Lab7

Irlanda Ayon-Moreno

4/13/2018

Lab 7: Simple Functions and Conditionals

Gaston Sanchez

Learning Objectives

- Learn how to write simple functions
- Get into the habit of writing simple functions
- Get into the habit of documenting functions
- Make sure your functions work
- Use conditionals if-then-else

Introduction

In this lab you will practice writing simple functions, and some basic examples to make sure that the functions work as expected. Later in the course we will see how to write assertions (i.e. tests) for your functions in a more formal way.

In addition to writing the functions, you should also practice documenting your functions. Writing this type of documentation should become second nature. To do this, include roxygen comments such as:

- `#' @title Name of your function`
- `#' @description What the function does`
- `#' @param x input(s) of your function`
- `#' @return output of your function`

Before you start ...

By setting the global option `error = TRUE` you avoid the knitting process to be stopped in case a code chunk generates an error.

Since you will be writing a couple of functions with `stop()` statements, it is essential that you set up `error = TRUE`, otherwise `"knitr"` will stop knitting your `Rmd` file if it encounters an error.

Toy Example

Here's an example of how to write a function with its documentation:

```
#' @title area of rectangle
#' @description calculates the area of a rectangle
#' @param len length of the rectangle (numeric)
#' @param wid width of the rectangle (numeric)
#' @return computed area
rect_area <- function(len = 1, wid = 1) {
  if (len < 0) {
    stop("len must be positive")
  }
  if (wid < 0) {
    stop("wid must be positive")
  }
  area <- len * wid
  return(area)
}
```

Some tests:

```
# default
rect_area()
```

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

```
# len=2, wid=3
rect_area(len = 2, wid = 3)
```

```
## [1] 6
```

```
# bad len
rect_area(len = -2, wid = 3)
```

```
## Error in rect_area(len = -2, wid = 3): len must be positive
```

Simple Math Functions

Consider the following mathematical functions:

- $f(x) = x2$
- $g(x) = 2x + 5$

Write two functions `f()` and `g()` based on the previous equations. Don't forget to include roxygen comments to document your function!

```
# function f()
#' @title squared
#' @description multiplies a number times itself
```

```r
#' @param x (numeric)
#' @return computed product
f <- function(x) {
  x^2
}
```

```r
# function g()
#' @title equation
#' @description multiplies a number times 2 and adds 5 to the product
#' @param x (numeric)
#' @return computed answer
g <- function(x) {
  2 * x + 5
}
```

Test your functions with:

```r
f(2)     # 4
```

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

```r
f(-5)    # 25
```

```
## [1] 25
```

```r
g(0)     # 5
```

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

```r
g(-5/2)  # 0
```

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

Use your functions `f()` and `g()` to create the following composite functions:

- `fog()` for the composite function: $f \quad g(x)$
- `gof()` for the composite function: $g \quad f(x)$

```r
# function fog()
#' @title F of G
#' @description the function g is an input to the function f
#' @param x (numeric)
#' @return computed answer
fog <- function(x) {f(g(x))}
```

```r
# function gof()
#' @title G of F
#' @description the function f is an input to the function g
#' @param x (numeric)
#' @return computed answer
gof <- function(x) {g(f(x))}
```

Test your composite functions with:

```r
fog(2)     # 81
```

```
## [1] 81
```

```r
fog(-5)    # 25
```

```
## [1] 25
```

```r
gof(0)     # 5
```

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

```r
gof(-5/2)  # 17.5
```

```
## [1] 17.5
```

Pythagoras

The pythagoras formula is used to compute the length of the hypotenuse, c, of a right triangle with legs of length a and b.

