YaRrr! The Pirate's Guide to R

Nathaniel D. Phillips

2017-02-27

Contents

Chapter 1

Preface

Chapter 2

Getting Started

Chapter 3

Jump In!

Chapter 4

The Basics

Chapter 5

Scalers and vectors

Chapter 6

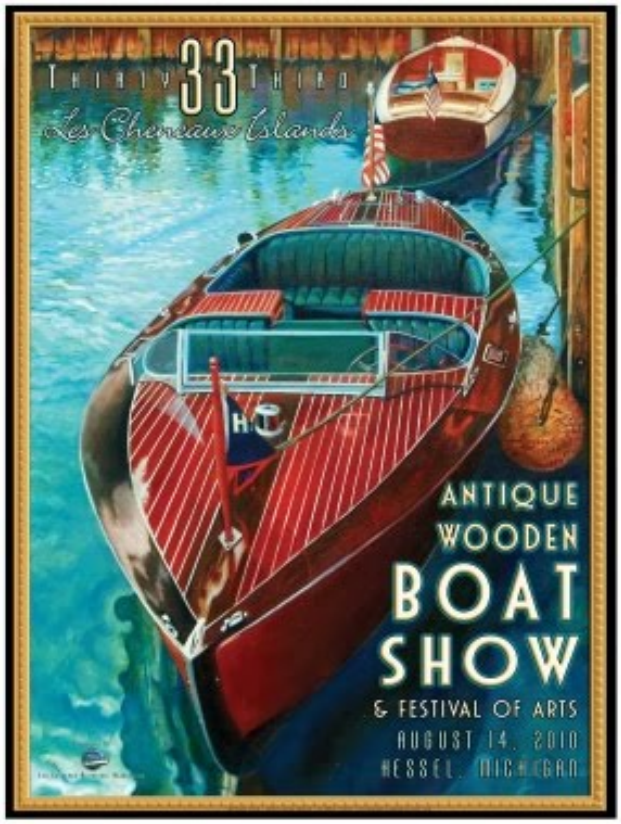
Vector functions

Chapter 7

Indexing Vectors with [ ]

boat.names	boat.colors	boat.ages	boat.prices	boat.costs
a	black	143	53	52
b	green	53	87	80
c	pink	356	54	20
d	blue	23	66	100
e	blue	647	264	189
f	green	24	32	12
g	green	532	532	520
h	yellow	43	58	68
i	black	66	99	80
j	black	86	132	100

```
# Boat sale. Creating the data vectors
boat.names <- c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j")
boat.colors <- c("black", "green", "pink", "blue", "blue",
                 "green", "green", "yellow", "black", "black")
boat.ages <- c(143, 53, 356, 23, 647, 24, 532, 43, 66, 86)
boat.prices <- c(53, 87, 54, 66, 264, 32, 532, 58, 99, 132)
boat.costs <- c(52, 80, 20, 100, 189, 12, 520, 68, 80, 100)

# What was the price of the first boat?
boat.prices[1]
## [1] 53

# What were the ages of the first 5 boats?
boat.ages[1:5]
## [1] 143  53 356  23 647

# What were the names of the black boats?
boat.names[boat.colors == "black"]
## [1] "a" "i" "j"

# What were the prices of either green or yellow boats?
boat.prices[boat.colors == "green" | boat.colors == "yellow"]
## [1]  87  32 532  58

# Change the price of boat "s" to 100
boat.prices[boat.names == "s"] <- 100
```




```
# What was the median price of black boats less than 100 years old?
median(boat.prices[boat.colors == "black" & boat.ages < 100])
## [1] 115.5

# How many pink boats were there?
sum(boat.colors == "pink")
## [1] 1

# What percent of boats were older than 100 years old?
mean(boat.ages < 100)
## [1] 0.6
```

By now you should be a whiz at applying functions like `mean()` and `table()` to vectors. However, in many analyses, you won't want to calculate statistics of an entire vector. Instead, you will want to access specific *subsets* of values of a vector based on some criteria. For example, you may want to access values in a specific location in the vector (i.e.; the first 10 elements) or based on some criteria within that vector (i.e.; all values greater than 0), or based on criterion from values in a *different* vector (e.g.; All values of age where sex is Female). To access specific values of a vector in R, we use *indexing* using brackets `[]`. In general, whatever you put inside the brackets, tells R which values of the vector object you want. There are two main ways that you can use indexing to access subsets of data in a vector: numerical and logical indexing.

