Introduction

Canhong Wen

2020-9-15

Agenda

- Data types
- Built-in functions and operators

Overall class summary: Functional programming

2 sorts of things (**objects**): **data** and **functions**

- **Data**: things like 7, "seven", 7.000, the matrix $\begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$
- **Functions**: things like $\log$, $+$ (two arguments), $<$ (two), $\mod$ (two), `mean` (one)

  A function is a machine which turns input objects (**arguments**) into an output object (**return value**), possibly with **side effects**, according to a definite rule

Data object

All data is represented in binary format, by **bits** (TRUE/FALSE, YES/NO, 1/0)

- **Booleans** Direct binary values: TRUE or FALSE in R
- **Integers**: whole numbers (positive, negative or zero), represented by a fixed-length block of bits
- **Characters** fixed-length blocks of bits, with special coding; **strings** = sequences of characters
- **Floating point numbers**: a fraction (with a finite number of bits) times an exponent, like 1.87×10^6, but in binary form
- **Missing or ill-defined values**: NA, NaN, etc.

Operators

- **Unary** `-` for arithmetic negation, `!` for Boolean
- **Binary** usual arithmetic operators, plus ones for modulo and integer division; take two numbers and give a number

```r
7+5
```

```
## [1] 12
```

```r
7-5
```

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

```r
7*5
```

```
## [1] 35
```

```r
7^5
```

```
## [1] 16807
```

```r
7/5
```

```
## [1] 1.4
```

```r
7 %% 5 # the modulo operator
```

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

```r
7 %/% 5 # indicates integer division
```

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

Operators

Comparisons are also binary operators; they take two objects, like numbers, and give a Boolean

```r
7 > 5
```

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

```r
7 < 5
```

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

```r
7 >= 7
```

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

```r
7 <= 5
```

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

```r
7 == 5
```

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

```r
7 != 5
```

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

Boolean operators

```r
(5 > 7) & (6*7 == 42)
```

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

```r
(5 > 7) | (6*7 == 42)
```

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

(will see special doubled forms, `&&` and `||`, later)

More types

`typeof()` function returns the type

`is.`*foo*`()` functions return Booleans for whether the argument is of type *foo*

`as.`*foo*`()` (tries to) "cast" its argument to type *foo* — to translate it sensibly into a *foo*-type value

```r
typeof(7)
```

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

```r
is.numeric(7)
```

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

```r
is.na(7)
```

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

```r
is.na(7/0)
```

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

```r
is.na(0/0)
```

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

Why is 7/0 not NA, but 0/0 is?

```r
is.character(7)
```

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

```r
is.character("7")
```

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

```r
is.character("seven")
```

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

```r
is.na("seven")
```

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

```r
as.character(5/6)
```

```
## [1] "0.833333333333333"
```

```r
as.numeric(as.character(5/6))
```

```
## [1] 0.8333333
```

```r
6*as.numeric(as.character(5/6))
```

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

```r
5/6 == as.numeric(as.character(5/6))
```

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

(why is that last FALSE?)

Floating point numbers

The R floating point data type is `double`.

Finite precision $\Rightarrow$ arithmetic on `doubles` $\neq$ arithmetic on $\mathbb{R}$.

```r
0.45 == 3*0.15
```

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

```r
0.45 - 3*0.15
```

```
## [1] 5.551115e-17
```

Often ignorable, but not always - Rounding errors tend to accumulate in long calculations - When results should be ≈ 0, errors can flip signs - Usually better to use `all.equal()` than exact comparison.

```r
all.equal(0.45, 3*0.15)
```

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

Data can have names

We can give names to data objects; these give us **variables**

A few varuabkes are built in:

```r
pi
```

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

Variables can be arguments to functions or operators, just like constants:

```r
pi*10
```

```
## [1] 31.41593
```

```r
cos(pi)
```

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

Most variables are created with the **assignment operator**, `<-` or `=`

```r
approx.pi <- 22/7
approx.pi
```

```
## [1] 3.142857
```

```
diameter.in.meters = 10
approx.pi * diameter.in.meters
```

```
## [1] 31.42857
```

The assignment operator also changes values:

```
circumference.in.meters <- approx.pi * diameter.in.meters
circumference.in.meters
```

```
## [1] 31.42857
```

```
circumference.in.meters <- 30
circumference.in.meters
```

```
## [1] 30
```

On the names of data

- Using names and variables makes code: easier to design, easier to debug, less prone to bugs, easier to improve, and easier for others to read
- Avoid "magic constants"; use named variables you will be graded on this!
- Named variables are a first step towards **abstraction**

The workspace

What names have you defined values for?

```
ls()
```

```
## [1] "approx.pi"               "circumference.in.meters"
## [3] "diameter.in.meters"
```

```
objects()
```

```
## [1] "approx.pi"               "circumference.in.meters"
## [3] "diameter.in.meters"
```

Get rid of variables:

```
rm("circumference.in.meters")
ls()
```

```
## [1] "approx.pi"          "diameter.in.meters"
```

The work directory

- Many scripts and data sets are provided, and many will be created by users. It is convenient to create a folder or directory with a short path name to store these files.
- Better to create a project to store all the files in `RStudio`.
- A user can get or set the current working directory by `getwd` and `setwd`.

```
getwd()
```

Using Scripts

R scripts are plain text files containing R code. Once code is saved in a script, all of it can be submitted via the source command, or part of it can be executed by copy and paste (to the console).

```r
source("data/example.R")
source("data/example.R", echo=TRUE)
```

```
##
## > sqrt(pi)
## [1] 1.772454
```

```r
source("data/example.R", print.eval=TRUE)
```

```
## [1] 1.772454
```