Introduction: Basics of Data

James M. Flegal

Agenda

- Built-in data types
- Built-in functions and operators
- First data structures: Vectors and arrays

Why good statisticians learn to program?

- *Independence*: Otherwise, you rely on someone else having given you exactly the right tool
- *Honesty*: Otherwise, you end up distorting your problem to match the tools you have
- *Clarity*: Making your method something a machine can do disciplines your thinking and makes it public; that's science

How this class will work

- No programming knowledge presumed
- Some statistics knowledge presumed
- General programming mixed with data-manipulation and statistical inference
- Class will be *very* cumulative
- Keep up with the readings and assignments!
- Assignments, office hours, class notes, grading policies, useful links on Canvas

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), $\bmod$ (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

Overall class summary: Functional programming

- Programming is writing functions to transform inputs into outputs
- Good programming ensures the transformation is done easily and correctly
- Machines are made out of machines; functions are made out of functions, like $f(a, b) = a^2 + b^2$
- Good programming takes the big transformation and breaks it down into smaller ones, and then break those down, until you come to tasks which the built-in functions can do

Data

- Different kinds of 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

The R Console

```r
7+5
```

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

```r
7-5
```

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

```r
7*5
```

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

```r
7^5
```

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

The R Console

```r
7/5
```

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

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

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

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

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

The R Console

- Basic interaction with R is by typing in the **console**, a.k.a. **terminal** or **command-line**
- You type in commands, R gives back answers (or errors)
- Menus and other graphical interfaces are extras built on top of the console

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
```

Operators

```r
7 <= 5
```

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

```r
7 == 5
```

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

```r
7 != 5
```

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

Boolean operators

- Basically "and" and "or":

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

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

```
(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

More types

```r
typeof(7)
```

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

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

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

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

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

More types

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

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

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

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

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

More types

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

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

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

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

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

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

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

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

More types

```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`?

Data can have names

- Can give names to data objects, gives us **variables** (a few are built in)

```
pi
```

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

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

```
pi*10
```

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

```
cos(pi)
```

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

Data can have names

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

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

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

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

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

Data can have names

- Assignment operator also changes values

```r
circumference.in.cubits <- approx.pi*diameter.in.cubits
circumference.in.cubits
```

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

```r
circumference.in.cubits <- 30
circumference.in.cubits
```

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

Data can have names

- 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 (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.cubits"
## [3] "diameter.in.cubits"
```

```
objects()
```

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

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

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

Vectors

- First data structure
- Group related data values into one object, a **data structure**
- A **vector** is a sequence of values, all of the same type

```
x <- c(7, 8, 10, 45)
x
```

```
## [1]  7  8 10 45
```

```
is.vector(x)
```

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

Vectors

- `c()` function returns a vector containing all its arguments in order
- `x[1]` is the first element, `x[4]` is the 4th element
- `x[-4]` is a vector containing all but the fourth element

```
x
```

```
## [1]  7  8 10 45
```

```
x[1]
```

```
## [1] 7
```

```
x[-4]
```

```
## [1]  7  8 10
```

Vectors

- `vector(length=6)` returns an empty vector of length 6; helpful for filling things up later

```
weekly.hours <- vector(length=5)
weekly.hours[5] <- 8
```

Vector arithmetic

- Operators apply to vectors "pairwise" or "elementwise"

```
y <- c(-7, -8, -10, -45)
x+y
```

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

```
x*y
```

```
## [1]   -49   -64  -100 -2025
```

Recycling

- **Recycling** repeat elements in shorter vector when combined with longer

```
x + c(-7,-8)
```

```
## [1]  0  0  3 37
```

```
x^c(1,0,-1,0.5)
```

```
## [1] 7.000000 1.000000 0.100000 6.708204
```

- Single numbers are vectors of length 1 for purposes of recycling:

```
2*x
```

```
## [1] 14 16 20 90
```

Comparisons

- Can also do pairwise comparisons (returns Boolean vector)

```
x > 9
```

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

- Boolean operators work elementwise

```
(x > 9) & (x < 20)
```

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

Comparisons

- To compare whole vectors, best to use `identical()` or `all.equal()`

```
x == -y
```

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

```
identical(x,-y)
```

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

```
identical(c(0.5-0.3,0.3-0.1),c(0.3-0.1,0.5-0.3))
```

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

```
all.equal(c(0.5-0.3,0.3-0.1),c(0.3-0.1,0.5-0.3))
```

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

Functions on vectors

Lots of functions take vectors as arguments:

- `mean()`, `median()`, `sd()`, `var()`, `max()`, `min()`, `length()`, `sum()`: return single numbers
- `sort()` returns a new vector
- `hist()` takes a vector of numbers and produces a histogram, a highly structured object, with the side-effect of making a plot
- Similarly `ecdf()` produces a cumulative-density-function object
- `summary()` gives a five-number summary of numerical vectors
- `any()` and `all()` are useful on Boolean vectors

Addressing vectors

- Vector of indices

```
x[c(2,4)]
```

```
## [1]  8 45
```

- Vector of negative indices

```
x[c(-1,-3)]
```

```
## [1]  8 45
```

Addressing vectors

- Boolean vector

```
x[x>9]
```

```
## [1] 10 45
```

```
y[x>9]
```

```
## [1] -10 -45
```

- `which()` turns a Boolean vector in vector of TRUE indices

```
places <- which(x > 9)
places
```

```
## [1] 3 4
```

```
y[places]
```

```
## [1] -10 -45
```

Named components

- Can give names to elements or components of vectors

```
names(x) <- c("v1","v2","v3","fred")
names(x)
```

```
## [1] "v1"   "v2"   "v3"   "fred"
```

```
x[c("fred","v1")]
```

```
## fred   v1
##   45    7
```

- Note the labels in what R prints; not actually part of the value

Named components

- `names(x)` is just another vector (of characters)

```
names(y) <- names(x)
sort(names(x))
```

```
## [1] "fred" "v1"   "v2"   "v3"
```

```
which(names(x)=="fred")
```

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

Big Idea

- We write programs by composing functions to manipulate data
- The basic data types let us represent Booleans, numbers, and characters
- Data structure let us group related values together
- Vectors let us group values of the same type
- Use variables rather a profusion of magic constants
- Name components of structures to make data more meaningful

Peculiarities of floating-point numbers

- The more bits in the fraction part, the more precision
- The R floating-point data type is a `double`, a.k.a. `numeric`
  - back when memory was expensive, the now-standard number of bits was twice the default
- Finite precision $\Rightarrow$ arithmetic on `doubles` $\neq$ arithmetic on $\mathbb{R}$.

Peculiarities of floating-point numbers

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

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

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

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

Peculiarities of floating-point numbers

- 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
(0.5 - 0.3) == (0.3 - 0.1)
```

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

```r
all.equal(0.5-0.3, 0.3-0.1)
```

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

Peculiarities of Integers

- Typing a whole number in the terminal doesn't make an integer; it makes a double, whose fractional part is 0

```
is.integer(7)
```

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

- Looks like an integer

```
as.integer(7)
```

```
## [1] 7
```

- To test for being a whole number, use `round()`:

```
round(7) == 7
```

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