

Section 3: R basics

Gov 50 Section

Week of September 21, 2026

Today

By the end of the hour you can:

1. Create a variable and say what type of value it holds.
2. Put several values in a vector and pull some of them back out.
3. Notice a missing value and work around it.
4. Pick rows and columns out of a table.
5. Call a function, know its arguments and defaults, and write a short one of your own.

The hour has five blocks, one for each objective, and every block works the same way. First your TF codes on the screen while you run the same lines on your laptop. That part is labeled **Watch**. Then you and a partner try a similar task on your own. That part is labeled **Try it**. When most pairs are done, we come back together and go over the answer. At the end of section you will knit your file to a PDF and upload it to Canvas. It is graded for completion only.

You will use two shortcuts all hour. To run the single line your cursor is on, press **Cmd + Enter** on a Mac or **Ctrl + Enter** on Windows. To run a whole chunk at once, click the green arrow at the top right of the chunk.

Block 0: Warm up

The gray box below is a code chunk. Put your cursor on the line inside it and run it. The answer appears right under the chunk.

```
2 + 2
```

Try it

Now knit the whole document with the **Knit** button at the top of the editor, or with **Cmd + Shift + K** on a Mac or **Ctrl + Shift + K** on Windows. Knitting runs every chunk in order and turns the file into a PDF, the same way you did in Lab 1. If a PDF opens, your setup works and you are ready for the hour. If it does not, raise a hand now rather than later.

Block 1: Objects and types

Watch

Almost everything in R starts with storing a value under a name, so that you can use the name later instead of retyping the number.

```
# Japan's life expectancy in 1952 and in 2007 (years)
japan_1952 <- 63.0
japan_2007 <- 82.6

japan_2007 - japan_1952           # prints the answer, does not store it

japan_change <- japan_2007 - japan_1952  # stores it under a name
japan_change
```

The arrow `<-` means “take the value on the right and store it under the name on the left.” Once a value is stored, its name appears in the **Environment** pane at the top right of RStudio, and you can use the name anywhere you would have typed the number. A name that holds a value is called a **variable**. Creating one is called **assignment**, and `<-` is the assignment operator. So `japan_1952` is a variable, and it holds the number 63.0. Anything after a `#` on a line is a comment. R skips comments entirely, so they are there for the people reading the code.

Every value in R has a type, which R calls its class. The class tells R what it is allowed to do with the value. You can do arithmetic on a number but not on a piece of text, for example. You need three classes today.

Type	Looks like	<code>class()</code> says
number	63.0	"numeric"
text, in quotes	"Japan"	"character"
yes/no	TRUE, 63 > 40	"logical"

```
class(japan_1952)
class("Japan")
class(japan_2007 > japan_1952)  # a yes/no question has a yes/no type
```

The class matters in practice. If you put a number inside quotes, R treats it as text, and text cannot be added to anything. The next chunk shows the error you get when you try.

```
"7" + 1
```

Try it

Now do the same thing for India, whose life expectancy was 37.4 in 1952 and 64.7 in 2007. The numbered comments in the chunk tell you what to do at each step. Type your code under each comment.

```
# 1. Store the two numbers under two names (like india_1952)
# 2. Compute the change, store it, print it
# 3. Before running: what will class() say for "2007" and for 2007? Then run it.
```

Block 2: Vectors

Watch

A vector is a set of values stored together under one name, in a fixed order. You build one with the function `c()`, which is short for combine. You list the values inside the parentheses, separated by commas.

```
# 2007 life expectancy: Japan, India, China, Vietnam, Afghanistan
asia_2007 <- c(82.6, 64.7, 73.0, 74.2, 43.8)
asia_2007

length(asia_2007)           # how many values
asia_2007[1]                # the first one
mean(asia_2007)
```

When you subtract one vector from another, R lines the two up and subtracts the first value from the first value, the second from the second, and so on down the line. One line of code does the arithmetic for all five countries. A comparison such as `> 70` works the same way. R checks each value in turn and gives back a TRUE or a FALSE for every one of them. Those answers are useful. Put them inside square brackets and R keeps only the values where the answer was TRUE. Add them up with `sum()` and R counts how many were TRUE, because TRUE counts as 1 and FALSE counts as 0.

```
asia_1952 <- c(63.0, 37.4, 44.0, 40.4, 28.8)
asia_2007 - asia_1952      # change for each country, in one line

asia_2007 > 70             # one TRUE or FALSE per country
asia_2007[asia_2007 > 70] # keep only the values where that is TRUE
sum(asia_2007 > 70)       # how many TRUES (TRUE counts as 1)
```