7.1 Numerical Indexing

With numerical indexing, you enter a vector of integers corresponding to the values in the vector you want to access in the form `a[index]`, where `a` is the vector, and `index` is a vector of index values. For example, let's use numerical indexing to get values from our boat vectors.

```
# What is the first boat name?
boat.names[1]
## [1] "a"

# What are the first five boat colors?
boat.colors[1:5]
## [1] "black" "green" "pink"  "blue"  "blue"

# What is every second boat age?
boat.ages[seq(1, 5, by = 2)]
## [1] 143 356 647
```

You can use any indexing vector as long as it contains integers. You can even access the same elements multiple times:

```
# What is the first boat age (3 times)
boat.ages[c(1, 1, 1)]
## [1] 143 143 143
```

It it makes your code clearer, you can define an indexing object before doing your actual indexing. For example, let's define an object called `my.index` and use this object to index our data vector:

```
my.index <- 3:5
boat.names[my.index]
## [1] "c" "d" "e"
```

7.2 Logical Indexing

The second way to index vectors is with *logical vectors*. A logical vector is a vector that *only* contains TRUE and FALSE values. In R, true values are designated with TRUE, and false values with FALSE. When you index a vector with a logical vector, R will return values of the vector for which the indexing vector is TRUE. If that was confusing, think about it this way: a logical vector, combined with the brackets `[ ]`, acts as a *filter* for the vector it is indexing. It only lets values of the vector pass through for which the logical vector is TRUE.

You could create logical vectors directly using `c()`. For example, I could access every other value of the following vector as follows:

```
a <- c(1, 2, 3, 4, 5)
a[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
## [1] 1 3 5
```

As you can see, R returns all values of the vector `a` for which the logical vector is TRUE.

However, creating logical vectors using `c()` is tedious. Instead, it's better to create logical vectors from *existing vectors* using comparison operators like < (less than), == (equals to), and != (not equal to). A complete list of the most common comparison operators is in Figure~**??**. For example, let's create some logical vectors from our `boat.ages` vector:

```
# Which ages are > 100?
boat.ages > 100
```

Figure 7.1: Logical indexing. Good for R aliens and R pirates.

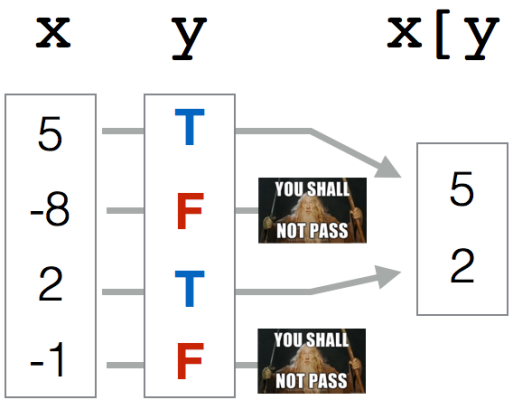


Figure 7.2: FALSE values in a logical vector are like lots of mini-Gandolfs. In this example, I am indexing a vector x with a logical vector y (y for example could be $x > 0$, so all positive values of x are TRUE and all negative values are FALSE). The result is a vector of length 2, which are the values of x for which the logical vector y was true. Gandolf stopped all the values of x for which y was FALSE.