Write a function `pythagoras()` that takes two arguments `a` and `b`, and returns the length of the hypotenuse. Don't forget to include roxygen comments to document your function!

```r
# pythagoras() function
#' @title pythagoras formula
#' @description computes the length of the hypotenuse
#' @param a length of one leg (numeric)
#' @param b length of second leg (numeric)
#' @return length of the hypotenuse
pythagoras <- function(a, b) {sqrt(a^2 + b^2)}

# pythagoras ver2
pythagoras <- function(a, b = a) {
  sqrt(a^2 + b^2)
}
```

Test your `pythagoras()` with two leg values: `pythagoras(3, 4)`

```r
# test
pythagoras(3, 4)
```

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

Modify your function `pythagoras()` so that argument `b` takes the same value of argument `a`. Test it with just one leg value: `pythagoras(5)`

```r
# test
pythagoras(5)
```

```
## [1] 7.071068
```

Area of a circle

Consider a circle with `radius = 2`. The area of this circle can be computed in R as:

```
# area of circle with radius 2
r <- 2
area <- pi * r^2
area
```

```
## [1] 12.56637
```

Write a function `circle_area()` that calculates the area of a circle. This function must take one argument `radius`. Give `radius` a default value of 1. Don't forget to include roxygen comments to document your function!

```
#' @title area of a circle
#' @description calculates the area of a circle
#' @param radius radius of a circle (numeric)
#' @return computed area
circle_area <- function(radius = 1) {
  if (radius < 0) {
    stop("radius must be positive")
  }
  area <- pi * radius^2
  return(area)
}
```

```
# default (radius 1)
circle_area()
```

```
## [1] 3.141593
```

```
# radius 3
circle_area(radius = 3)
```

```
## [1] 28.27433
```

Modify your `circle_area()` function in order to include a `stop()` statement. If `radius` is negative, then the function should stop with a message like: `"radius cannot be negative"`.

Test your modified `circle_area()` with `radius = -2`; the function should return a stop message:

```
# bad radius
circle_area(radius = -2)
```

```
## Error in circle_area(radius = -2): radius must be positive
```

Area of a cylinder

For a given cylinder of radius r and height h the area A is:

$A = 2\ \boldsymbol{r}h + 2$* **r*2

For example. Say you have a cylinder with radius = 2, and height = 3.

```
# cylinder variables
r = 2  # radius
h = 3  # height

# area of cylinder
2 * pi * r * h + 2 * pi * r^2
```

```
## [1] 62.83185
```

Notice that the formula of the area of a cylinder includes the area of a circle: * **r*2. Write a function `cylinder_area()`, that calls `circle_area()`, to compute the area of a cylinder.

This function must take two arguments: `radius` and `height`. Give both arguments a default value of 1. In addition, the function should stop if any of `radius` or `height` are negative.

```
#' @title area of a cylinder
#' @description calculates the area of a cylinder
#' @param radius radius of a cylinder (numeric)
#' @param height height of a cylinder (numeric)
#' @return computed area
cylinder_area <- function(radius = 1, height = 1) {
  if (radius < 0) {
    stop("radius must be positive")
  }
  if (height < 0) {
    stop("height must be positive")
  }
  area <- 2 * pi * radius * height + 2 * pi * radius^2
  return(area)
}
```

For instance:

```
# default (radius 1, height 1)
cylinder_area()
```

```
## [1] 12.56637
```

```
# radius 2, height 3
cylinder_area(radius = 2, height = 3)
```

```
## [1] 62.83185
```

These should return an error message:

```
# bad radius
cylinder_area(radius = -2, height = 1)
```

```
## Error in cylinder_area(radius = -2, height = 1): radius must be positive
```

```r
# bad height
cylinder_area(radius = 2, height = -1)
```

```
## Error in cylinder_area(radius = 2, height = -1): height must be positive
```

```r
# bad radius and height
cylinder_area(radius = -2, height = -1)
```

```
## Error in cylinder_area(radius = -2, height = -1): radius must be positive
```

Volume of a cylinder

For a given cylinder of radius r and height h the volume V is:

$V = * **r2h*$

Write a function `cylinder_volume()`, that calls `circle_area()`, to compute the volume of a cylinder. This function must take two arguments: `radius` and `height`. Give both arguments a default value of 1.

```r
#' @title volume of a cylinder
#' @description calculates the volume of a cylinder
#' @param radius radius of a cylinder (numeric)
#' @param height height of a cylinder (numeric)
#' @return computed volume
cylinder_volume <- function(radius = 1, height = 1) {
  circle_area(radius) * height
}
```

For example:

```r
# default (radius 1, height 1)
cylinder_volume()
```

```
## [1] 3.141593
```

```r
cylinder_volume(radius = 3, height = 10)
```

```
## [1] 282.7433
```

```r
cylinder_volume(height = 10, radius = 3)
```

```
## [1] 282.7433
```

Unit Conversion Formulas

The following exercises involve writing simple functions to convert from one type of unit to other.