Try it

Here are the 2007 life expectancies for five African countries: Egypt 71.3, Kenya 54.1, Nigeria 46.9, Botswana 50.7, and Ethiopia 52.9. Repeat the moves from the Watch chunk on these numbers.

```
# 1. Store the five numbers as one vector called africa_2007
# 2. Its mean
# 3. Keep only the values above 60
# 4. How many are above 60?
```

Block 3: Missing values

Watch

Real datasets have holes in them. A survey respondent skips a question, or a country does not report a number for some year. R marks a missing value with NA, short for not available. An NA is contagious. If any value in a calculation is NA, the result is NA too, because R cannot know what the answer would have been.

```
some_1952 <- c(63.0, NA, 44.0)
mean(some_1952)           # NA, because one value is unknown

is.na(some_1952)          # where are the holes?
sum(is.na(some_1952))     # how many?
mean(some_1952, na.rm = TRUE) # mean of the values we do have
```

Adding `na.rm = TRUE` inside the parentheses tells the function to remove the missing values before it does anything else. Most of the summary functions you will use, including `mean()`, `sum()`, and `max()`, accept this option.

Try it

Here are the 1952 values for the same five African countries, but pretend that Kenya's number was never recorded. Run this chunk to store them, then answer the two questions in the chunk below it.

```
africa_1952 <- c(41.9, NA, 36.3, 47.6, 34.1)
```

```
# 1. How many values are missing?  
# 2. The mean of the values you do have
```

Block 4: Data frames

Watch

A data frame is R's name for a table of data. Each row is one unit, here a country, and each column is one variable measured for every unit. Every dataset you load this semester, including the replication data, will arrive as a data frame. The chunk below builds a small one by hand from four vectors. You do not need to type it. Just run it as it is.

```
countries <- data.frame(  
  country = c("Japan", "India", "China", "Vietnam", "Afghanistan",  
              "Egypt", "Kenya", "Nigeria", "Botswana", "Ethiopia"),  
  continent = c("Asia", "Asia", "Asia", "Asia", "Asia",  
               "Africa", "Africa", "Africa", "Africa", "Africa"),  
  life_1952 = c(63.0, 37.4, 44.0, 40.4, 28.8, 41.9, 42.3, 36.3, 47.6, 34.1),  
  life_2007 = c(82.6, 64.7, 73.0, 74.2, 43.8, 71.3, 54.1, 46.9, 50.7, 52.9)  
)  
countries  
  
nrow(countries)      # how many rows  
str(countries)       # each column and its class: num is numeric, chr is character  
summary(countries)   # a few summary numbers for each column
```

The dollar sign pulls a single column out of a table by name. What comes out is an ordinary vector, so everything you learned in Block 2 works on it.

```
countries$life_2007  
mean(countries$life_2007)
```

You can add a column the same way you store any value. Put a new column name after the dollar sign and assign a vector to it. Here the new column is the change in life expectancy for each country.

```
countries$change <- countries$life_2007 - countries$life_1952  
countries
```

To look at only some of the rows, use `subset()`. You give it the table and a test. It checks the test on every row and keeps the rows where the test comes out TRUE.

```
subset(countries, continent == "Asia")
```

The next line computes the mean change for the Asian countries alone. It is the hardest line of the hour, so read it from the inside out. The test `countries$continent == "Asia"` gives a TRUE or FALSE for each of the ten rows. The square brackets use those answers to keep only the `change` values from the TRUE rows, which are the five Asian countries. Then `mean()` averages those five numbers.

```
mean(countries$change[countries$continent == "Asia"])
```

`subset()` is the readable way to pick rows, because you refer to columns by name. There is a second way, with square brackets, and you will see it often in other people's code. Recall from Block 2 that `asia_2007[1]` pulled the first value out of a vector. The same idea works on a data frame, except that a table has two dimensions, so you give a row and a column. For instance:

```
countries[1, 1]
```

This gives "Japan", the value in the first row and first column of `countries`. The row always comes first and the column second, so `countries[1,2]` is the first row and second column, not the other way around.