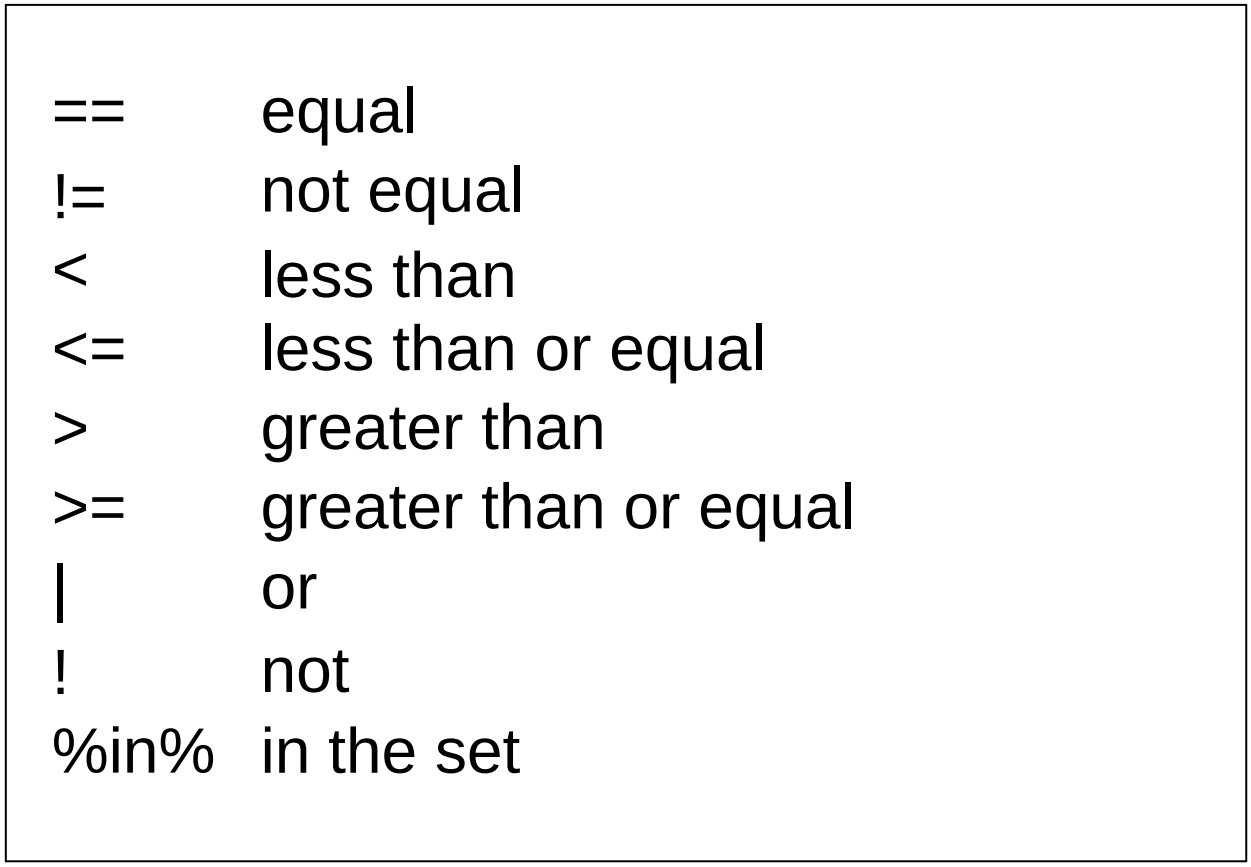


Figure 7.3: Logical comparison operators in R

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

# Which ages are equal to 23?
boat.ages == 23
##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE

# Which boat names are equal to c?
boat.names == "c"
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

You can also create logical vectors by comparing a vector to another vector of the same length. When you do this, R will compare values in the same position (e.g.; the first values will be compared, then the second values, etc.). For example, we can compare the `boat.cost` and `boat.price` vectors to see which boats sold for a higher price than their cost:

```
# Which boats had a higher price than cost?
boat.prices > boat.costs
##  [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE

# Which boats had a lower price than cost?
boat.prices < boat.costs
##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE
```

Once you've created a logical vector using a comparison operator, you can use it to index any vector with the same length. Here, I'll use logical vectors to get the prices of boats whose ages were greater than 100:

```
# What were the prices of boats older than 100?
boat.prices[boat.ages > 100]
## [1]  53  54 264 532
```

Here's how logical indexing works step-by-step:

```
# Which boat prices are greater than 100?
boat.ages > 100
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE

# Writing the logical index by hand (you'd never do this!)
boat.prices[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, FALSE, FALSE)]
## [1]  53  54 264 532

# Doing it all in one step! You get the same answer
boat.prices[boat.ages > 100]
## [1]  53  54 264 532
```

7.2.1 `&` (and), `|` (or), `%in%`

In addition to using single comparison operators, you can combine multiple logical vectors using the OR (which looks like `|` and AND `\&` commands. The OR `|` operation will return TRUE if any of the logical vectors is TRUE, while the AND `&` operation will only return TRUE if all of the values in the logical vectors is TRUE. This is especially powerful when you want to create a logical vector based on criteria from multiple vectors.

