Exploring data 2

Regression models

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 data now looks like:

```r
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
```

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.

```r
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:

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

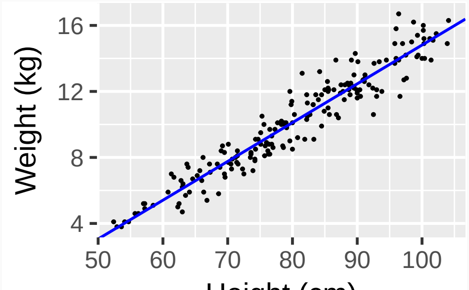
```
##           1           2           3           4           5
##  0.05820415 -0.82693141 -0.08223993  0.48644436 -0.46920621 -
```

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:

```r
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 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
##
## Residual standard error: 0.9017 on 183 degrees of freedom
##   (15 observations deleted due to missingness)
## Multiple R-squared:  0.9161, Adjusted R-squared:  0.9157
## F-statistic:  1999 on 1 and 183 DF,  p-value: < 2.2e-16
```

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"         "residuals"     "coeffic
##  [5] "aliased"       "sigma"         "df"            "r.squar
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"  "na.acti
```

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

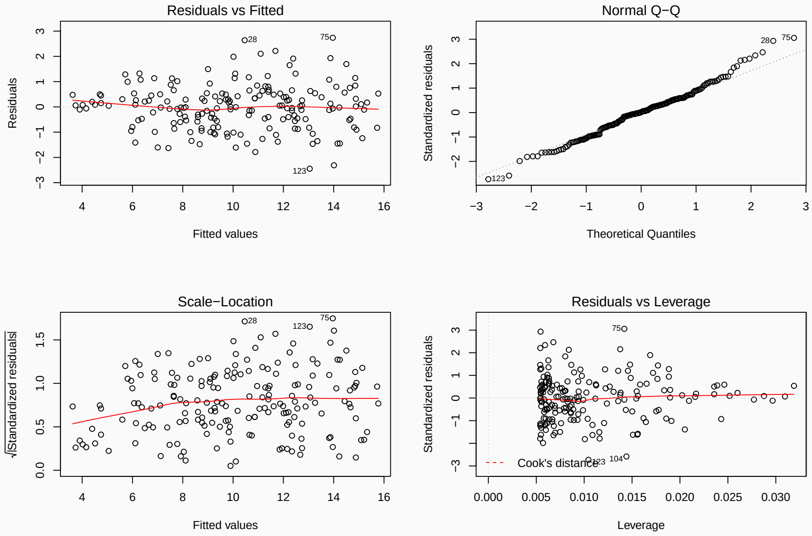
```
##              Estimate  Std. Error   t value      Pr(>|t|)
## (Intercept) -8.694768 0.427397843 -20.34350  7.424640e-49
## ht           0.235050 0.005256822  44.71334 1.962647e-100
```

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
Standard residuals
Theoretical Quantiles
Scale−Location
√|Standardized residuals|
Residuals vs Leverage
Standardized residuals
Leverage
Cook's distance
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     Pr(>|t|)
## (Intercept) 10.497980  0.3110957 33.745177 1.704550e-80
## sexFemale   -0.674724  0.4562792 -1.478752 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      Pr(>|t|)
## (Intercept) -8.694768 0.427397843 -20.34350  7.424640e-49
## ht           0.235050 0.005256822  44.71334 1.962647e-100
```

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

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

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     Pr(>|z|)
## (Intercept) -32.7016520 5.85196755 -5.588147 2.295060e-08
## ht            0.3495227 0.06331892  5.520036 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

Parentheses

Parentheses

If you put parentheses around an entire code statement, it will both run the code and print out the answer.