Miles to Kilometers

Write a function `miles2kms()` that converts miles into kilometers: 1 mile is equal to 1.6 kilometers. Give the argument a default value of 1.

```
#' @title miles converter
#' @description converts miles into kilometers
#' @param miles miles to convert (numeric)
#' @return kilometers
miles2kms <- function(miles = 1) {miles * 1.6}
```

Use `miles2kms()` to obtain mile conversions, in order to create a table (i.e. data frame). The first ten rows range from 1 to 10 miles, and then from 10 to 100 in 10 mile steps. The second column corresponds to kms.

```
data.frame(miles = c(1:9, seq(10, 100, 10)),
           kms = miles2kms(miles = c(1:9, seq(10, 100, 10))))
```

```
##    miles   kms
## 1      1   1.6
## 2      2   3.2
## 3      3   4.8
## 4      4   6.4
## 5      5   8.0
## 6      6   9.6
## 7      7  11.2
## 8      8  12.8
## 9      9  14.4
## 10    10  16.0
## 11    20  32.0
## 12    30  48.0
## 13    40  64.0
## 14    50  80.0
## 15    60  96.0
## 16    70 112.0
## 17    80 128.0
## 18    90 144.0
## 19   100 160.0
```

Gallons to Liters, and viceversa

Write a function `gallon2liters()` that converts gallons to liters: 1 gallon is equal to 3.78541 liters:

```
#' @title gallon converter
#' @description converts gallons to liters
#' @param gallons number of gallons (numeric)
#' @return liters
gallons2liters <- function(gallons) {
  gallons * 3.78541
```

```
}
```

Use `gallon2liters()` to make an inverse function `liters2gallons()`.

```
#' @title liters converter
#' @description converts liters to gallons
#' @param liters number of liters (numeric)
#' @return gallons
liters2gallons <- function(liters = 1) {
  1 / gallons2liters(liters)
}
```

Use `liters2gallons()` to obtain liter conversions, in order to create a table (i.e. data frame). The first ten rows range from 1 to 10 liters, and then from 10 to 100 in 10 liter steps. The second column corresponds to gallons.

```
data.frame(liters = c(1:9, seq(10, 100, 10)),
           gallons = liters2gallons(c(1:9, seq(10, 100, 10)))
           )
```

```
##    liters     gallons
## 1       1 0.264172177
## 2       2 0.132086088
## 3       3 0.088057392
## 4       4 0.066043044
## 5       5 0.052834435
## 6       6 0.044028696
## 7       7 0.037738882
## 8       8 0.033021522
## 9       9 0.029352464
## 10     10 0.026417218
## 11     20 0.013208609
## 12     30 0.008805739
## 13     40 0.006604304
## 14     50 0.005283444
## 15     60 0.004402870
## 16     70 0.003773888
## 17     80 0.003302152
## 18     90 0.002935246
## 19    100 0.002641722
```

liters	gallons
1	0.2641722
2	0.5283444
...	...
10	2.6417218
20	5.283444
...	...
100	26.417218

Seconds to Years

According to Wikipedia, in 2015 the life expectancy of a person born in the US was 79 years. Consider the following question: Can a newborn baby in USA expect to live for one billion (109) seconds?

To answer this question, write a function `seconds2years()` that takes a number in seconds and returns the equivalent number of years. Test the function with `seconds2years(1000000000)`

```
seconds2years <- function(sec = 1) {
  sec / (365 * 24 * 60 * 60)
}
```

Gaussian Function

The Gaussian (Normal) function, given in the equation below, is one of the most widely used functions in science and statistics:

The parameters and are real numbers, where must be greater than zero.