For example, let's create a logical vector indicating which boats had a price greater than 200 OR less than 100, and then use that vector to see what the names of these boats were:

```
# Which boats had prices greater than 400 OR less than 100?
boat.prices > 200 | boat.prices < 100
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE

# What were the NAMES of these boats
boat.names[boat.prices > 200 | boat.prices < 100]
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

You can combine as many logical vectors as you want (as long as they all have the same length!):

```
# Boat names of boats with a color of black OR with a price > 100
boat.names[boat.colors == "black" | boat.prices > 100]
## [1] "a" "e" "g" "i" "j"

# Names of blue boats with a price greater than 200
boat.names[boat.colors == "blue" & boat.prices > 200]
## [1] "e"
```

You can combine as many logical vectors as you want to create increasingly complex selection criteria. For example, the following logical vector returns TRUE for cases where the boat colors are black OR brown, AND where the price was not equal to 100:

```
# Which boats were eithe black or brown, AND had a price less than 100?
(boat.colors == "black" | boat.colors == "brown") & boat.prices < 100
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE

# What were the names of these boats?
boat.names[(boat.colors == "black" | boat.colors == "brown") & boat.prices < 100]
## [1] "a" "i"
```

When using multiple criteria, make sure to use parentheses when appropriate. If I didn't use parentheses above, I would get a different answer.

The `%in%` operation helps you to easily create multiple OR arguments.Imagine you have a vector of categorical data that can take on many different values. For example, you could have a vector x indicating people's favorite letters.

```
x <- c("a", "t", "a", "b", "z")
```

Now, let's say you want to create a logical vector indicating which values are either a or b or c or d. You could create this logical vector with multiple | (OR) commands:

```
x == "a" | x == "b" | x == "c" | x == "d"
## [1]  TRUE FALSE  TRUE  TRUE FALSE
```

However, this takes a long time to write. Thankfully, the `%in%` operation allows you to combine multiple OR comparisons much faster. To use the `%in%` function, just put it in between the original vector, and a new vector of possible values. The `%in%` function goes through every value in the vector x, and returns TRUE if it finds it in the vector of possible values – otherwise it returns FALSE.

```
x %in% c("a", "b", "c", "d")
## [1]  TRUE FALSE  TRUE  TRUE FALSE
```

As you can see, the result is identical to our previous result.

7.2.2 Counts and percentages from logical vectors

Many (if not all) R functions will interpret TRUE values as 1 and FALSE values as 0. This allows us to easily answer questions like "How many values in a data vector are greater than 0?" or "What percentage of values are equal to 5?" by applying the `sum()` or `mean()` function to a logical vector.

We'll start with a vector x of length 10, containing 3 positive numbers and 5 negative numbers.

```
x <- c(1, 2, 3, -5, -5, -5, -5, -5)
```

We can create a logical vector to see which values are greater than 0:

```
x > 0
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
```

Now, we'll use `sum()` and `mean()` on that logical vector to see how many of the values in x are positive, and what percent are positive. We should find that there are 5 TRUE values, and that 50% of the values (5 / 10) are TRUE.

```
sum(x > 0)
## [1] 3
mean(x > 0)
## [1] 0.375
```

This is a *really* powerful tool. Pretty much *any* time you want to answer a question like "How many of X are Y" or "What percent of X are Y", you use `sum()` or `mean()` function with a logical vector as an argument.

7.2.3 Additional Logical functions

R has lots of special functions that take vectors as arguments, and return logical vectors based on multiple criteria. For example, you can use the `is.na()` function to test which values of a vector are missing. Table 7.1 contains some that I frequently use:

Table 7.1: Functions to create and use logical vectors.

Function	Description	Example	Result
`is.na(x)`	Which values in x are NA?	`is.na(c(2, NA, 5))`	FALSE, TRUE, FALSE
`is.finite(x)`	Which values in x are numbers?	`is.finite(c(NA, 89, 0))`	FALSE, TRUE, TRUE
`duplicated(x)`	Which values in x are duplicated?	`duplicated(c(1, 4, 1, 2))`	FALSE, FALSE, TRUE, FALSE
`which(x)`	Which values in x are TRUE?	`which(c(TRUE, FALSE, TRUE))`	1, 3

Logical vectors aren't just good for indexing, you can also use them to figure out which values in a vector satisfy some criteria. To do this, use the function `which()`. If you apply the function `which()` to a logical vector, R will tell you which values of the index are TRUE. For example:

```
# A vector of sex information
sex <- c("m", "m", "f", "m", "f", "f")
```

```
# Which values of sex are m?
which(sex == "m")
## [1] 1 2 4

# Which values of sex are f?
which(sex == "f")
## [1] 3 5 6
```

7.3 Changing values of a vector

Now that you know how to index a vector, you can easily change specific values in a vector using the assignment (<-) operation. To do this, just assign a vector of new values to the indexed values of the original vector:

Let's create a vector a which contains 10 1s:

```
a <- rep(1, 10)
```

Now, let's change the first 5 values in the vector to 9s by indexing the first five values, and assigning the value of 9:

```
a[1:5] <- 9
a
##  [1] 9 9 9 9 9 1 1 1 1 1
```

Now let's change the last 5 values to 0s. We'll index the values 6 through 10, and assign a value of 0.