```
study_months <- c("Jan", "Feb", "Mar")
study_months
```

```
## [1] "Jan" "Feb" "Mar"
```

```
(study_months <- c("Jan", "Feb", "Mar"))
```

```
## [1] "Jan" "Feb" "Mar"
```

Loops

Loops

Loops allow you to “walk through” and repeat the same code for different values of an index.

For each run of the loop, R is told that, for **some index** in **some vector**, do **some code**.

For `i` in `1:3`, `print(i)`:

```
for(i in c(1, 2, 3)){
        print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

Loops

Note that this code is equivalent to:

```r
i <- 1
print(i)
```

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

```r
i <- 2
print(i)
```

```
## [1] 2
```

```r
i <- 3
print(i)
```

```
## [1] 3
```

Loops

Often, the index will be set to a number for each cycle of the loop, and then the index will be used within the code to index vectors or data frames:

```
study_months <- c("Jan", "Feb", "Mar")
for(i in c(1, 3)){
        print(study_months[i])
}
```

```
## [1] "Jan"
## [1] "Mar"
```

Loops

Often, you want to set the index to sequential numbers (e.g., 1, 2, 3, 4). In this case, you can save time by using the `:` notation to create a vector of a sequence of numbers:

```
for(i in 1:3){
        print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

Loops

With this notation, sometimes it may be helpful to use the `length` function to set the largest index value for the loop as the length of a vector (or `nrow` for indexing a data frame). For example:

```r
study_months <- c("Jan", "Feb", "Mar")
for(i in 1:length(study_months)){
        print(study_months[i])
}
```

```
## [1] "Jan"
## [1] "Feb"
## [1] "Mar"
```

Loops

Sometimes, you want to set the index for each cycle of the loop to something that is not a number. You can set the index to any class of vector.

Remember that a loop works by saying for **some index** in **some vector**, do **some code**.

For example, you may want to run: for `study_month` in `study_months`, `print(study_month)`:

```r
study_months <- c("Jan", "Feb", "Mar")
for(study_month in study_months){
        print(study_month)
}
```

```
## [1] "Jan"
## [1] "Feb"
## [1] "Mar"
```

Loops

Note that this is equivalent to:

```r
study_month <- "Jan"
print(study_month)
```

```
## [1] "Jan"
```

```r
study_month <- "Feb"
print(study_month)
```

```
## [1] "Feb"
```

```r
study_month <- "Mar"
print(study_month)
```

```
## [1] "Mar"
```

Loops

What would this loop do?

```r
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
for(i in 1:length(vars)){
        var_mean <- mean(worldcup[ , vars[i]])
        print(var_mean)
}
```

Loops

```
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
for(i in 1:length(vars)){
        var_mean <- mean(worldcup[ , vars[i]])
        print(var_mean)
}
```

```
## [1] 208.8639
## [1] 2.304202
## [1] 84.52101
## [1] 4.191597
## [1] 0.6672269
```

Loops

What would this loop do?

```r
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
for(i in 1:length(vars)){
        var_mean <- mean(worldcup[ , vars[i]])
        var_mean <- round(var_mean, 1)
        out <- paste0("mean of ", vars[i], ": ", var_mean)
        print(out)
}
```

Loops

To figure out, you can set `i <- 1` and then walk through the loop:

```r
i <- 1
(var_mean <- mean(worldcup[ , vars[i]]))
```

```
## [1] 208.8639
```

```r
(var_mean <- round(var_mean, 1))
```

```
## [1] 208.9
```

```r
(out <- paste0("mean of ", vars[i], ": ", var_mean))
```

```
## [1] "mean of Time: 208.9"
```

Loops

```r
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
for(i in 1:length(vars)){
        var_mean <- mean(worldcup[ , vars[i]])
        var_mean <- round(var_mean, 1)
        out <- paste0("mean of ", vars[i], ": ", var_mean)
        print(out)
}
```

```
## [1] "mean of Time: 208.9"
## [1] "mean of Shots: 2.3"
## [1] "mean of Passes: 84.5"
## [1] "mean of Tackles: 4.2"
## [1] "mean of Saves: 0.7"
```

Loops

Often, it's convenient to create a data set to fill up as you loop through:

```r
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
my_df <- data.frame(variable = vars, mean = NA)
for(i in 1:nrow(my_df)){
        var_mean <- mean(worldcup[ , vars[i]])
        my_df[i , "mean"] <- round(var_mean, 1)
}
```

Loops

```r
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
(my_df <- data.frame(variable = vars, mean = NA))
```

```
##   variable mean
## 1     Time   NA
## 2    Shots   NA
## 3   Passes   NA
## 4  Tackles   NA
## 5    Saves   NA
```

Loops

```r
i <- 1
(var_mean <- mean(worldcup[ , vars[i]]))
```

```
## [1] 208.8639
```

```r
my_df[i , "mean"] <- round(var_mean, 1)
my_df
```

```
##   variable  mean
## 1     Time 208.9
## 2    Shots    NA
## 3   Passes    NA
## 4  Tackles    NA
## 5    Saves    NA
```

Loops