Make a function `gaussian()` that takes three arguments: `x`, `m`, and `s`. Evaluate the function with $m = 0$, $s = 2$, and $x = 1$.

```
# gaussian
gaussian <- function(x = 1, m = 0, s = 1) {
  constant <- 1/(s * sqrt(2*pi))
  constant * exp(-1/2 * ((x - m) /s)^2)
}
```

Test your `gaussian()` function and compare it with the R function `dnorm()`

```
# compare with dnorm()
gaussian(x = 1, m = 0, s = 2)
```

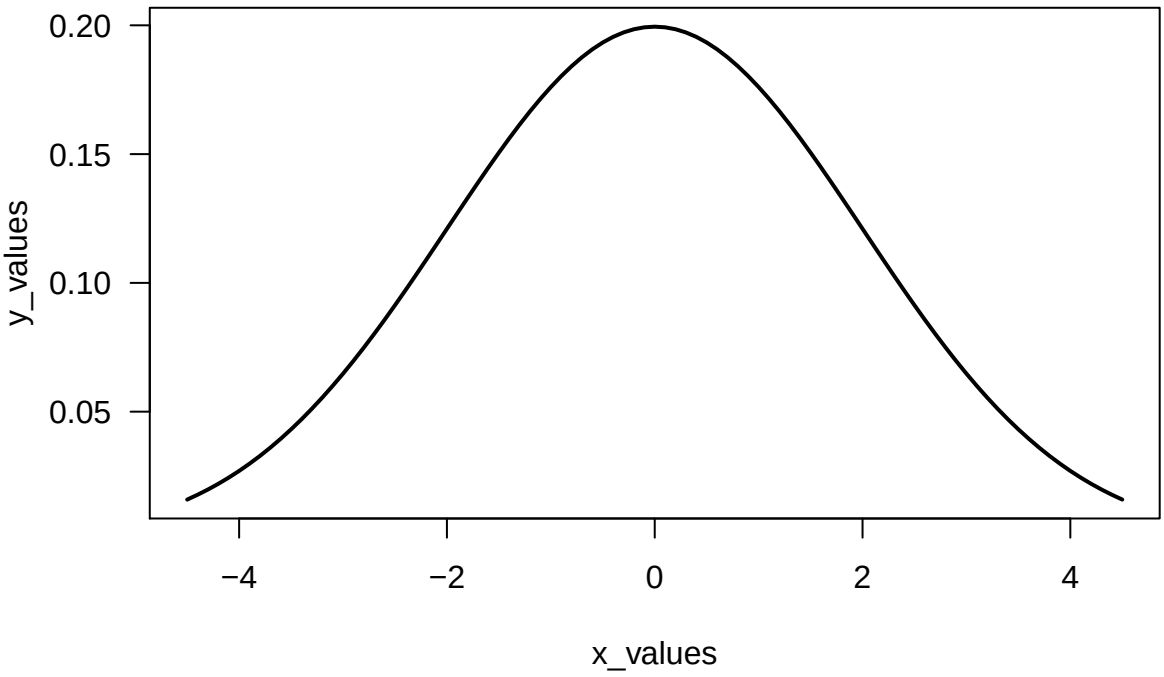
```
## [1] 0.1760327
```

```
dnorm(x = 1, mean = 0, sd = 2)
```

```
## [1] 0.1760327
```

