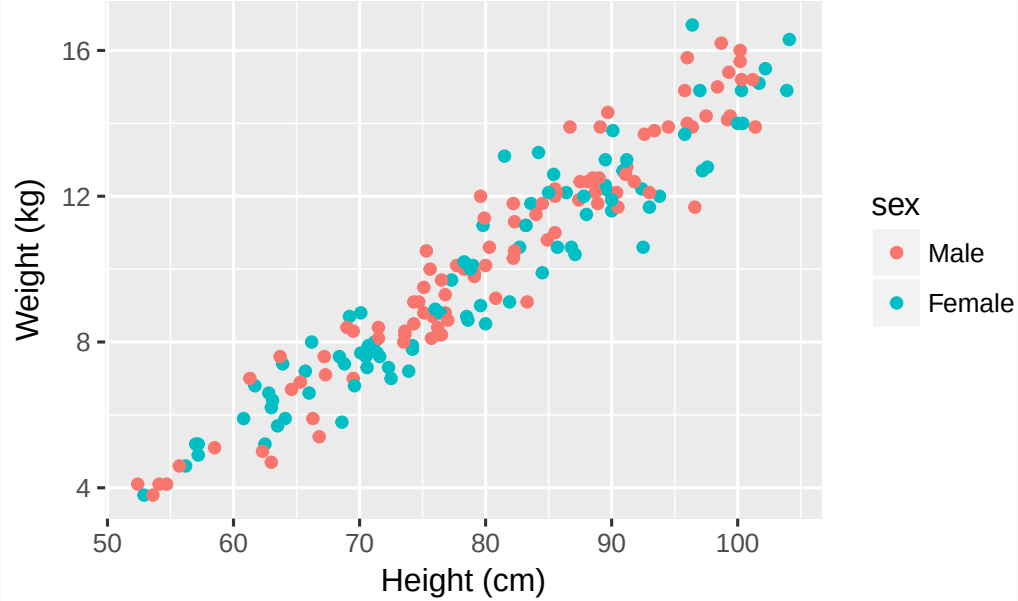
```
## Generic code
ggplot(dataframe, aes(x = column_1, y = column_2))
## or
object <- ggplot(dataframe, aes(x = column_1, y = column_2))
```

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

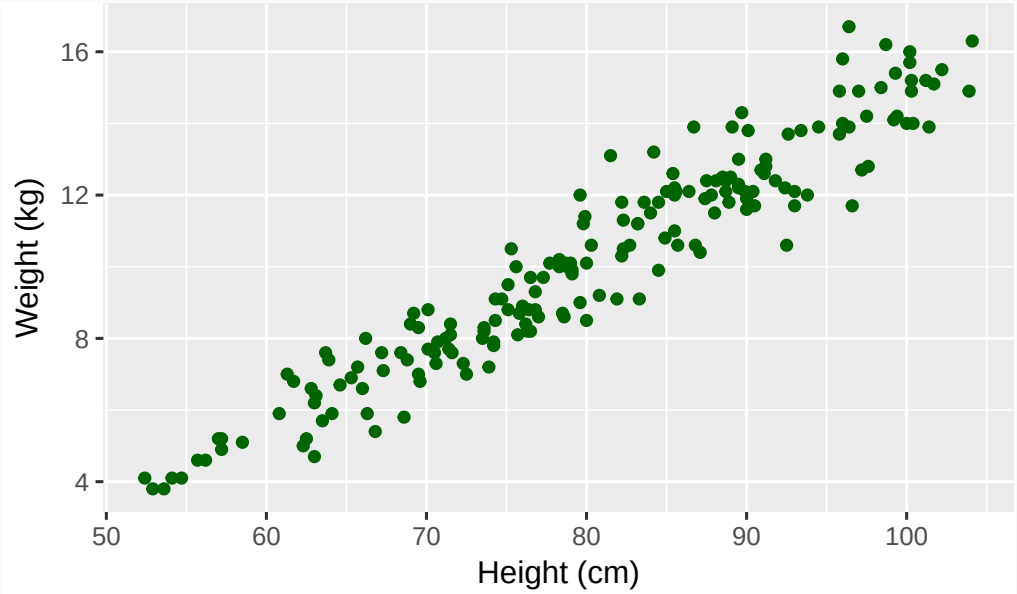
The second step in plotting using `ggplot2` is to add one or more `geoms` to create the plot. You can add these with + to the ggplot object you created.

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:

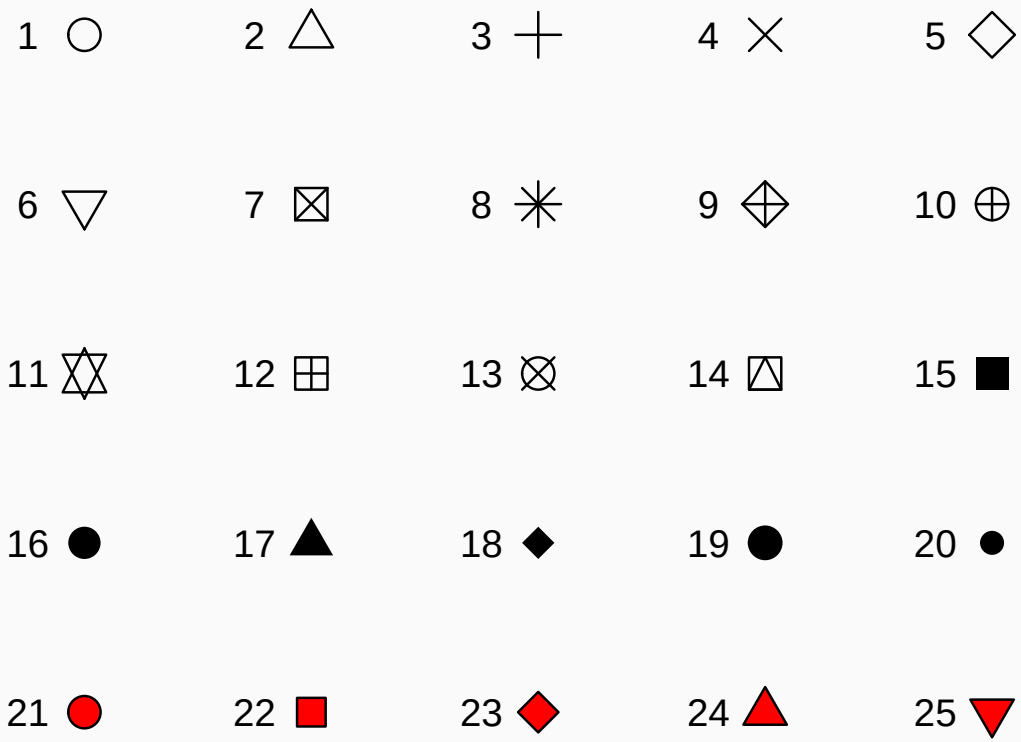


In this case, you'll define that aesthetic when you add the geom, outside of an `aes` statement.