```r
for(i in 1:nrow(my_df)){
        var_mean <- mean(worldcup[ , vars[i]])
        my_df[i , "mean"] <- round(var_mean, 1)
}
my_df
```

```
##   variable  mean
## 1     Time 208.9
## 2    Shots   2.3
## 3   Passes  84.5
## 4  Tackles   4.2
## 5    Saves   0.7
```

Loops

Note: This is a pretty simplistic example. There are some easier ways to have done this:

```
worldcup %>%
  summarize(Time = mean(Time), Passes = mean(Passes),
            Shots = mean(Shots), Tackles = mean(Tackles),
            Saves = mean(Saves)) %>%
  gather(key = var, value = mean) %>%
  mutate(mean = round(mean, 1))
```

```
##       var  mean
## 1    Time 208.9
## 2  Passes  84.5
## 3   Shots   2.3
## 4 Tackles   4.2
## 5   Saves   0.7
```

Loops

Note: This is a pretty simplistic example. There are some easier ways to have done this:

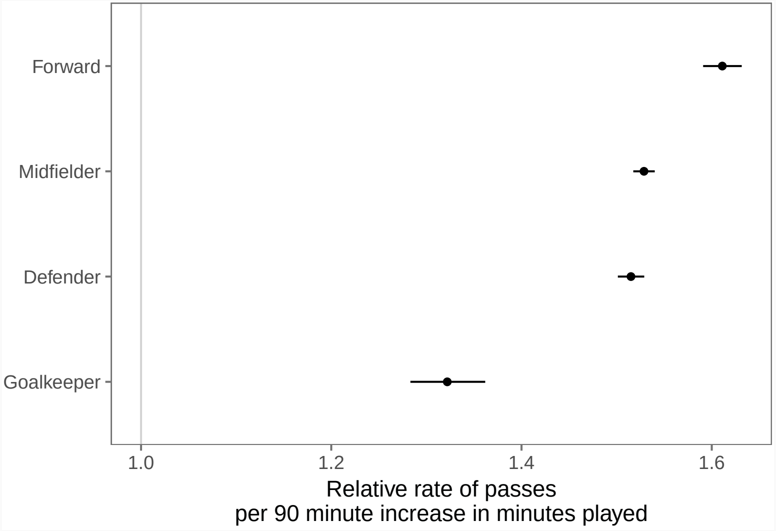
```
means <- apply(worldcup[ , vars], 2, mean)
(means <- round(means, 1))
```

```
##    Time   Shots  Passes Tackles   Saves
##   208.9     2.3    84.5     4.2     0.7
```

However, you can use this same looping process for much more complex tasks that you can't do as easily with `apply` or `dplyr` tools.

Loops

Loops can be very useful for more complex repeated tasks. For example:



Loops

Creating this graph requires:

- Create a subset limited to each of the four positions
- Fit a Poisson regression of Passes on Time within each subset
- Pull the regression coefficient and standard error from each model
- Use those values to calculate 95% confidence intervals
- Convert everything from log relative rate to relative rate
- Plot everything

Loops

Create a vector with the names of all positions. Create an empty data frame to store regression results.

```
(positions <- unique(worldcup$Position))
```

```
## [1] Midfielder Defender   Forward    Goalkeeper
## Levels: Defender Forward Goalkeeper Midfielder
```

```
(pos_est <- data.frame(position = positions,
                       est = NA, se = NA))
```

```
##     position est se
## 1 Midfielder  NA NA
## 2   Defender  NA NA
## 3    Forward  NA NA
## 4 Goalkeeper  NA NA
```

Loops

Loop through and fit a Poisson regression model for each subset of data. Save regression coefficients in the empty data frame.

```r
for(i in 1:nrow(pos_est)){
        pos_df <- worldcup %>%
          filter(Position == positions[i])
        pos_mod <- glm(Passes ~ Time,
                       data = pos_df,
                       family = poisson(link = "log"))
        pos_coefs <- summary(pos_mod)$coefficients[2, 1:2]
        pos_est[i, c("est", "se")] <- pos_coefs
}
pos_est[1:2, ]
```

```
##     position         est           se
## 1 Midfielder 0.004716096 4.185925e-05
## 2   Defender 0.004616260 5.192736e-05
```

Loops

Calculate 95% confidence intervals for log relative risk values.