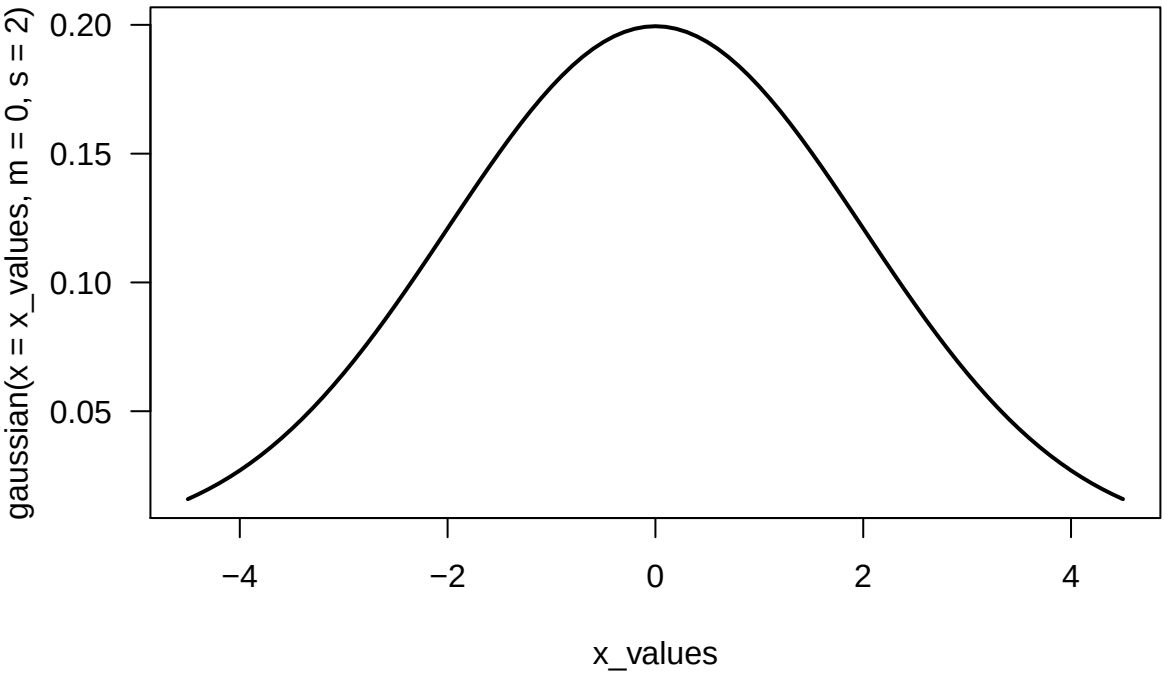
Now try `gaussian()` with a vector `seq(-4.5, 4.5, by = 0.1)`, and pass the values to `plot()` to get a normal curve. Here's code with values obtained from `dnorm()`

```
# you should get a plot like this one
x_values <- seq(from = -4.5, to = 4.5, by = 0.1)
y_values <- dnorm(x_values, mean = 0, sd = 2)
plot(x_values, y_values, las = 1, type = "l", lwd = 2)
```



Your turn:

```
plot(x_values, gaussian(x = x_values, m = 0, s = 2), las = 1, type = "l", lwd = 2)
```



Polynomials

In this problem we want to see whether the graph of a given polynomial will cross or touch the x axis in a given interval.