To take a whole row or a whole column, leave the other side of the comma blank.

```
countries[, 2]      # the whole second column, continent
countries[1, ]      # the whole first row, Japan
countries[, 1:2]    # the first two columns, for every row
```

Try it

```
# 1. Show only the African rows, with subset()
# 2. The mean change for Africa
# 3. How many of the ten countries were above 70 in 2007?
# 4. With bracket notation, find the mean life expectancy across all ten
#    countries in 2007. That is the fourth column.
```

Block 5: Functions

Watch

You have been calling functions all hour. `mean()`, `sum()`, and `c()` are all functions. A function is a named recipe that takes some inputs and gives back a result. Here is one call with its parts labeled.

```
round(3.14159, digits = 2)
|      |      |      |
|      |      |      +--- argument: the value handed to digits
|      |      +----- parameter: the name of the second input
|      +----- argument: the value handed to x, by position
+----- the function
```

A **parameter** is a named slot in the function's definition. `round()` has two that matter here, `x` and `digits`. An **argument** is the value you put into a slot when you call the function. R matches arguments to parameters in two ways. A bare value is matched by position. The first bare value goes to the first parameter, the second to the second, and so on. A value written as `name = value` is matched by name, and then its position does not matter. Positional is shorter. Named is safer once a function has more than one or two parameters, because you do not have to remember the order. A common habit is to give the first argument by position and the rest by name, as in `mean(x, na.rm = TRUE)`.

```
round(3.14159, 2)           # both by position
round(3.14159, digits = 2)  # the second one by name
round(digits = 2, x = 3.14159) # both by name, so the order can change
```

A **default** is the value a parameter falls back on when you do not supply an argument for it. `digits` has a default of 0, so `round(3.14159)` works and rounds to a whole number. `x` has no default. If you give `mean()` nothing, R has nothing to average and stops with an error.

```
round(3.14159)
```

```
mean()
```

`Sys.Date()` is a different case again. It has no parameters at all, because it needs no input to do its job, which is to report today's date. You still type the parentheses. In R, a name followed by parentheses means "run this function." The name alone only refers to the function and does not run it.

```
Sys.Date()
```

Every function has a help page. Type `?round` in the Console and one opens in the bottom right pane. Under the heading **Usage**, find the line `round(x, digits = 0, ...)`. That line lists the parameters in order, and the `= 0` after `digits` is how the help page tells you the default. You can ignore the `...` for now.

You can also write your own function. The pattern is `name <- function(parameters) { body }`. The body, inside the curly brackets, is the recipe, and whatever the last line of the body produces is what the function gives back. Here is a function that converts a temperature from Fahrenheit to Celsius.

```
f_to_c <- function(fahrenheit) {
  (fahrenheit - 32) * 5 / 9
}

f_to_c(212)           # boiling
f_to_c(98.6)          # body temperature
f_to_c(c(32, 50, 100)) # works on a whole vector at once
```

The last call handed the function a vector of three temperatures and got three answers back. That works because the body of the function is ordinary arithmetic, and arithmetic in R works on a whole vector at once, as you saw in Block 2. Anything you can do to one number, this function can do to a whole column.

Try it

Write the function that goes the other way, from Celsius to Fahrenheit. The recipe is to multiply by 9/5 and then add 32. Start from the skeleton below and fill in the blank line.

```
c_to_f <- function(celsius) {
  ---
}
```

```
# 1. Fill in the skeleton and run the chunk so R learns the recipe
# 2. Call it for 100 and for 37
# 3. Call it on the vector c(0, 20, 30)
```

Try it: arguments and defaults

```
# 1. Run round(2.71828) and round(2.71828, 3).  
#    Which parameter got its default in the first call?  
# 2. Give c_to_f a default of 100 by writing function(celsius = 100),  
#    then call c_to_f() with nothing inside the parentheses  
# 3. Call it by name: c_to_f(celsius = 37)
```

What you learned today

Objective	The move	Looks like
1. Create a variable, know its type	<code><-</code> (assignment), <code>class()</code>	<code>japan_1952 <- 63.0</code>
2. Several values under one name	<code>c()</code> , <code>[]</code> , <code>mean()</code>	<code>asia_2007[asia_2007 > 70]</code>
3. Missing values	<code>is.na()</code> , <code>na.rm = TRUE</code>	<code>mean(x, na.rm = TRUE)</code>
4. Rows and columns of a table	<code>\$</code> , <code>subset()</code>	<code>subset(countries, continent == "Asia")</code>
5. Call and write functions	<code>name(arg = value)</code> , <code>?name</code> , <code>function()</code>	<code>round(3.14, digits = 1)</code> , <code>f_to_c(212)</code>

Type	Looks like	<code>class()</code> says
number	63.0	"numeric"
text	"Japan"	"character"
yes/no	TRUE, 63 > 40	"logical"
missing	NA	"logical"
vector	<code>c(82.6, 64.7)</code>	the type of its values
table	<code>countries</code>	"data.frame"

Before you leave

Knit this file to a PDF one more time and upload the PDF to Canvas. It is graded for completion, so a file with honest attempts at each Try it counts in full.

Sam's Intro to Base R handout covers the same material at more length, and goes on to loops, `apply()`, and reading data files.