```r
pos_est <- pos_est %>%
  mutate(lower_ci = est - 1.96 * se,
         upper_ci = est + 1.96 * se)

pos_est %>%
  select(position, est, lower_ci, upper_ci)
```

```
##     position         est    lower_ci    upper_ci
## 1 Midfielder 0.004716096 0.004634052 0.004798140
## 2   Defender 0.004616260 0.004514483 0.004718038
## 3    Forward 0.005299009 0.005158945 0.005439074
## 4 Goalkeeper 0.003101124 0.002770562 0.003431687
```

Loops

Calculate relative risk per 90 minute increase in minutes played.

```
pos_est <- pos_est %>%
  mutate(rr_est = exp(90 * est),
         rr_low = exp(90 * lower_ci),
         rr_high = exp(90 * upper_ci))
pos_est %>%
  select(position, rr_est, rr_low, rr_high)
```

```
##     position   rr_est   rr_low  rr_high
## 1 Midfielder 1.528747 1.517501 1.540077
## 2   Defender 1.515073 1.501258 1.529015
## 3    Forward 1.611090 1.590908 1.631527
## 4 Goalkeeper 1.321941 1.283192 1.361861
```

Loops

Re-level the `position` factor so the plot will be ordered from highest to lowest estimates.

```
pos_est <- arrange(pos_est, rr_est) %>%
        mutate(position = factor(position,
                                 levels = position))
pos_est %>% select(position, est)
```

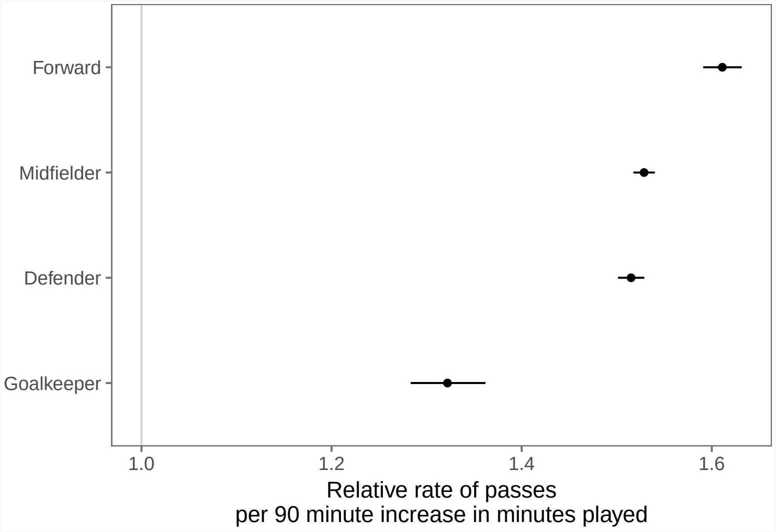
```
##     position         est
## 1 Goalkeeper 0.003101124
## 2   Defender 0.004616260
## 3 Midfielder 0.004716096
## 4    Forward 0.005299009
```

Loops

Create the plot:

```
ggplot(pos_est, aes(x = rr_low, y = position)) +
        geom_segment(aes(xend = rr_high, yend = position)) +
        geom_point(aes(x = rr_est, y = position)) +
        theme_few() +
        ylab("") +
        scale_x_continuous(paste("Relative rate of",
                                 "passes\nper 90 minute",
                                 "increase in minutes played"),
                           limits = c(1.0,
                                      max(pos_est$rr_high))) +
        geom_vline(aes(xintercept = 1), color = "lightgray")
```

Loops


Forward
Midfielder
Defender
Goalkeeper
1.0
1.2
1.4
1.6
Relative rate of passes
per 90 minute increase in minutes played

Other control structures

if / else

There are other control structures you can use in your R code. Two that you will commonly use within R functions are `if` and `ifelse` statements.

An `if` statement tells R that, **if** a certain condition is true, **do** run some code. For example, if you wanted to print out only odd numbers between 1 and 5, one way to do that is with an `if` statement:

```r
for(i in 1:5){
  if(i %% 2 == 1){
    print(i)
  }
}
```

```
## [1] 1
## [1] 3
## [1] 5
```

if / else

The `if` statement runs some code if a condition is true, but does nothing if it is false. If you'd like different code to run depending on whether the condition is true or false, you can us an if / else or an if / else if / else statement.