Constant aesthetics

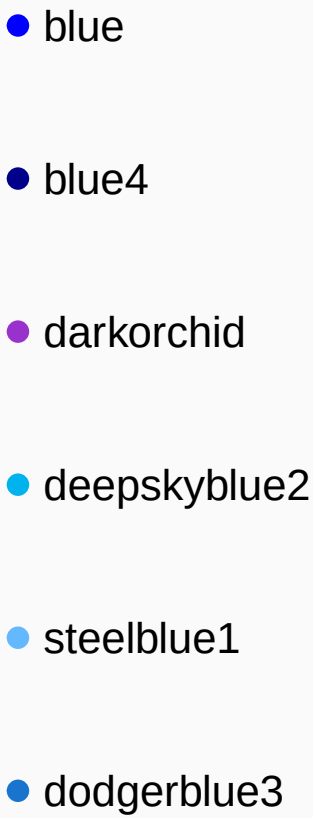
In R, you can specify point shape with a number.

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

Example plots

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`:

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

Example plots

Each observation is a single measurement for a child; there can be multiple observations per child.

I'll subset out child id, sex, weight, height, and age, and I'll limit to each child's first measurement.

```
nepali <- nepali %>%
  # Subset 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 data now looks like:

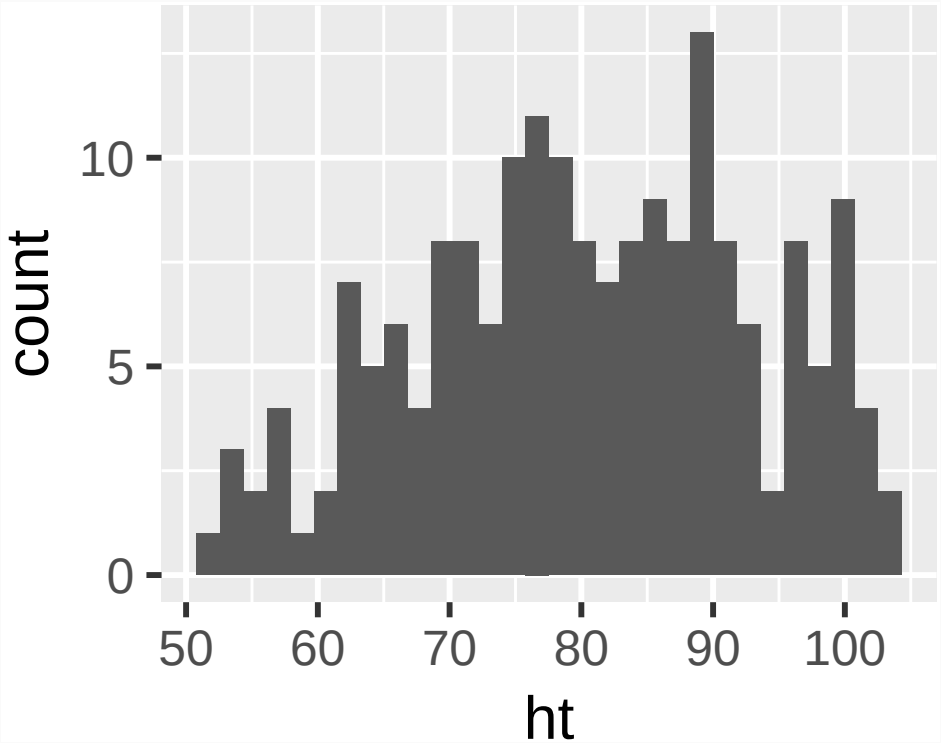
```
head(nepali)
```

