R Basics Exercises

Contents

Introduction

This document contains brief descriptions of core R functionality. Examples are provided as code blocks. Code block results are printed below the code block, prefixed by `##`.

```
"This is a code block. Code blocks use different fonts and have light gray backgrounds.
They contain examples of valid R code."
```

```
## The results of running a code block are shown underneath each block.
```

As you read through this document, follow along by typing the code block examples into an R console and running them yourself. Check that you get the same output as displayed in this document.

You will also find a series of exercises at the end of the document.

Basic R syntax

Math operations

Basic arithmetic

```r
(17 * 0.35) ^ 1/3
```

```
## [1] 1.983333
```

Logarithms and exponentiation use the `log` and `exp` functions. The default base is `e`, with a value of approximately 2.71828.

```r
log(2.71828)
```

```
## [1] 0.9999993
```

```r
exp(1)
```

```
## [1] 2.718282
```

Specify the base of your logarithm with the `base` argument. Common logarithm bases have their own functions.

```r
log(10, base=10)
```

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

```r
log10(10)
```

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

```r
log2(10)
```

```
## [1] 3.321928
```

Logical operators

Logical operators are used for value and/or variable comparisons.

- Equivalent: `==`
- Not equivalent: `!=`
- Less than: `<`
- Greater than: `>`
- Or: `|`
- And: `&`

```r
g = 12
h = 13
g == h
```

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

Variables

There are a few important properties of R variables:

1. Variables are given values by the process of variable assignment
2. Variables have different types
3. Variables can be reused
4. Variables are mutable; their values can be modified or changed

Variable assignment

Assignment by convention uses the `<-` operator, with the variable name on the left and its assigned value on the right.

```r
a <- 100
print(a)
```

```
## [1] 100
```

Variable types

Numeric variables are integers and decimal values (also known as type "double", for double-precision floating point value). *For most of your work, this distinction is irrelevant,* but sometimes it does impact calculations so it's good to be aware this distinction exists.

The `typeof` function can be used to report the type of a variable. Recall variable `a` assigned above.

```r
typeof(a)
```

```
## [1] "double"
```

Even though `a` was assigned the value of "100" without a decimal, it is by default treated as a variable that can have decimals. If you need an integer data type, you can ask for one explicitly:

```r
b <- as.integer(100)
typeof(b)
```

```
## [1] "integer"
```

Character strings are text variables. They must be enclosed in either single or double quotes.

```r
d <- 'Hello world!'
typeof(d)
```

```
## [1] "character"
```

Logical variables are `TRUE` or `FALSE`.

```r
e <- TRUE
typeof(e)
```

```
## [1] "logical"
```

```r
f <- FALSE
typeof(f)
```

```
## [1] "logical"
```

Missing values

Variables of each data type (numeric, character and logical) can also take the value of `NA`: "not available". - `NA` is not the same as 0 - `NA` is not the same as " " - `NA` is not the same as FALSE

Any operations (calculations or comparisons) that involve `NA` may or may not produce `NA`.

Variable reassignment and modification

You can also assign variables the value of other variables. The code block below assigns variable `f` the value of `a` converted to an integer. This assignment does not change the value of `a`.

```r
g <- as.integer(a)
print(typeof(g))
```

```
## [1] "integer"
```

```r
print(typeof(a))
```

```
## [1] "double"
```

If we change the value of `a`, what happens to the value of `f`?

```r
a <- 'Now a is a character string'
print(a)
```

```
## [1] "Now a is a character string"
```

```r
print(g)
```

```
## [1] 100
```

The value of `f` remains unchanged because in R, variable assignment copies values rather than referencing them. For more details, see examples here.

Scalar vs vector variables

So far we have considered only **scalar** variables – variables with a single value. Variables can also hold multiple variables in vectors.

We often use functions to create vector variables. The `c()` function combines values into vectors:

```
my_vector <- c(1, 1, 2, 3, 5, 8)
print(my_vector)
```

```
## [1] 1 1 2 3 5 8
```

The `seq()` function makes a sequence. Its arguments are start, stop, and step.

```
my_sequence <- seq(30, 20, -1)
print(my_sequence)
```

```
##  [1] 30 29 28 27 26 25 24 23 22 21 20
```

Getting help with functions

You've seen a few examples of functions so far: `print()`, `typeof()`, `c()` and `seq()`. There are many, many functions in R and its optional packages. For most, you can get access standardized documentations for functions with the `help()` function or `?` operator.

```
help("c")
```

```
?c
```

The help documentation has several sections of which these are typically the most informative.

1. "Description" is a basic description of the function.
2. "Usage" shows brief examples of the function call.
3. "Arguments" lists the arguments (the variables or data you pass to the function, upon which the function does work). This section will also explain the expected types for argument values and the argument default values (if any).
4. "Details" expands on the description a bit.
5. "Value" describes the value the function returns.
6. "Examples" shows usage examples in more detail.

Basic data structure

Vectors

We briefly discussed vectors above, in contrast to scalar variables. Vectors are ordered collections of data of the same type. Elements in vectors are usually accessed by index. In R, indexing starts with "1".

Recall the `my_vector` variable defined above. Use square brackets to reference elements in the list by their index number:

```
print(my_vector)
```

```
## [1] 1 1 2 3 5 8
```

```
print(my_vector[1])
```

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

```
print(my_vector[4])
```

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

We can perform mathematical operations on vectors. We can multiply each value in the vector by a scalar value:

```
my_vector*3
```

```
## [1]  3  3  6  9 15 24
```

We can also perform aggregations on the vector:

```
sum(my_vector)
```

```
## [1] 20
```

Matrices

Matrices have a `[row, column]` data structure; like vectors, all elements in a matrix must have the same data type. Matrices can be used for linear algebra computations.

```
A = matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3)
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Extract single elements from a matrix with their `[row, column]` positional indexes.

```
A[2, 2]
```

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

Slice out rows and columns by specifying only the row or column number:

```
print(A[2, ])
```

```
## [1] 2 5 8
```

```
print(A[ ,2])
```

```
## [1] 4 5 6
```

Perform matrix multiplication with the `%*%` operator:

```
x <- c(1, 2, 3)
A %*% x
```

```
##      [,1]
## [1,]   30
## [2,]   36
## [3,]   42
```

Lists

Lists are a special type of vector containing ordered key-value pairs and can contain arbitrary data types. Elements in lists can be accessed by key or by positional index.

```
phonebook <- list(name="Jenny", number="867-5309")
print(phonebook$name)
```

```
## [1] "Jenny"
```

```
print(phonebook[1])
```

```
## $name
## [1] "Jenny"
```

Dataframes

Data frames are `[row, column]` organized data objects. Rows contain data items (e.g. public health records) and columns contain values of different attributes (e.g. age, address). Values within a column should all have the same type.

R has a built-in `data.frame` type, and and the `tibble` package implements a version of the dataframe that has become very popular for data analysis. This document will use the `tbl_df` dataframe; for more information about the built-in `data.frame` see here. The `tibble` documentation outlines the differences between `tbl_df` and `data.frame`.