```
for(i in 1:5){
  if(i %% 2 == 1){
    print(i)
  } else {
    print(paste(i, "is even"))
  }
}
```

```
## [1] 1
## [1] "2 is even"
## [1] 3
## [1] "4 is even"
## [1] 5
```

if / else

What would this code do?

```
for(i in 1:100){
  if(i %% 3 == 0 & i %% 5 == 0){
    print("FizzBuzz")
  } else if(i %% 3 == 0){
    print("Fizz")
  } else if(i %% 5 == 0){
    print("Buzz")
  } else {
    print(i)
  }
}
```

if / else

If / else statements are extremely useful in functions.

In R, the `if` statement evaluates everything in the parentheses and, if that evaluates to `TRUE`, runs everything in the braces. This means that you can trigger code in an `if` statement with a single-value logical vector:

```
weekend <- TRUE
if(weekend){
  print("It's the weekend!")
}
```

```
## [1] "It's the weekend!"
```

This functionality can be useful with parameters you choose to include when writing your own functions (e.g., `print = TRUE`).

Control structures

The control structures you are most likely to use in data analysis with R are “for” loops and “if / else” statements. However, there are a few other control structures you may occasionally find useful:

- `next`
- `break`
- `while`

next

You can use the next structure to skip to the next round of a loop when a certain condition is met. For example, we could have used this code to print out odd numbers between 1 and 5:

```
for(i in 1:5){
  if(i %% 2 == 0){
    next
  }
  print(i)
}
```

```
## [1] 1
## [1] 3
## [1] 5
```

break

You can use `break` to break out of a loop if a certain condition is met. For example, the final code will break out of the loop once `i` is over 3, so it will only print the numbers 1 through 3:

```r
for(i in 1:5){
  if(i > 3){
    break
  }
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

while

```r
my_sum <- 1
while(my_sum < 10){
  my_sum <- my_sum * 2
  print(my_sum)
}
```

```
## [1] 2
## [1] 4
## [1] 8
## [1] 16
```

Functions

Functions

As you move to larger projects, you will find yourself using the same code a lot.

Examples include:

- Reading in data from a specific type of equipment (air pollution monitor, accelerometer)
- Running a specific type of analysis
- Creating a specific type of plot or map

If you find yourself cutting and pasting a lot, convert the code to a function.

Functions

Advantages of writing functions include:

- Coding is more efficient
- Easier to change your code (if you've cut and paste code and you want to change something, you have to change it everywhere)
- Easier to share code with others

Functions

You can name a function anything you want (although try to avoid names of preexisting-existing functions). You then define any inputs (arguments; separate multiple arguments with commas) and put the code to run in braces:

```
## Note: this code will not run
[function name] <- function([any arguments]){
        [code to run]
}
```

Functions

Here is an example of a very basic function. This function takes a number as input and adds 1 to that number.

```
add_one <- function(number){
        out <- number + 1
        return(out)
}

add_one(number = 3)
```

```
## [1] 4
```

```
add_one(number = -1)
```

```
## [1] 0
```

Functions

- Functions can input any type of R object (for example, vectors, data frames, even other functions and ggplot objects)
- Similarly, functions can output any type of R object
- When defining a function, you can set default values for some of the parameters
- You can explicitly specify the value to return from the function
- There are ways to check for errors in the arguments a user inputs to the function

Functions

For example, the following function inputs a data frame (`datafr`) and a one-element vector (`child_id`) and returns only rows in the data frame where it's `id` column matches `child_id`. It includes a default value for `datafr`, but not for `child_id`.

```
subset_nepali <- function(datafr = nepali, child_id){
  datafr <- datafr %>%
    filter(id == child_id)
  return(datafr)
}
```

Functions

If an argument is not given for a parameter with a default, the function will run using the default value for that parameter. For example:

```
subset_nepali(child_id = "120011")
```

```
##       id  sex   wt   ht age wt_over_13
## 1 120011 Male 12.8 91.2  41      FALSE
```

Functions

If an argument is not given for a parameter without a default, the function call will result in an error. For example:

```r
subset_nepali(datafr = nepali)
```

```
## Error in filter_impl(.data, quo): Evaluation error: argument
```