```
##       id    sex   wt    ht age
## 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
## 5 120023   Male 14.2  99.4  49
## 6 120031   Male 13.9  96.4  46
```

Histogram example

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

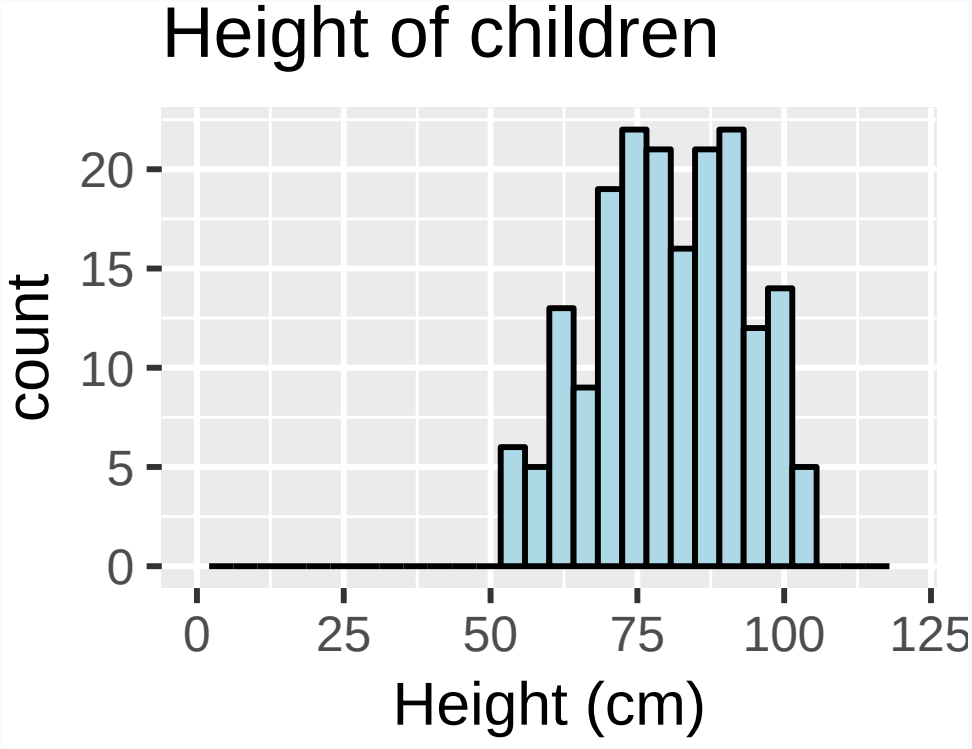
```
ggplot(nepali, aes(x = ht)) +
  geom_histogram()
```



Histogram example

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

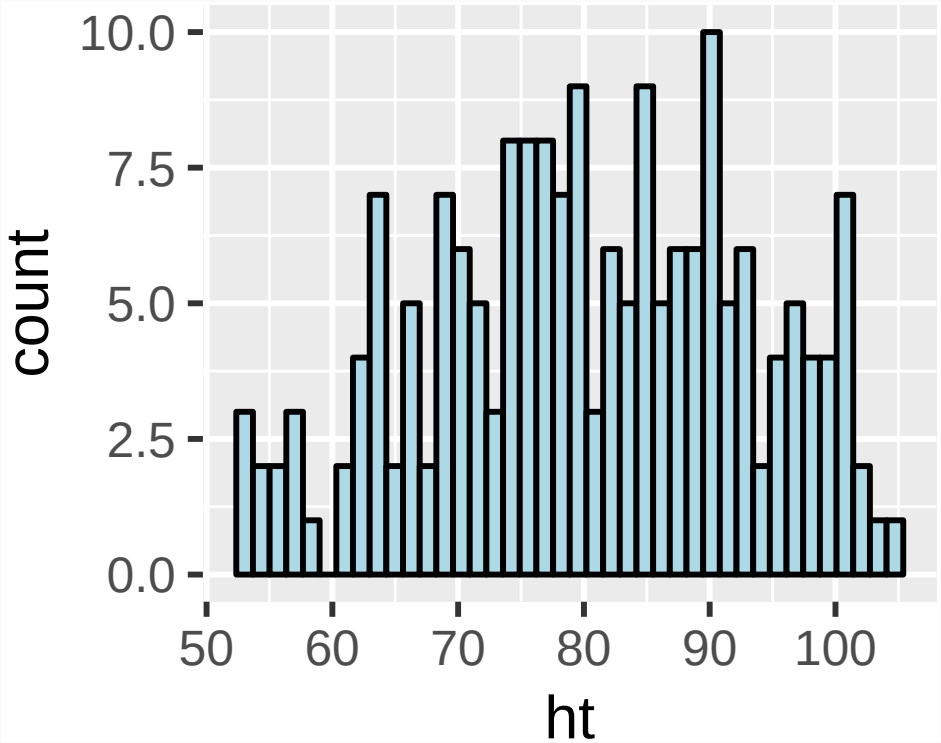
```
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black") +
  ggtitle("Height of children") +
  xlab("Height (cm)") + xlim(c(0, 120))
```



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:

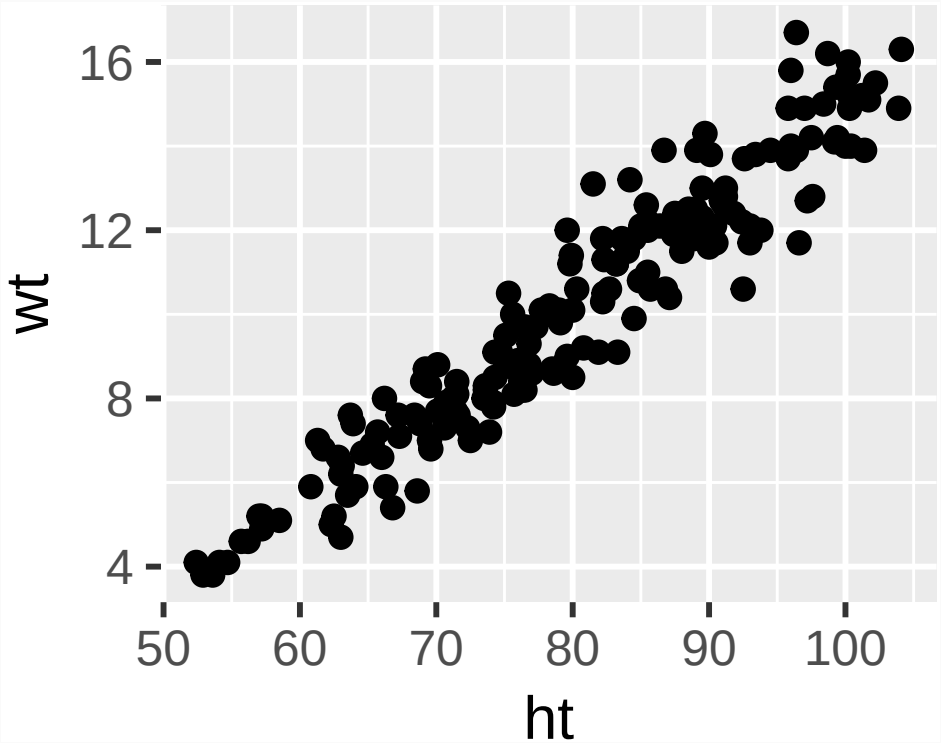
```
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black",
                 bins = 40)
```



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:

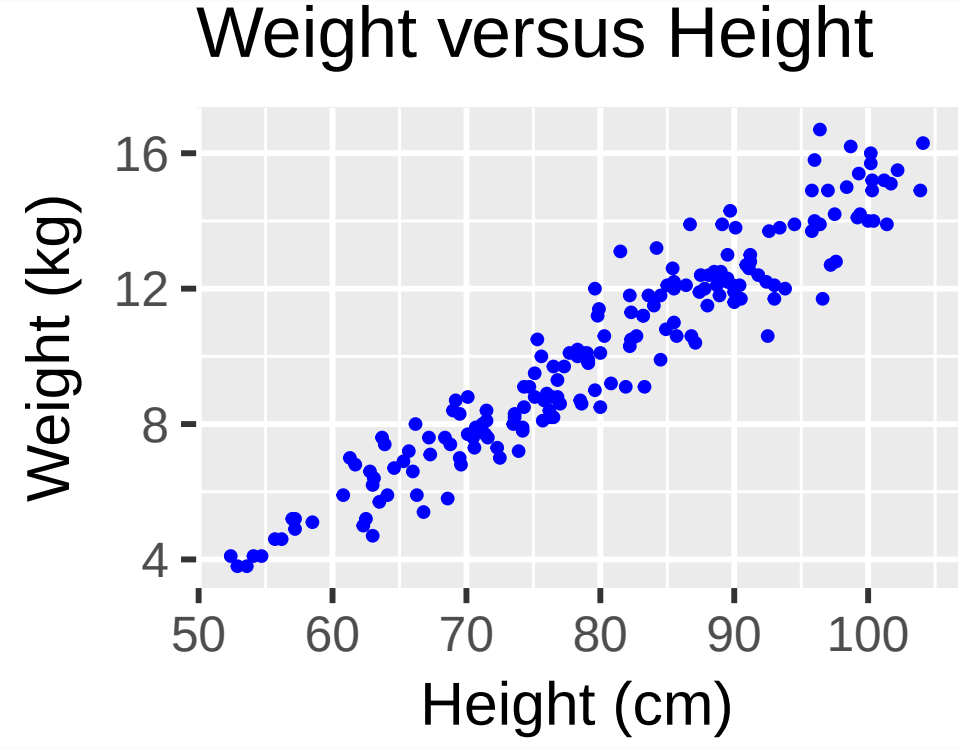
