L06 Functions I

EPID 799B

Fall 2016

Xiaojuan Li

Outline

- Functions
- User-defined functions
- Simple loops

Announcements

- Homework 1 assigned and due Monday 9/19

Functions in R

Recall a function in mathematics

$$z = f(x,y) = x^2+y^2$$

$$f(3,4) = 3^2+4^2 = 25$$

In R, define a function by an assignment of the form

name <- function(*arg1*, *arg2*, …) {

Expression

}

Call the function by the form

name(*expr1*, *expr2*, …)

Pre-specified functions

- **sum(7,2,2)**

[1] 11

- **rnorm(n=10, mean=0, sd=1)**

[1] 2.3230962 1.0689816 -0.5875304 -1.4647930 -0.3629024 1.7436890 -0.4847395 0.4803791 0.2758953 1.2291493

- For help
  - **help(rnorm)**
  - **?rnorm**
  - **??rnorm**
  - **example(rnorm)**

Pre-specified functions – want to see the code?

- To see the whole code

sample # Type the function name without ()

```
function (x, size, replace = FALSE, prob = NULL)
{
    if (length(x) == 1L && is.numeric(x) && x >= 1) {
        if (missing(size))
            size <- x
        sample.int(x, size, replace, prob)
    }
    else {
        if (missing(size))
            size <- length(x)
        x[sample.int(length(x), size, replace, prob)]
    }
}
<bytecode: 0x00000000118c8588>
<environment: namespace:base>
```

- To see the syntax of the arguments

args(sample)

```
function (x, size, replace = FALSE, prob = NULL)
NULL
```

Calling functions

- Some arguments have default values; no need to specify them unless you want to override the default values
- The order of the arguments matter if you are using positional matching
- Always specify arguments when you use them out of their default order
- You can specify arguments by partial name with a sufficient number of letters so that R can uniquely identify it from the other arguments
- Argument values can be any valid R expression that returns an appropriate value

*Adapted from *Population Health Data Science with R* by Tomás J. Aragón

Using pre-specified functions

Dealing with missing values

- **mean(births$female)** # returns NA if any of the input values are NA
- **mean(births$female, na.rm=TRUE)** # excludes NA from the calculation
- **length(births$female)** # counts NAs just like any other value
- **sum(!is.na(births$female))** # is.na() returns a logical vector, it has a TRUE for
  # each NA. It is inverted by !, and the sum() adds
  # up the number of TRUEs then returns a count
  # of non-NAs

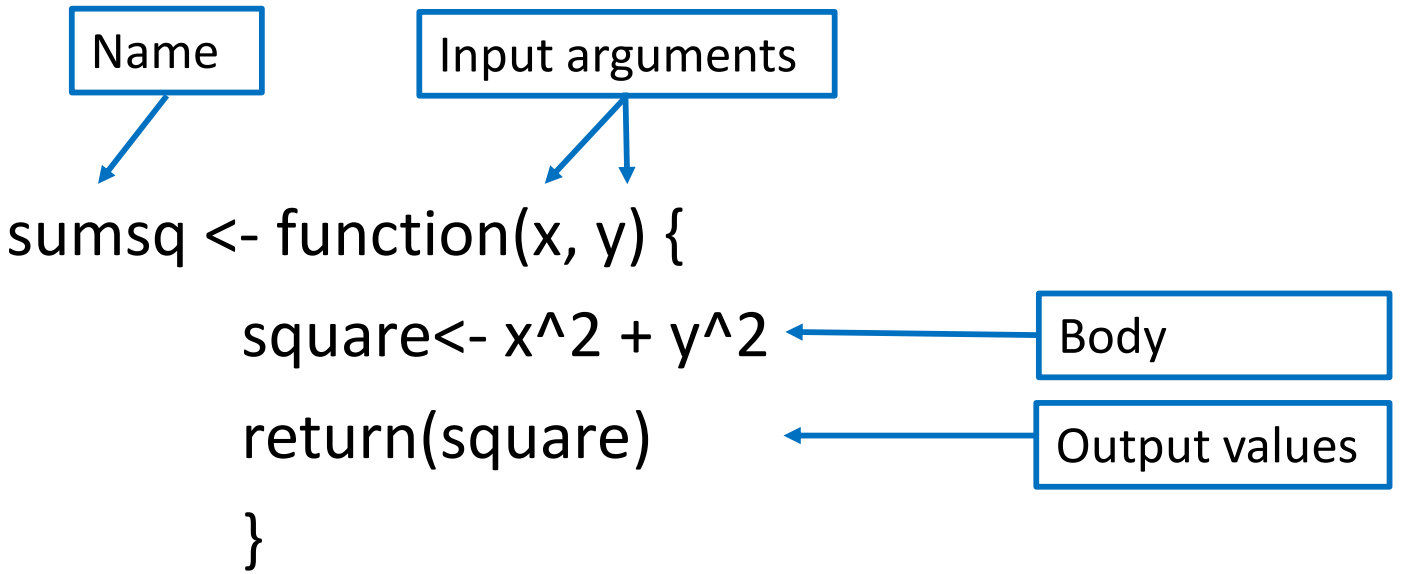
User-defined functions

- Idea – avoid repetitive coding (errors will creep in)

- How?
  - Extract common core
  - Wrap it in a function
  - Make it reusable

- Similar to writing a macro in SAS

Write your own function



Write your own function

```
sumsq <- function(x,y)
        { square<- x^2 + y^2
          return(square)
         }

sumsq(3,4)

sumsq(x=3,y=4)
```

```
# define your function
# name <- function(arg1, arg2, …)
#         {expression}

 # call the function
 # name(expr1, expr2, …)

 # equivalent way to call
```

Another example

```
twosamt <- function(x, y)
        {  n1 <- length(x)
           n2 <- length(y)
           m1 <- mean(x)
           m2 <- mean(y)
           s1 <- var(x)
           s2 <- var(y)
           s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
           tst <- (m1 - m2)/sqrt(s*(1/n1 + 1/n2))
           tst
        }
tstat <- twosamt(births$var4, births$var5)
tstat
```

*Adapted from *An Introduction to R* by WN Venables, DM Smith and the R Core Team

for() loops

- For repetitive executions

```
for (name in expression_1)
  {
    expression_2
  }
```

- Everything inside the curly brackets {...} is done N times
- for (i in 1:N) # sets up a vector (i) of length N
- for(k in 3:20) # doesn't have to count from 1

for() loop examples

```
for(i in c(1, 3, 6, 9))
      {
        z <- i*2
      }
z
```

```
[1] 18
```

```
for(i in c(1, 3, 6, 9))
      {
        z <- i*2
        print(z)
      }
```

```
[1] 2
[1] 6
[1] 12
[1] 18
```