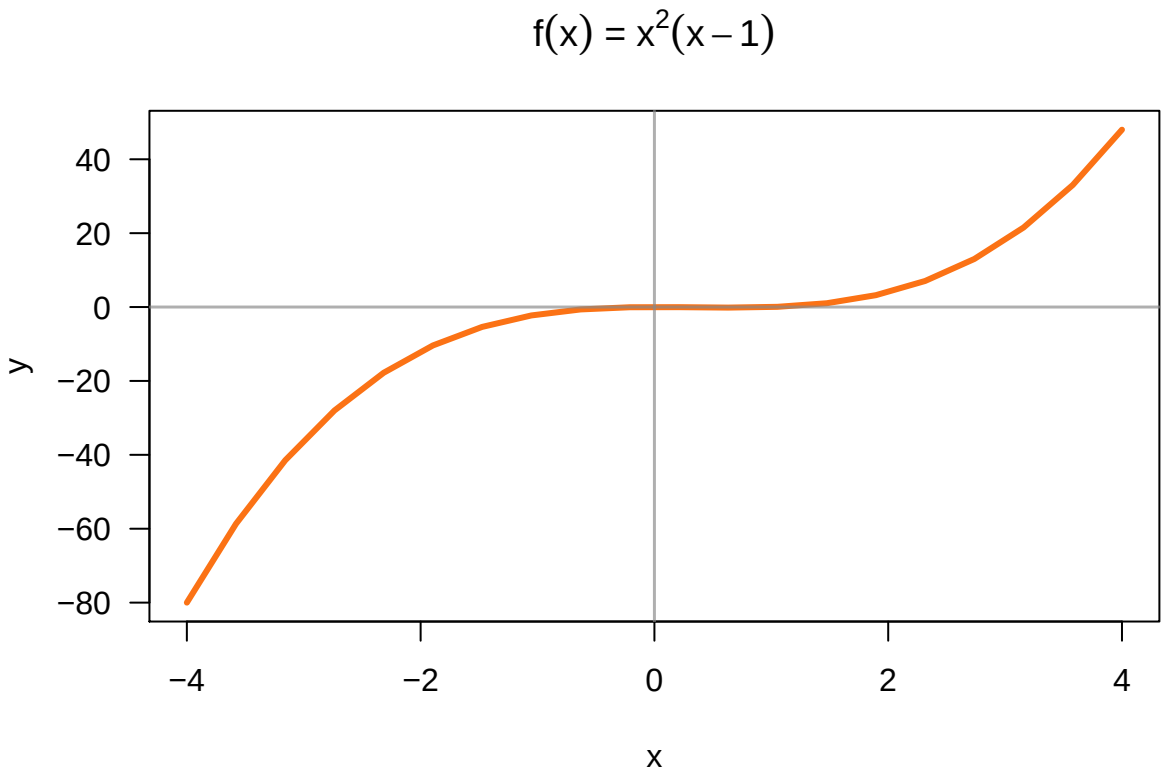
Let's begin with the polynomial: $f(x)=x2(x-1)$. The first thing to do is write a function for the polynomial, for instance:

```
poly1 <- function(x) {
  (x^2) * (x - 1)
}
```

Once you have a function for the polynomial, you can create a set of pairs of points x and $y = f(x)$, and then use them to graph the polynomial

```
# set of points
x <- seq(-4, 4, length.out = 20)
y <- poly1(x)

# graph polynomial
plot(x, y, type = 'l', lwd = 3, col = "#FB7215", las = 1)
abline(h = 0, v = 0, col = '#888888aa', lwd = 1.5)
title(main = expression(paste(f(x), ' = ', x^2, (x - 1))))
```



Write functions and graph the following polynomials in the x-axis interval -4 to 4:

1. $f(x)=x3$
2. $f(x)=(x2-1)(x+3)3$
3. $f(x)=(x2-1)(x2-9)$

```
poly_a <- function(x) {x^3}
poly_b <- function(x) {(x^2 - 1) * (x + 3)^3}
poly_c <- function(x) {(x^2 - 1) * (x^2 - 9)}
```

Descriptive Statistics

Write a function `descriptive()` that takes a numeric vector as input, and returns a named vector with the following descriptive statistics:

- `min`: minimum
- `q1`: first quartile (Q2)
- `median`: median
- `mean`: mean
- `q3`: third quartile (Q3)
- `max`: maximum
- `range`: range or span (max - min)
- `iqr`: interquartile range (IQR)
- `sd`: standard deviation

```r
# your descriptive() function
descriptive <- function(x, na.rm = FALSE) {
  c(
    "min" = min(x, na.rm = na.rm),
    "q1" = quantile(x, probs = 0.25, na.rm = na.rm),
    "median" = median(x, na.rm = na.rm),
    "mean" = mean(x, na.rm = na.rm),
    "q3" = quantile(x, probs = 0.75, na.rm = na.rm),
    "max" = max(x, na.rm = na.rm),
    "range" = max(x, na.rm = na.rm) - min(x, na.rm = na.rm),
    "iqr" = IQR(x, na.rm = na.rm),
    "sd" = sd(x, na.rm = na.rm)
  )
}
```