```
ggplot(nepali, aes(x = ht, y = wt)) +
  geom_point()
```



Scatterplot example

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

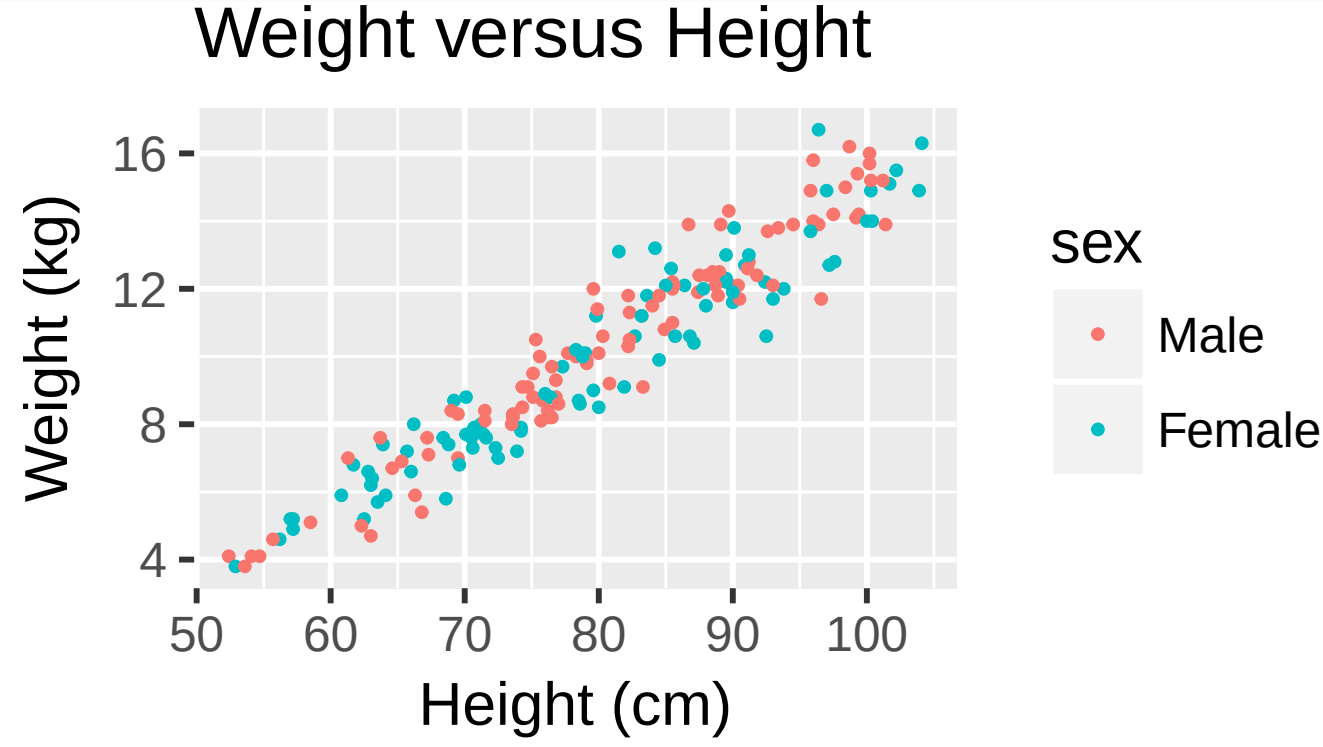
```
ggplot(nepali, aes(x = ht, y = wt)) +
  geom_point(color = "blue", size = 0.5) +
  ggtitle("Weight versus Height") +
  xlab("Height (cm)") + ylab("Weight (kg)")
```



Scatterplot example

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

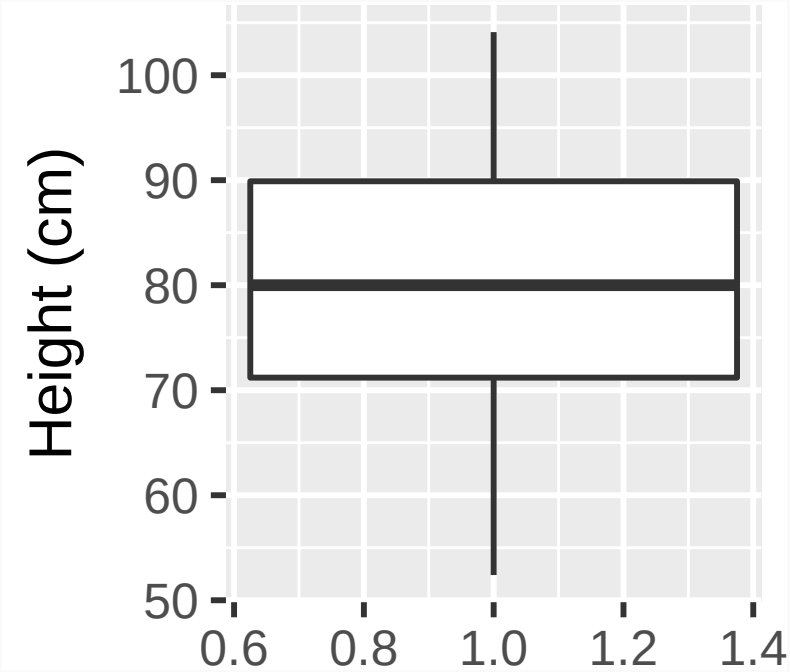
```
ggplot(nepali, aes(x = ht, y = wt, color = sex)) +
  geom_point(size = 0.5) +
  ggtitle("Weight versus Height") +
  xlab("Height (cm)") + ylab("Weight (kg)")
```



Boxplot example

To create a boxplot, use `geom_boxplot`:

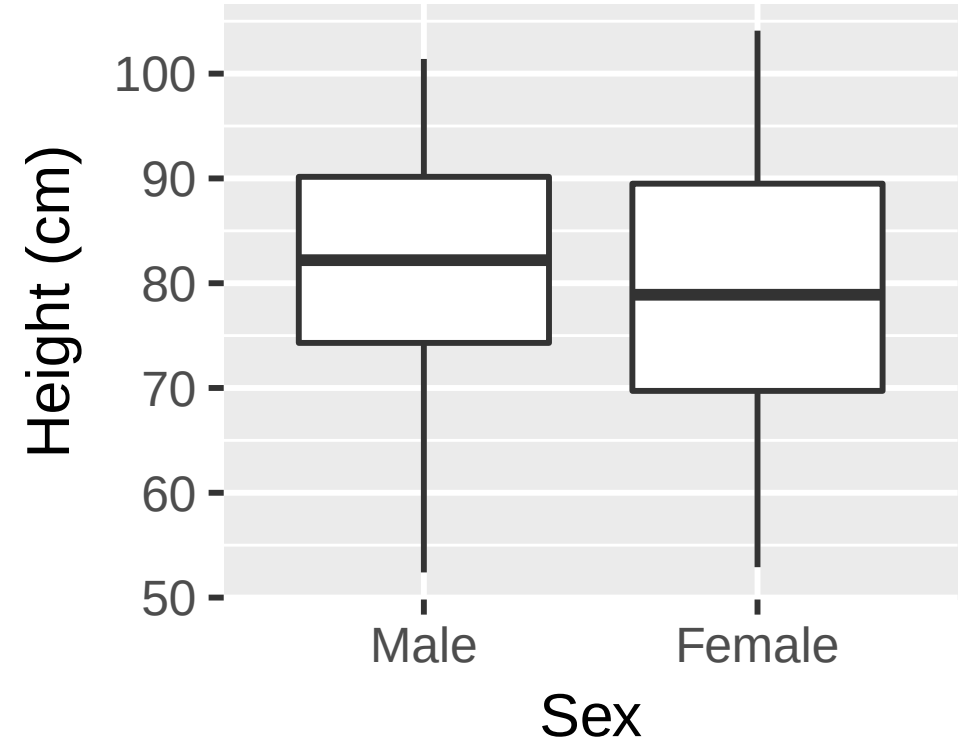
```
ggplot(nepali, aes(x = 1, y = ht)) +
  geom_boxplot() +
  xlab("")+ ylab("Height (cm)")
```



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)")
```



