Functions, and Functional Programming

Kylie A. Bemis

Northeastern University
Khoury College of Computer Sciences

Northeastern University

Goals for this session

- Review of R basics
- Functions
- Functional programming

R BASICS

A brief history of R

- S created by John Chambers at Bell Labs in 1976
- "Turn ideas into software quickly and faithfully"
- R created in 1993 as FOSS implementation of S
- Influenced by S, Scheme, and XLispStat

The R Language

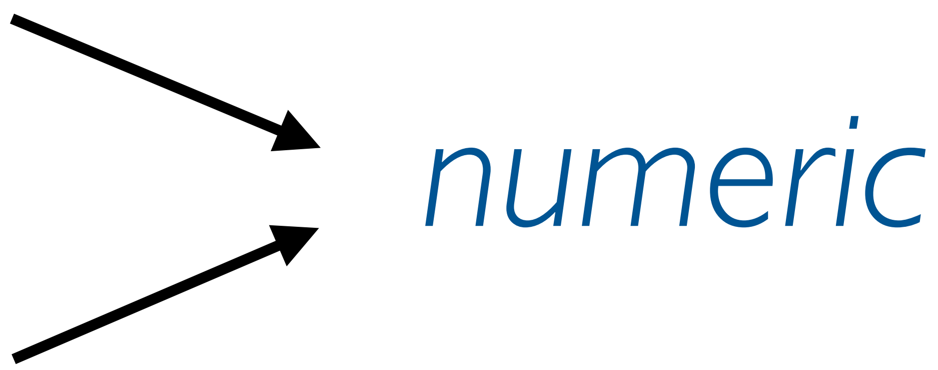
- Language and environment for statistical computing
- Written in C, FORTRAN, and R
- Flexible and extensible
- Over 10,000 user-contributed packages

The R Language (2)

- Interpreted
- Functional
- Dynamic typing
- Lexical scoping
- Object-oriented

Atomic data types in R

- **character**
- **double**
- **integer**
- **logical**
- raw
- complex



numeric

Non-atomic data types in R

- list
- array
- matrix
- etc.

Operations in R

- Everything is a vector
- Standard arithmetic operations are vectorized
- Linear algebra powered by BLAS/LAPACK
- Functional programming (more on this later)

FUNCTIONS

Functions in R

To understand computations in R, two slogans are helpful:

- *Everything that exists is an object*
- *Everything that happens is a function call*

— *John Chambers, creator of S*

"Everything that exists is an object"

- R code can be represented as an object
- Expressions can be manipulated
- Useful for modifying the language

"Everything that happens is a function call"

- All basic operators are functions
- *Everything* that happens is a function call
- Yes, even assignment and *parentheses*

Functions as first-class citizens

- Functions are objects
- Functions can be assigned to variables
- Functions can be manipulated
- Functions have an *environment*

Components of a function

- **Formals** (named parameters)
- **Body** (code to be evaluated)
- **Environment** (where to find variables)

Why write a function?

- Avoid duplication
    - No copy-pasting!
- Easily re-use a common workflow
- Share functionality with others

Which is better?

```
df$age <- ifelse(is.na(df$age), median(df$age, na.rm=TRUE), df$age)
df$time <- ifelse(is.na(df$time), median(df$sex, na.rm=TRUE), df$time)
df$year <- ifelse(is.na(df$year), median(df$year, na.rm=TRUE), df$year)
df$rate <- ifelse(is.na(df$rate), median(df$trt, na.rm=TRUE), df$rate)
```

versus

```
df$age <- impute_NA(df$age)
df$time <- impute_NA(df$time)
df$year <- impute_NA(df$year)
df$rate <- impute_NA(df$rate)
```

Exercise

- Implement the `impute_NA()` function
- Replace NAs of a vector with median

Writing the function

```
impute_NA <- function(x)
{
    ifelse(is.na(x), median(x, na.rm=TRUE), x)
}
```

What makes a good function

- Obviously correct
- Specific and simple utility
- Concise but readable
- Intuitive name
- Speed (sometimes)

FUNCTIONAL PROGRAMMING

Functional programming

- R programming emphasizes functions
- Functions map input to output
- Does not rely on *side effects*
  - Functions are used solely for the *result* they return
- Objects are treated as **immutable**
  - Functions do NOT (and cannot) *modify* their input

Functional programming (2)

- Easier to ***reason*** about inputs/outputs
- Makes testing and debugging simpler
- **Parallelization** is more straightforward

Functionals

- "*Higher-order functions*"
- Take one or more functions as input, ***or***
- Return a function as a result

Using functionals

- The "*apply*" family of functions
- Apply a function over elements of a list
- Typically preferred to *for* loops

Apply functions

```
x <- list(1:3, 4:6, 7:9)

vapply(x, sum, numeric(1))
```

BREAK