If Conditionals

Write R code that will "squish" a number into the interval

$$0, 100$$

, so that a number less than 0 is replaced by 0 and a number greater than 100 is replaced by 100.

```r
z <- 100*pi
# Fill in the following if-else statements. You may (or may not)
# have to add or subtract else if or else statements.
if (TRUE) { # Replace TRUE with a condition.

} else if (TRUE) { # Replace TRUE with a condition.
```

```
} else {

}
```

```
## NULL
```

Multiple If's

A common situation involves working with multiple conditions at the same time. You can chain multiple if-else statements:

```
y <- 1 # Change this value!

if (y > 0) {
  print("positive")
} else if (y < 0) {
  print("negative")
} else {
  print("zero?")
}
```

```
## [1] "positive"
```

Even number

Write a function `is_even()` that determines whether a number is even (i.e. multiple of 2). If the input number is even, the output should be `TRUE`. If the input number is odd, the output should be `FALSE`. If the input is not a number, the output should be `NA`

```
is_even <- function(x) {
  if (is.numeric(x)) {
    return(x %% 2 == 0)
  } else {
    return(NA)
  }
}
```

Test your function:

```
# even number
is_even(10)
```

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

```
# odd number
is_even(33)
```

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

```r
# not a number
is_even('a')
```

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

Odd number

Use your function `is_even()` to write a function `is_odd()` that determines if a number is odd (i.e. not a multiple of 2). If a number is odd, the output should be `TRUE`; if a number is even the output should be `FALSE`; if the input is not a number the output should be `NA`

```r
is_odd <- function(x) {
  !is_even(x)
}
```

Test `is_odd()` with the following cases:

```r
# odd number
is_odd(1)
```

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

```r
# even number
is_odd(4)
```

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

```r
# not a number
is_odd('a')
```

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

Switch

Working with multiple chained if's becomes cumbersome. Consider the following example that uses several if's to convert a day of the week into a number:

```r
# Convert the day of the week into a number.
day <- "Tuesday" # Change this value!

if (day == 'Sunday') {
  num_day <- 1
} else {
  if (day == "Monday") {
    num_day <- 2
  } else {
    if (day == "Tuesday") {
      num_day <- 3
    } else {
      if (day == "Wednesday") {
```

```
        num_day <- 4
      } else {
        if (day == "Thursday") {
          num_day <- 5
        } else {
          if (day == "Friday") {
            num_day <- 6
          } else {
            if (day == "Saturday") {
              num_day <- 7
            }
          }
        }
      }
    }
  }
}


num_day
```

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

Working with several nested if's like in the example above can be a nigthmare.

In R, you can get rid of many of the braces like this:

```
# Convert the day of the week into a number.
day <- "Tuesday" # Change this value!

if (day == 'Sunday') {
  num_day <- 1
} else if (day == "Monday") {
  num_day <- 2
} else if (day == "Tuesday") {
  num_day <- 3
} else if (day == "Wednesday") {
  num_day <- 4
} else if (day == "Thursday") {
  num_day <- 5
} else if (day == "Friday") {
  num_day <- 6
} else if (day == "Saturday") {
  num_day <- 7
}

num_day
```

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

Your turn: a grading function

Write a function `grade()` that takes a `score` argument (i.e. a numeric value between 0 and 100), and returns a letter grade (i.e. character) based on the following grading scheme:

score	grade
90 – 100	`"A"`
[80 - 90)	`"B"`
[70 - 80)	`"C"`
[60 - 70)	`"D"`
< 60	`"F"`

```
grade <- function(score) {
  if (score >= 90) {
    return("A")
  } else if (score >= 80 & score < 90) {
    return("B")
  } else if (score >= 70 & score < 80) {
    return("C")
  } else if (score >= 60 & score < 70) {
    return("D")
  } else {
    return("F")
  }
}
```

You should be able to call your `grade()` function like this:

```
# grade "A"
grade(score = 90)
```

```
## [1] "A"
```

```
# grade "B"
grade(score = 89.9999)
```

```
## [1] "B"
```

```
# grade "C"
grade(score = 70.0000001)
```

```
## [1] "C"
```

```
# grade "F"
grade(score = 50)
```

```
## [1] "F"
```