ggpairs() example

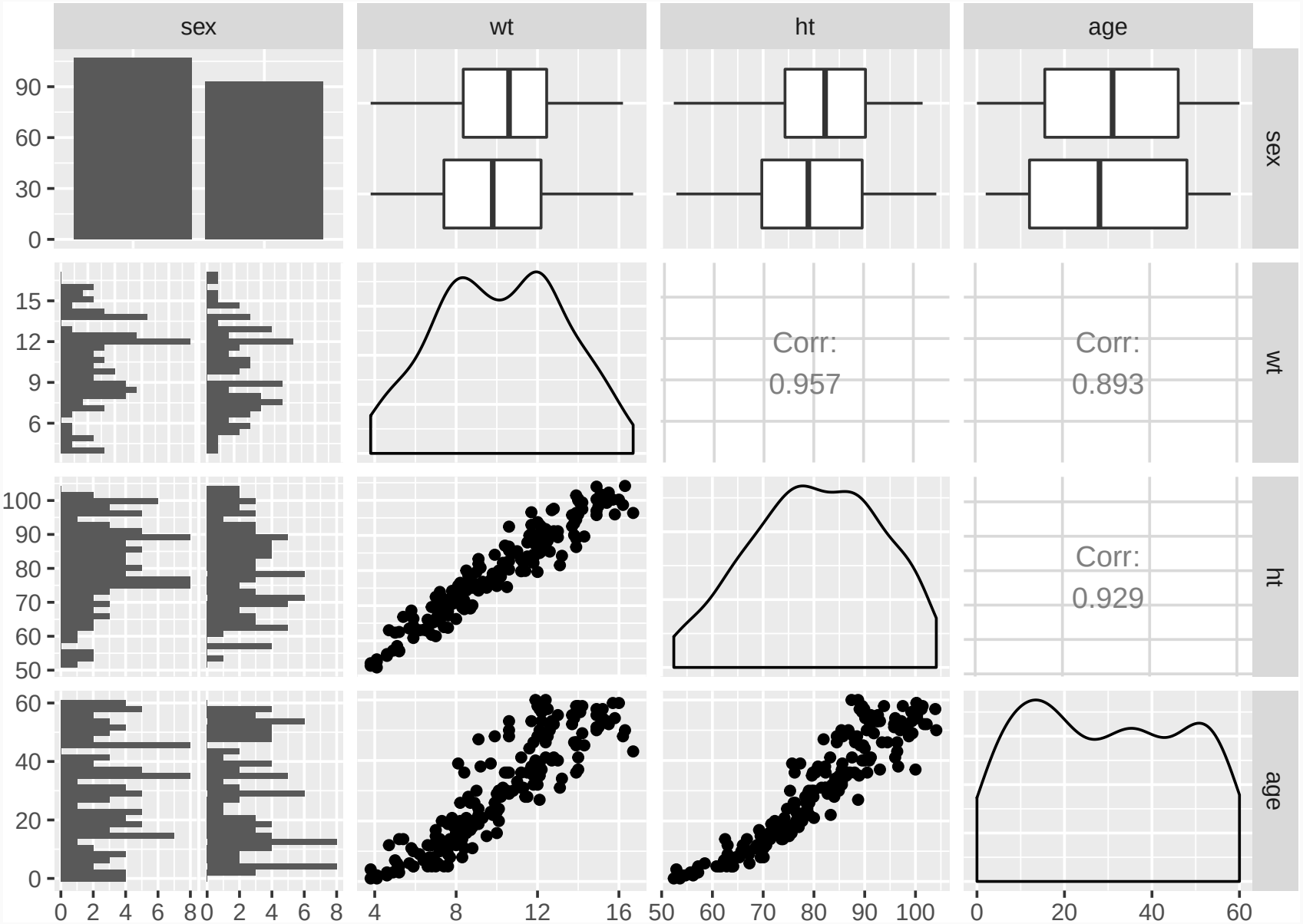
There are lots of R extensions for creating other interesting plots.