```
a[6:10] <- 0
a
##  [1] 9 9 9 9 9 0 0 0 0 0
```

Of course, you can also change values of a vector using a logical indexing vector. For example, let's say you have a vector of numbers that should be from 1 to 10. If values are outside of this range, you want to set them to either the minimum (1) or maximum (10) value:

```
# x is a vector of numbers that should be from 1 to 10
x <- c(5, -5, 7, 4, 11, 5, -2)

# Assign values less than 1 to 1
x[x < 1] <- 1

# Assign values greater than 10 to 10
x[x > 10] <- 10

# Print the result!
x
## [1]  5  1  7  4 10  5  1
```

As you can see, our new values of x are now never less than 1 or greater than 10!

A note on indexing...

Technically, when you assign new values to a vector, you should always assign a vector of the same length as the number of values that you are updating. For example, given a vector a with 10 1s:

```
a <- rep(1, 10)
```

To update the first 5 values with 5 9s, we should assign a new vector of 5 9s


```
a[1:5] <- c(9, 9, 9, 9, 9)
a
##  [1] 9 9 9 9 9 1 1 1 1 1
```

However, if we repeat this code but just assign a single 9, R will repeat the value as many times as necessary to fill the indexed value of the vector. That's why the following code still works:

```
a[1:5] <- 9
a
##  [1] 9 9 9 9 9 1 1 1 1 1
```

In other languages this code wouldn't work because we're trying to replace 5 values with just 1. However, this is a case where R bends the rules a bit.

7.3.1 Ex: Fixing invalid responses to a Happiness survey

Assigning and indexing is a particularly helpful tool when, for example, you want to remove invalid values in a vector before performing an analysis. For example, let's say you asked 10 people how happy they were on a scale of 1 to 5 and received the following responses:

```
happy <- c(1, 4, 2, 999, 2, 3, -2, 3, 2, 999)
```

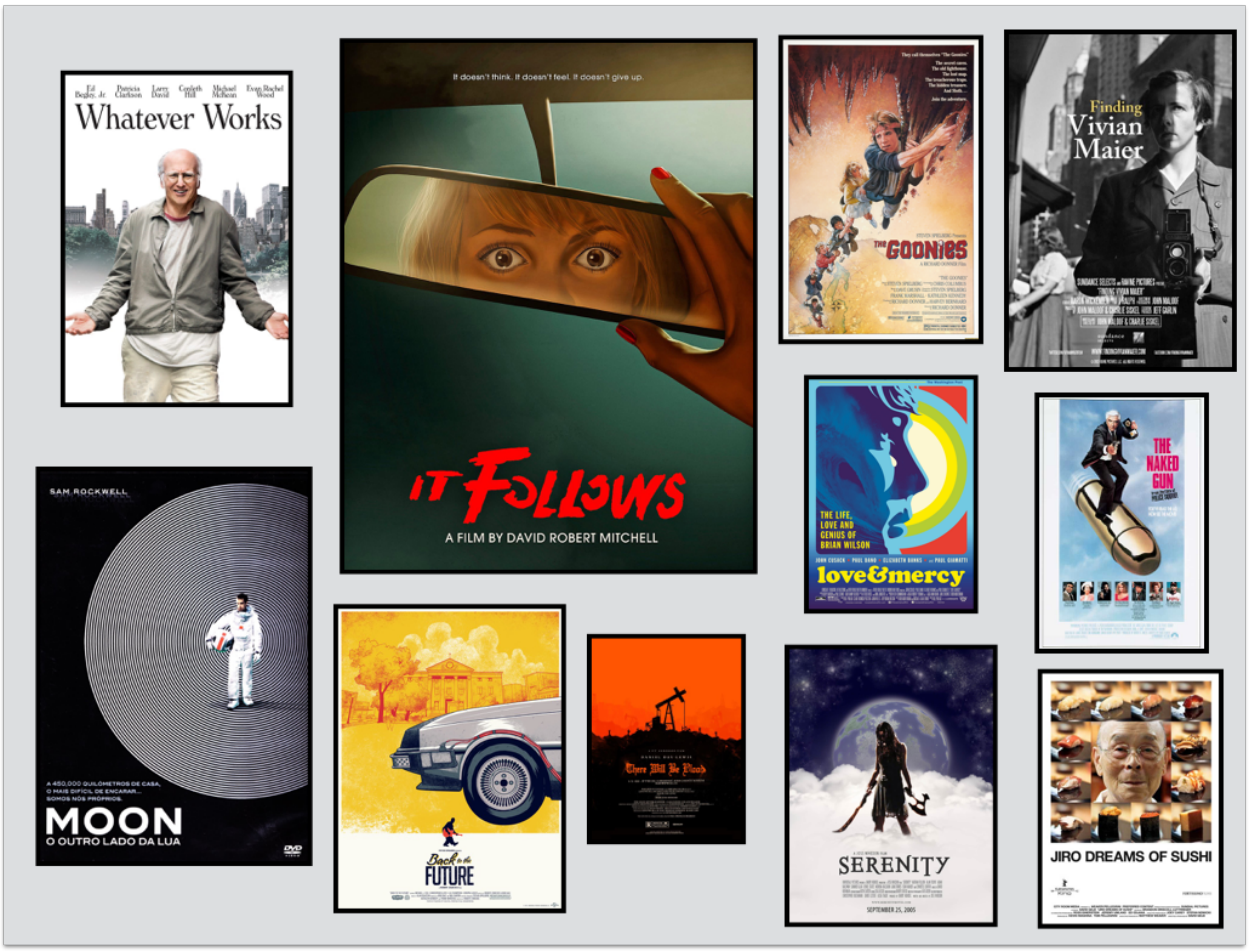
As you can see, we have some invalid values (999 and -2) in this vector. To remove them, we'll use logical indexing to change the invalid values (999 and -2) to NA. We'll create a logical vector indicating which values of `happy` are *invalid* using the `%in%` operation. Because we want to see which values are *invalid*, we'll add the `== FALSE` condition (If we don't, the index will tell us which values *are* valid).