Modify your `grade()` function to include a `stop()` condition when the input `score` value is less than zero or greater than 100. The error message should say something like:

```
"score must be a number between 0 and 100"
```

```
grade <- function(score) {
  if (grade < 0 | grade > 100) {
    stop("score must be a number between 0 and 100")
  }
  if (score >= 90) {
    return("A")
  } else if (score >= 80 & score < 90) {
    return("B")
  } else if (score >= 70 & score < 80) {
    return("C")
  } else if (score >= 60 & score < 70) {
    return("D")
  } else {
    return("F")
  }
}
```

The `switch()` function

Consider again the chain of if-then-else's used to convert a name day into a number dat. We have too many if's, and there's a lot of repetition in the code. If you find yourself using many if-else statements with identical structure for slightly different cases, you may want to consider a **switch** statement instead:

```
# Convert the day of the week into a number.
day <- "Tuesday" # Change this value!

switch(day, # The expression to be evaluated.
  Sunday = 1,
  Monday = 2,
  Tuesday = 3,
  Wednesday = 4,
  Thursday = 5,
  Friday = 6,
  Saturday = 7,
  NA) # an (optional) default value if there are no matches
```

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

Switch statements can also accept integer arguments, which will act as indices to choose a corresponding element:

```
# Convert a number into a day of the week.
day_num <- 3 # Change this value!

switch(day_num,
```

```
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday")
```

```
## [1] "Tuesday"
```

Converting Miles to other units

The table below shows the different formulas for converting miles (mi) into other scales:

Units	Formula
Inches	mi x 63360
Feet	mi x 5280
Yards	mi x 1760
Meters	mi / 0.00062137
Kms	mi / 0.62137

Write the following five functions for each type of conversion. Each function must take one argument `x` with default value: `x = 1`.

- `miles2inches()`
- `miles2feet()`
- `miles2yards()`
- `miles2meters()`
- `miles2kms()`

```
# converting miles
miles2inches <- function(x = 1) {
  x * 63360
}

miles2feet <- function(x = 1) {
  x * 5280
}

miles2yards <- function(x = 1) {
  x * 1760
}

miles2meters <- function(x = 1) {
  x / 0.00062137
}
```

```r
miles2kms <- function(x = 1) {
  x / 0.62137
}
```

For example:

```r
miles2inches(2)
```

```
## [1] 126720
```

```r
miles2feet(2)
```

```
## [1] 10560
```

```r
miles2yards(2)
```

```
## [1] 3520
```

```r
miles2meters(2)
```

```
## [1] 3218.694
```

```r
miles2kms(2)
```

```
## [1] 3.218694
```

Using switch()

Create a function `convert()` that converts miles into the specified units. Use `switch()` and the previously defined functions—`miles2inches()`, `miles2feet()`, …, `miles2kms`—to define `convert()`. Use two arguments: `x` and `to`, like this:

```r
convert(40, to = "in")
```

By default, `to = "km"`, but it can take values such as `"in"`, `"ft"`, `"yd"`, or `"m"`.

```r
convert <- function(x, to = "km") {
  switch(to,
         "in" = miles2inches(x),
         "ft" = miles2feet(x),
         "yd" = miles2yards(x),
         "m" = miles2meters(x),
         "km" = miles2kms(x))
}
```

Test `convert()` with:

```r
convert(3, "in")
```

```
## [1] 190080
```

```r
convert(3, "ft")
```

```
## [1] 15840
```

```r
convert(3, "yd")
```

```
## [1] 5280
```

```r
convert(3, "m")
```

```
## [1] 4828.041
```

```r
convert(3, "km")
```

```
## [1] 4.828041
```

```r
x <- 1:5

if (x > 0) {
  print("positive")
} else {
  print("not positive")
}
```

```
## Warning in if (x > 0) {: the condition has length > 1 and only the first
## element will be used
```

```
## [1] "positive"
```