For example, you can use the ggpairs function from the GGally package to plot all pairs of scatterplots for several variables.

Notice how this output shows continuous and binary variables differently.

The next slide shows the output for:

```
library(GGally)
ggpairs(nepali[, c("sex", "wt", "ht", "age")])
```


sex
wt
ht
age
sex
wt
ht
age
Corr:
0.957
Corr:
0.893
Corr:
0.929

Simple statistics

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
```

We'll see more of this when we do regression models.

Simple statistic examples

You can also perform many of these tasks using `dplyr`. For example, to get the mean weight, you can use the `summarize` function:

```
nepali %>%
  summarize(mean_wt = mean(wt, na.rm = TRUE))
```

```
##    mean_wt
## 1 10.18432
```

Simple statistic examples

The basic format for using `summarize` is:

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

There are some special functions that you can use with `summarize`. For example, `n` and `first` (see the Data Wrangling cheatsheet for more):

```
nepali %>%
  summarize(n_children =n(),
            first_id = first(id))
```

```
##   n_children first_id
## 1        200   120011
```

Simple statistic examples

If you want to get summaries by group using `dplyr` (e.g., mean weight by sex), use `group_by` before running `summarize`:

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

```
## # A tibble: 2 x 4
##      sex   mean_wt n_children first_id
##   <fctr>     <dbl>      <int>   <fctr>
## 1   Male 10.497980        107   120011
## 2 Female  9.823256         93   120012
```

Logical statements

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 other things. For example, you can use them directly in the square bracket indexing (`[..., ...]`).

Logical vectors

A logical statement run on a vector will create a logical vector. This logical vector will be the same length as the original vector:

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

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

```
length(is_male)
```

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

Logical vectors

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

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

```
head(is_male)
```

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

```
head(!is_male)
```

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

Logical vectors

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

```
head(nepali[is_male, ])
```

```
##        id  sex   wt   ht age
## 1  120011 Male 12.8 91.2  41
## 5  120023 Male 14.2 99.4  49
## 6  120031 Male 13.9 96.4  46
## 7  120051 Male  8.3 69.5   8
## 9  120053 Male 15.8 96.0  54
## 11 120062 Male 12.1 89.9  57
```

Logical vectors

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

```
head(nepali[!is_male, ])
```

```
##         id    sex   wt    ht age
## 2   120012 Female 14.9 103.9  57
## 3   120021 Female  7.7  70.1   8
## 4   120022 Female 12.1  86.4  35
## 8   120052 Female 11.8  83.6  32
## 10 120061 Female  8.7  78.5  26
## 15 120082 Female 11.2  79.8  36
```

Logical vectors

You can also use `sum()` and `table()` to find out how many males and females are in the dataset:

```
sum(is_male); sum(!is_male)
```

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

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

```
table(is_male)
```

```
## is_male
## FALSE  TRUE
##    93   107
```

dplyr equivalent

As a note, you could also achieve that with dplyr functions. One way to do this is to use mutate with a logical statement to create an is_male column, then group by that new column and summarize:

```
nepali %>%
  mutate(is_male = sex == "Male") %>%
  group_by(is_male) %>%
  summarize(n_children = n())
```

```
## # A tibble: 2 x 2
##   is_male n_children
##     <lgl>      <int>
## 1   FALSE         93
## 2    TRUE        107
```

Regression models

Formula structure

Regression models can be used to estimate how the expected value of a *dependent variable* changes as *independent variables* change.

In R, regression formulas take this structure:

```
## Generic code
[response variable] ~ [indep. var. 1] +  [indep. var. 2] + ...
```

Notice that ~ used to separate the independent and dependent variables and the + used to join independent variables. This format mimics the statistical notation:

$$Y_i \sim X_1 + X_2 + X_3$$

You will use this type of structure in R fo a lot of different function calls, including those for linear models (`lm`) and generalized linear models (`glm`).

Linear models

To fit a linear model, you can use the function `lm()`. Use the `data` option to specify the dataframe from which to get the vectors. You can save the model as an object.

```
mod_a <- lm(wt ~ ht, data = nepali)
```

This call fits the model:

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \epsilon_i$$

where:

- Y_i : weight of child i
- $X_{1,i}$: height of child i

Using model objects

Some functions you can use on model objects:

Function	Description
`summary`	Get a variety of information on the model, including coefficients and p-values for the coefficients
`coef`	Pull out just the coefficients for a model
`fitted`	Get the fitted values from the model (for the data used to fit the model)
`plot`	Create plots to help assess model assumptions
`residuals`	Get the model residuals

Examples of using a model object

For example, you can get the coefficients from the model we just fit:

```
coef(mod_a)
```

```
## (Intercept)          ht
##   -8.694768    0.235050
```

The estimated coefficient for the intercept is always given under the name "(Intercept)".

Estimated coefficients for independent variables are given based on their column names in the original data ("ht" here, for β_1, or the estimated increase in expected weight for a one unit increase in height).

Examples of using a model object

You can also pull out the residuals from the model fit:

```
head(residuals(mod_a))
```

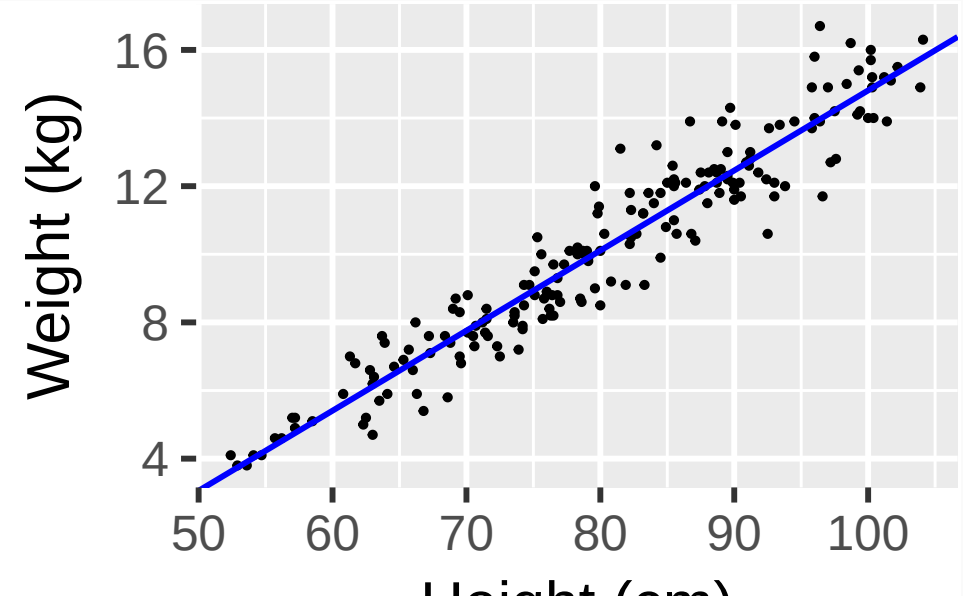
```
##           1           2           3           4
##  0.05820415 -0.82693141 -0.08223993  0.48644436
##           5           6
## -0.46920621 -0.06405608
```

This is a vector the same length as the number of observations (rows) in the dataframe you used to fit the model. The residuals are in the same order as the observations in the original dataframe.

Examples of using a model object

You can use the coef results to plot a regression line based on the model fit on top of points showing the original data:

```
mod_coef <- coef(mod_a)
ggplot(nepali, aes(x = ht, y = wt)) +
  geom_point(size = 0.2) +
  xlab("Height (cm)") + ylab("Weight (kg)") +
  geom_abline(aes(intercept = mod_coef[1],
                  slope = mod_coef[2]), col = "blue")
```



Examples of using a model object

The `summary()` function gives you a lot of information about the model:

```
summary(mod_a)
```

(see next slide)

```
##
## Call:
## lm(formula = wt ~ ht, data = nepali)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.44736 -0.55708  0.01925  0.49941  2.73594
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -8.694768   0.427398  -20.34   <2e-16
## ht           0.235050   0.005257   44.71   <2e-16
##
## (Intercept) ***
## ht          ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

summary for lm objects

The object created when you use the summary() function on an lm object has several different parts you can pull out using the $ operator:

```
names(summary(mod_a))
```

```
##  [1] "call"          "terms"
##  [3] "residuals"     "coefficients"
##  [5] "aliased"       "sigma"
##  [7] "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"
## [11] "cov.unscaled"  "na.action"
```

```
summary(mod_a)$coefficients
```

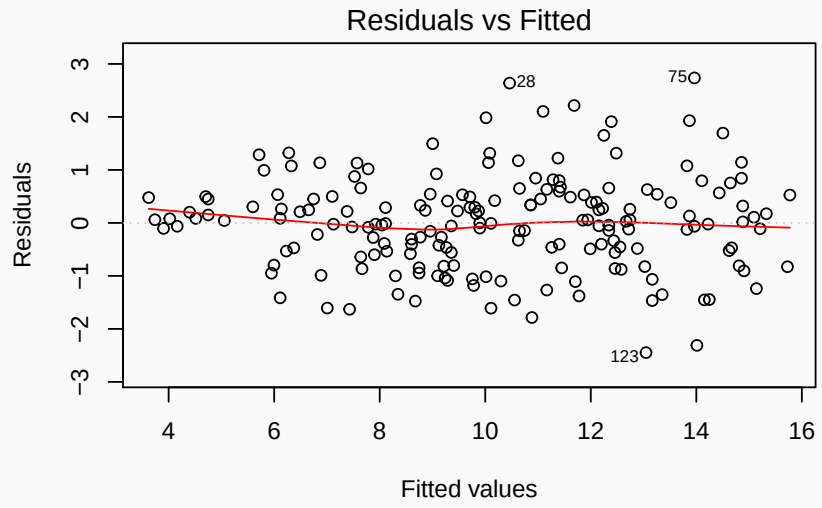
```
##               Estimate  Std. Error   t value
## (Intercept) -8.694768 0.427397843 -20.34350
## ht           0.235050 0.005256822  44.71334
##                  Pr(>|t|)
```

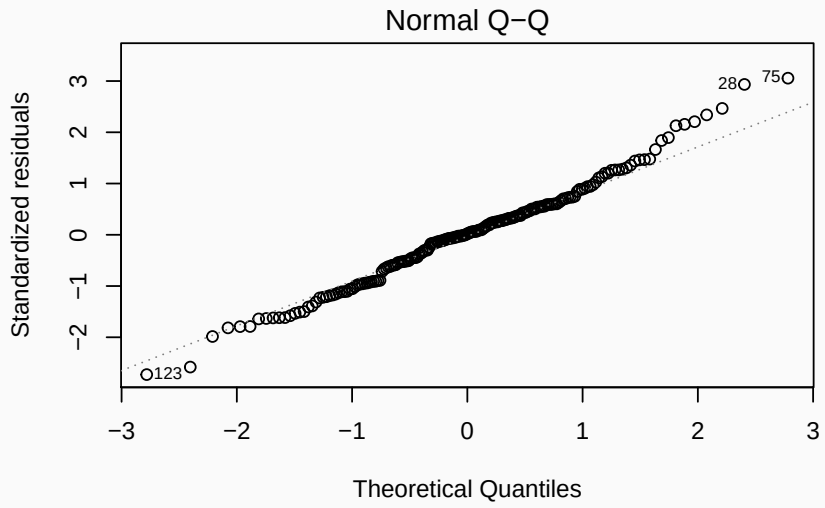
Using `plot()` with `lm` objects

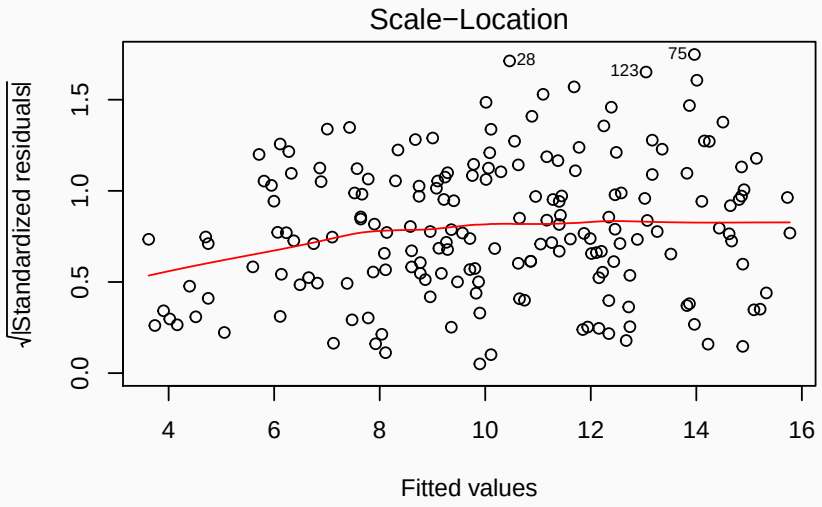
You can use `plot` with an `lm` object to get a number of useful diagnostic plots to check regression assumptions:

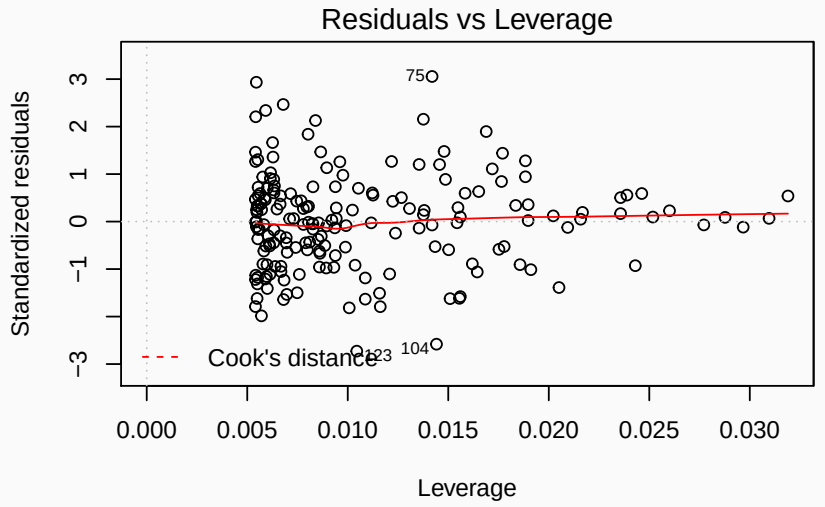
```
plot(mod_a)
```

(See next slide)


Residuals vs Fitted
Residuals
Fitted values
28
75
123


Normal Q−Q
Standardized residuals
Theoretical Quantiles
28
75
123


Scale−Location
√|Standardized residuals|
Fitted values
28
123
75


Residuals vs Leverage
Standardized residuals
Leverage
Cook's distance
75
123
104

Fitting a model with a factor

You can also use binary variables or factors as independent variables in regression models:

```
mod_b <- lm(wt ~ sex, data = nepali)
summary(mod_b)$coefficients
```

```
##                Estimate Std. Error   t value
## (Intercept) 10.497980  0.3110957 33.745177
## sexFemale   -0.674724  0.4562792 -1.478752
##                    Pr(>|t|)
## (Intercept) 1.704550e-80
## sexFemale   1.409257e-01
```

This call fits the model:

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \epsilon_i$$

where $X_{1,i}$: sex of child i, where 0 = male; 1 = female

Linear models versus GLMs

You can fit a variety of models, including linear models, logistic models, and Poisson models, using generalized linear models (GLMs).

For linear models, the only difference between `lm` and `glm` is how they're fitting the model (least squares versus maximum likelihood). You should get the same results regardless of which you pick.

Linear models versus GLMs

For example:

```
mod_c <- glm(wt ~ ht, data = nepali)
summary(mod_c)$coef
```

```
##                Estimate  Std. Error   t value
## (Intercept) -8.694768 0.427397843 -20.34350
## ht           0.235050 0.005256822  44.71334
##                  Pr(>|t|)
## (Intercept)  7.424640e-49
## ht          1.962647e-100
```

```
summary(mod_a)$coef
```

```
##                Estimate  Std. Error   t value
## (Intercept) -8.694768 0.427397843 -20.34350
## ht           0.235050 0.005256822  44.71334
##                  Pr(>|t|)
```

GLMs

You can fit other model types with `glm()` using the `family` option:

Model type	`family` option
Linear	`family = gaussian(link = 'identity')`
Logistic	`family = binomial(link = 'logit')`
Poisson	`family = poisson(link = 'log')`

Logistic example

For example, say we wanted to fit a logistic regression for the `nepali` data of whether the probability that a child weighs more than 13 kg is associated with the child's height.

First, create a binary variable for `wt_over_13`:

```
nepali <- nepali %>%
  mutate(wt_over_13 = wt > 13)
head(nepali)
```

```
##       id    sex   wt    ht age wt_over_13
## 1 120011   Male 12.8  91.2  41      FALSE
## 2 120012 Female 14.9 103.9  57       TRUE
## 3 120021 Female  7.7  70.1   8      FALSE
## 4 120022 Female 12.1  86.4  35      FALSE
## 5 120023   Male 14.2  99.4  49       TRUE
## 6 120031   Male 13.9  96.4  46       TRUE
```

Logistic example

Now you can fit a logistic regression:

```
mod_d <- glm(wt_over_13 ~ ht, data = nepali,
             family = binomial(link = "logit"))
summary(mod_d)$coef
```

```
##                Estimate Std. Error   z value
## (Intercept) -32.7016520 5.85196755 -5.588147
## ht            0.3495227 0.06331892  5.520036
##                  Pr(>|z|)
## (Intercept) 2.295060e-08
## ht          3.389307e-08
```

Here, the model coefficient gives the **log odds** of having a weight higher than 13 kg associated with a unit increase in height.

Formula structure

There are some conventions that can be used in R formulas. Common ones include:

Convention	Meaning
`I()`	calculate the value inside before fitting (e.g., `I(x1 + x2)`)
`:`	fit the interaction between two variables (e.g., `x1:x2`)
`*`	fit the main effects and interaction for both variables (e.g., `x1*x2` equals `x1 + x2 + x1:x2`)
`.`	fit all variables other than the response (e.g., `y ~ .`)
`-`	do not include a variable (e.g., `y ~ . - x1`)
`1`	intercept (e.g., `y ~ 1`)

To find out more

A great (and free for CSU students) resource to find out more about using R for basic statistics:

- Introductory Statistics with R

If you want all the details about fitting linear models and GLMs in R, Faraway's books are fantastic:

- Linear Models with R (also freely available through our library)
- Extending the Linear Model with R