Functions

By default, the function will return the last defined object, although the choice of using `return` can affect printing behavior when you run the function. For example, I could have written the `subset_nepali` function like this:

```r
subset_nepali <- function(datafr = nepali, child_id){
  datafr <- datafr %>%
    filter(id == child_id)
}
```

Functions

In this case, the output will not automatically print out when you call the function without assigning it to an R object:

```
subset_nepali(child_id = "120011")
```

However, the output can be assigned to an R object in the same way as when the function was defined without `return`:

```
first_childs_data <- subset_nepali(child_id = "120011")
first_childs_data
```

```
##       id  sex   wt   ht age wt_over_13
## 1 120011 Male 12.8 91.2  41      FALSE
```

Functions

The `return` function can also be used to return an object other than the last defined object (although doesn't tend to be something you need to do very often). For example, if you did not use `return` in the following code, it will output "Test output":

```
subset_nepali <- function(datafr = nepali, child_id){
  datafr <- datafr %>%
    filter(id == child_id)
  a <- "Test output"
}
(subset_nepali(child_id = "120011"))
```

```
## [1] "Test output"
```

Functions

Conversely, you can use `return` to output `datafr`, even though it's not the last object defined:

```r
subset_nepali <- function(datafr = nepali, child_id){
  datafr <- datafr %>%
    filter(id == child_id)
  a <- "Test output"
  return(datafr)
}
subset_nepali(child_id = "120011")
```

```
##       id  sex   wt   ht age wt_over_13
## 1 120011 Male 12.8 91.2  41      FALSE
```

Functions

You can use `stop` to stop execution of the function and give the user an error message. For example, the `subset_nepali` function will fail if the user inputs a data frame that does not have a column named "id":

```
subset_nepali(datafr = data.frame(wt = rnorm(10)),
              child_id = "12011")
```

```
Error: comparison (1) is possible only for
atomic and list types
```

Functions

You can rewrite the function to stop if the input `datafr` does not have a column named "id":

```r
subset_nepali <- function(datafr = nepali, child_id){
  if(!("id" %in% colnames(datafr))){
    stop("`datafr` must include a column named `id`")
  }
  datafr <- datafr %>%
    filter(id == child_id)
  return(datafr)
}
subset_nepali(datafr = data.frame(wt = rnorm(10)),
              child_id = "12011")
```

```
Error in subset_nepali(datafr = data.frame(wt = rnorm(10)),
child_id = "12011") :
  `datafr` must include a column named `id`
```

Functions

The `stop` function is particularly important if the function would keep running with the wrong input, but would result in the wrong output.

You can also output warnings and messages using the functions `warning` and `message`.

Regular expressions

Regular expressions

For these examples, we'll use some data on passengers of the Titanic. You can load this data using:

```r
# install.packages("titanic")
library(titanic)
data("titanic_train")
```

We will be using the `stringr` package:

```r
library(stringr)
```

Regular expressions

This data includes a column called "Name" with passenger names. This column is somewhat messy and includes several elements that we might want to separate (last name, first name, title). Here are the first few values of "Name":

```
titanic_train %>% select(Name) %>% slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                    Name
##                                                   <chr>
## 1                               Braund, Mr. Owen Harris
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 3                                Heikkinen, Miss. Laina
```

Regular expressions

We've already done some things to manipulate strings. For example, if we wanted to separate "Name" into last name and first name (including title), we could actually do that with the `separate` function:

```r
titanic_train %>%
  select(Name) %>%
  slice(1:3) %>%
  separate(Name, c("last_name", "first_name"), sep = ", ")
```

```
## # A tibble: 3 x 2
##   last_name                                first_name
## *     <chr>                                     <chr>
## 1    Braund                           Mr. Owen Harris
## 2   Cumings Mrs. John Bradley (Florence Briggs Thayer)
## 3 Heikkinen                               Miss. Laina
```

Regular expressions

Notice that `separate` is looking for a regular pattern (“,”) and then doing something based on the location of that pattern in each string (splitting the string).

There are a variety of functions in R that can perform manipulations based on finding regular patterns in character strings.