```
# Which values of happy are NOT in the set 1:5?
invalid <- (happy %in% 1:5) == FALSE
invalid
##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

Now that we have a logical index `invalid` telling us which values are invalid (that is, not in the set 1 through 5), we'll index `happy` with `invalid`, and assign the invalid values as NA:

```
# Convert any invalid values in happy to NA
happy[invalid] <- NA
happy
##  [1]  1  4  2 NA  2  3 NA  3  2 NA
```

We can also recode all the invalid values of `happy` in one line as follows:




```
# Convert all values of happy that are NOT integers from 1 to 5 to NA
happy[(happy %in% 1:5) == FALSE] <- NA
```

As you can see, `happy` now has NAs for previously invalid values. Now we can take a `mean()` of the vector and see the mean of the valid responses.

```
# Include na.rm = TRUE to ignore NA values
mean(happy, na.rm = TRUE)
## [1] 2.428571
```

7.4 Test your R Might!: Movie data

Table 7.2 contains data about 10 of my favorite movies.

0. Create new data vectors for each column.
1. What is the name of the 10th movie in the list?
2. What are the genres of the first 4 movies?
3. Some joker put Spice World in the movie names – it should be "The Naked Gun" Please correct the name.
4. What were the names of the movies made before 1990?

Table 7.2: Some of my favorite movies

movie	year	boxoffice	genre	time	rating
Whatever Works	2009	35.0	Comedy	92	PG-13
It Follows	2015	15.0	Horror	97	R
Love and Mercy	2015	15.0	Drama	120	R
The Goonies	1985	62.0	Adventure	90	PG
Jiro Dreams of Sushi	2012	3.0	Documentary	81	G
There Will be Blood	2007	10.0	Drama	158	R
Moon	2009	321.0	Science Fiction	97	R
Spice World	1988	79.0	Comedy	-84	PG-13
Serenity	2005	39.0	Science Fiction	119	PG-13
Finding Vivian Maier	2014	1.5	Documentary	84	Unrated

5. How many movies were Dramas? What percent of the 10 movies were Dramas?
6. One of the values in the `time` vector is invalid. Convert any invalid values in this vector to NA. Then, calculate the mean movie time
7. What were the names of the Comedy movies? What were their boxoffice totals? (Two separate questions)
8. What were the names of the movies that made less than $50 Million dollars AND were Comedies?
9. What was the median boxoffice revenue of movies rated either G or PG?
10. What percent of the movies were rated R OR were comedies?

Chapter 8

Matrices and Dataframes

Chapter 9

Importing, saving and managing data

Chapter 10

Advanced dataframe manipulation

Chapter 11

Solutions

Chapter 12

Placeholder

Bibliography