Regular expressions

The `str_detect` function will look through each element of a character vector for a designated pattern. If the pattern is there, it will return `TRUE`, and otherwise `FALSE`. The convention is:

```
## Generic code
str_detect(string = [vector you want to check],
           pattern = [pattern you want to check for])
```

For example, to create a logical vector specifying which of the Titanic passenger names include “Mrs.”, you can call:

```r
mrs <- str_detect(titanic_train$Name, "Mrs.")
head(mrs)
```

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

Regular expressions

The result is a logical vector, so `str_detect` can be used in `filter` to subset data to only rows where the passenger's name includes "Mrs.":

```r
titanic_train %>%
  filter(str_detect(Name, "Mrs.")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                  Name
##                                                 <chr>
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 2        Futrelle, Mrs. Jacques Heath (Lily May Peel)
## 3   Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```

Regular expressions

There is an older, base R function called `grepl` that does something very similar (although note that the order of the arguments is reversed).

```
titanic_train %>%
  filter(grepl("Mrs.", Name)) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                      Name
##                                                     <chr>
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 2       Futrelle, Mrs. Jacques Heath (Lily May Peel)
## 3   Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```

Regular expressions

The `str_extract` function can be used to extract a string (if it exists) from each value in a character vector. It follows similar conventions to `str_detect`:

```
## Generic code
str_extract(string = [vector you want to check],
            pattern = [pattern you want to check for])
```

Regular expressions

For example, you might want to extract “Mrs.” if it exists in a passenger’s name:

```r
titanic_train %>%
  mutate(mrs = str_extract(Name, "Mrs.")) %>%
  select(Name, mrs) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                    Name   mrs
##                                                   <chr> <chr>
## 1                               Braund, Mr. Owen Harris  <NA>
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                                Heikkinen, Miss. Laina  <NA>
```

Notice that now we’re creating a new column (`mrs`) that either has “Mrs.” (if there’s a match) or is missing (`NA`) if there’s not a match.

Regular expressions

For this first example, we were looking for an exact string (“Mrs”). However, you can use patterns that match a particular pattern, but not an exact string. For example, we could expand the regular expression to find “Mr.” or “Mrs.”:

```
titanic_train %>%
  mutate(title = str_extract(Name, "Mr\\.|Mrs\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                  Name title
##                                                 <chr> <chr>
## 1                             Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                              Heikkinen, Miss. Laina  <NA>
```

Note that this pattern uses a special operator (|) to find one pattern **or**

Regular expressions

As a note, in regular expressions, all of the following characters are special characters that need to be escaped with backslashes if you want to use them literally:

```
. * + ^ ? $ \ | ( ) [ ] { }
```

Regular expressions

Notice that “Mr.” and “Mrs.” both start with “Mr”, end with “.”, and may or may not have an “s” in between.

```
titanic_train %>%
  mutate(title = str_extract(Name, "Mr(s)*\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                  Name title
##                                                 <chr> <chr>
## 1                             Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                              Heikkinen, Miss. Laina  <NA>
```

This pattern uses `(s)*` to match zero or more “s”s at this spot in the pattern.

Regular expressions

In the previous code, we found “Mr.” and “Mrs.”, but missed “Miss.”. We could tweak the pattern again to try to capture that, as well. For all three, we have the pattern that it starts with “M”, has some lowercase letters, and then ends with “.”.

```
titanic_train %>%
  mutate(title = str_extract(Name, "M[a-z]+\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                   Name title
##                                                  <chr> <chr>
## 1                              Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                               Heikkinen, Miss. Laina Miss.
```

Regular expressions

The last pattern used `[a-z]+` to match one or more lowercase letters. The `[a-z]`is a **character class**.

You can also match digits (`[0-9]`), uppercase letters (`[A-Z]`), just some letters (`[aeiou]`), etc.

You can negate a character class by starting it with `^`. For example, `[^0-9]` will match anything that **isn't** a digit.

Regular expressions

Sometimes, you want to match a pattern, but then only subset a part of it. For example, each passenger seems to have a title ("Mr.", "Mrs.", etc.) that comes after "," and before ".". We can use this pattern to find the title, but then we get some extra stuff with the match:

```
titanic_train %>%
  mutate(title = str_extract(Name, ",\\s[A-Za-z]*\\.\\s")) %>%
  select(title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##       title
##       <chr>
## 1   , Mr.
## 2  , Mrs.
## 3 , Miss.
```

As a note, in this pattern, `\\s` is used to match a space.

Regular expressions

We are getting things like ", Mr.", when we really want "Mr". We can use the `str_match` function to do this. We group what we want to extract from the pattern in parentheses, and then the function returns a matrix. The first column is the full pattern match, and each following column gives just what matches within the groups.

```r
head(str_match(titanic_train$Name,
          pattern = ",\\s([A-Za-z]*)\\.\\s"))
```

```
##      [,1]       [,2]  
## [1,] ", Mr. "   "Mr"  
## [2,] ", Mrs. "  "Mrs" 
## [3,] ", Miss. " "Miss"
## [4,] ", Mrs. "  "Mrs" 
## [5,] ", Mr. "   "Mr"  
## [6,] ", Mr. "   "Mr"  
```

Regular expressions

To get just the title, then, we can run:

```
titanic_train %>%
  mutate(title =
           str_match(Name, ",\\s([A-Za-z]*)\\.\\s")[ , 2]) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                  Name title
##                                                 <chr> <chr>
## 1                             Braund, Mr. Owen Harris    Mr
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)   Mrs
## 3                              Heikkinen, Miss. Laina  Miss
```

The `[ , 2]` pulls out just the second column from the matrix returned by `str_match`.

Regular expressions

Here are some of the most common titles:

```
titanic_train %>%
  mutate(title =
           str_match(Name, ",\\s([A-Za-z]*)\\.\\s")[ , 2]) %>%
  group_by(title) %>% summarize(n = n()) %>%
  arrange(desc(n)) %>% slice(1:5)
```

```
## # A tibble: 5 x 2
##    title     n
##    <chr> <int>
## 1     Mr   517
## 2   Miss   182
## 3    Mrs   125
## 4 Master    40
## 5     Dr     7
```

Regular expressions

The following slides have a few other examples of regular expressions in action with this dataset.

Get just names that start with (“^”) the letter “A”:

```
titanic_train %>%
  filter(str_detect(Name, "^A")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                       Name
##                                                      <chr>
## 1                                    Allen, Mr. William Henry
## 2                                  Andersson, Mr. Anders Johan
## 3 Asplund, Mrs. Carl Oscar (Selma Augusta Emilia Johansson)
```

Regular expressions

Get names with “II” or “III” (`{2,}` says to match at least two times):

```
titanic_train %>%
  filter(str_detect(Name, "I{2,}")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 2 x 1
##                                    Name
##                                   <chr>
## 1  Carter, Master. William Thornton II
## 2 Roebling, Mr. Washington Augustus II
```

Regular expressions

Get names with “Andersen” or “Anderson” (alternatives in square brackets):

```
titanic_train %>%
  filter(str_detect(Name, "Anders[eo]n")) %>%
  select(Name)
```

```
##                                                    Name
## 1 Andersen-Jensen, Miss. Carla Christine Nielsine
## 2                                Anderson, Mr. Harry
## 3                   Walker, Mr. William Anderson
## 4                    Olsvigen, Mr. Thor Anderson
## 5     Soholt, Mr. Peter Andreas Lauritz Andersen
```

Regular expressions

Get names that start with ("^" outside of brackets) the letters "A" and "B":

```r
titanic_train %>%
  filter(str_detect(Name, "^[AB]")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                         Name
##                        <chr>
## 1  Braund, Mr. Owen Harris
## 2 Allen, Mr. William Henry
## 3 Bonnell, Miss. Elizabeth
```

Regular expressions

Get names that end with (“$”) the letter “b” (either lowercase or uppercase):

```
titanic_train %>%
  filter(str_detect(Name, "[bB]$")) %>%
  select(Name)
```

```
##                            Name
## 1   Emir, Mr. Farred Chehab
## 2 Goldschmidt, Mr. George B
## 3            Cook, Mr. Jacob
## 4           Pasic, Mr. Jakob
```

Regular expression

Some useful regular expression operators include:

Operator	Meaning
.	Any character
	Match 0 or more times (greedy)
?	Match 0 or more times (non-greedy)
+	Match 1 or more times (greedy)
+?	Match 1 or more times (non-greedy)
^	Starts with (in brackets, negates)
$	Ends with
[. . .]	Character classes

Regular expressions

For more on these patterns, see:

- Help file for the `stringi-search-regex` function in the `stringi` package (which should install when you install `stringr`)
- Introduction to stringr by Hadley Wickham
- Handling and Processing Strings in R by Gaston Sanchez (seven chapter ebook)
- `http://gskinner.com/RegExr` and `http://www.txt2re.com`: Interactive tools for helping you build regular expression pattern strings