Writing Functions

Canhong Wen

Agenda

- Defining functions: Tying related commands into bundles
- Interfaces: Controlling what the function can see and do
  - Inputs
  - Environment
  - Outputs and side effects

Why Functions?

- Data structures tie related values into one object
- Functions tie related commands into one object
- In both cases: easier to understand, easier to work with, easier to build into larger things

A Simple Example: cubic function

```
cube <- function(x) x ^ 3
cube
```

```
## function(x) x ^ 3
```

```
cube(3)
```

```
## [1] 27
```

```
cube(1:10)
```

```
##  [1]    1    8   27   64  125  216  343  512  729 1000
```

```
cube(matrix(1:8, 2, 4))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   27  125  343
## [2,]    8   64  216  512
```

```
matrix(cube(1:8), 2, 4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   27  125  343
## [2,]    8   64  216  512
```

```
# cube(array(1:24, c(2, 3, 4))) # cube each element in an array
mode(cube)
```

```
## [1] "function"
```

Reminder: "Robust" loss function

$$\psi(x) = \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2|x| - 1 & \text{if } |x| > 1 \end{cases}$$

```
if (x^2 < 1) {
  x^2
} else if (x >= 1) {
  2*x-1
} else {
 -2*x-1
}
```

Creating your own function

Call `function()` to create your own function. Document your function with comments

```
# "Robust" loss function, for outlier-resistant regression
# Inputs: vector of numbers (x)
# Outputs: vector with x^2 for small entries, 2|x|-1 for large ones
psi.1 <- function(x) {
  psi <- ifelse(x^2 > 1, 2*abs(x)-1, x^2)
  return(psi)
}
```

Using your created function

Our functions get used just like the built-in ones:

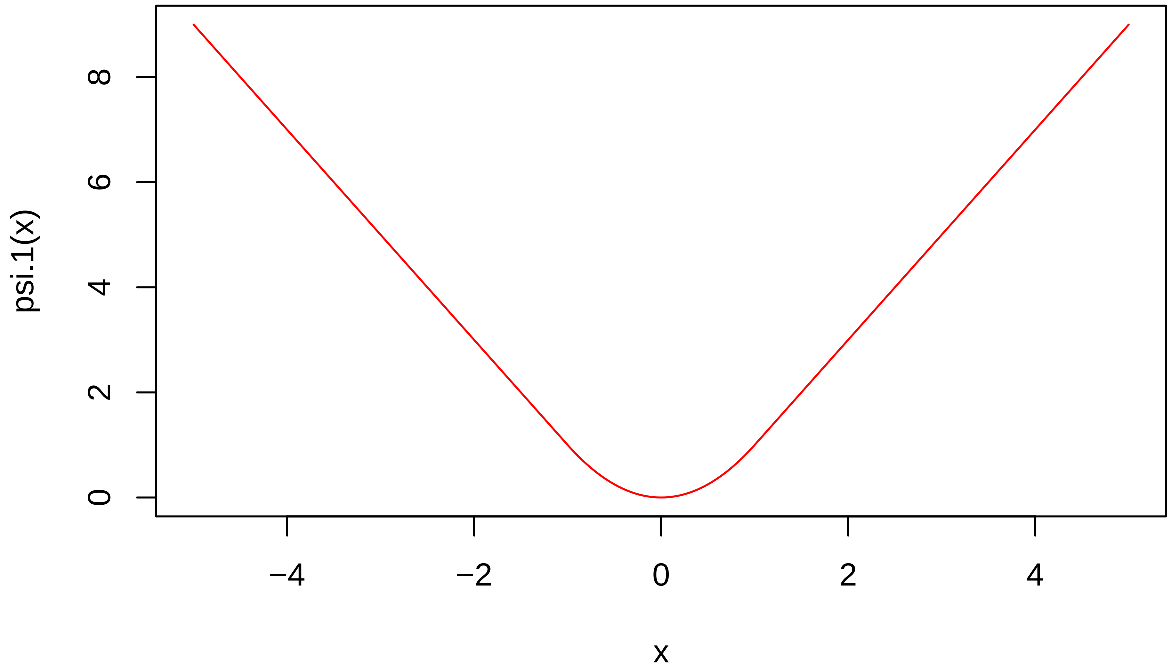
```
z <- c(-0.5,-5,0.9,9)
psi.1(z)
```

```
## [1]  0.25  9.00  0.81 17.00
```

```
psi.1
```

```
## function(x) {
##   psi <- ifelse(x^2 > 1, 2*abs(x)-1, x^2)
##   return(psi)
## }
```

```
x <- seq(from = -5, to = 5, by = 0.01)
plot(x, psi.1(x), type="l", col="red")
```



Function structure

The structure of a function has three basic parts:

- **Inputs** (or **arguments**)

- **Body** (code that is executed)
- **Output** (or **return value**)

R doesn't let your function have multiple outputs, but you can return a list

- Calls other functions `ifelse()`, `abs()`, operators `^` and `>`, and could also call other functions we've written
- `return()` says what the output is; alternately, return the last evaluation
- **Comments**: Not required by R, but a good idea

Default return value

With no explicit `return()` statement, the default is just to return whatever is on the last line. So the following is equivalent to what we had before

```
psi.1 <- function(x) {
  psi <- ifelse(x^2 > 1, 2*abs(x)-1, x^2)
  psi
}
```

Multiple inputs

```
# "Robust" loss function, for outlier-resistant regression
# Inputs: vector of numbers (x), scale for crossover (c)
# Outputs: vector with x^2 for small entries, 2c|x|-c^2 for large ones
psi.2 <- function(x, c) {
  psi <- ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(psi)
}
identical(psi.1(z), psi.2(z,c=1))
```

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

Default inputs

Our function can also specify default values for the inputs (if the user doesn't specify an input in the function call, then the default value is used)

```
# "Robust" loss function, for outlier-resistant regression
# Inputs: vector of numbers (x), scale for crossover (c)
# Outputs: vector with x^2 for small entries, 2c|x|-c^2 for large ones
psi.2 <- function(x, c = 1) {
  psi <- ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(psi)
}
identical(psi.2(z,c=1), psi.2(z))
```

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

The dangers of using inputs without names

While named inputs can go in any order, unnamed inputs must go in the proper order (as they are specified in the function's definition). Sometimes, the code would even throw an error:

```
psi.2(z, 1)
```

```
## [1]  0.25  9.00  0.81 17.00
```

```
psi.2(1, z)
```

```
## [1] -1.25  1.00  0.99  1.00
```

The dangers of using inputs without names

When calling a function with multiple arguments, **use input names** for safety, unless you're absolutely certain of the right order for (some) inputs

Named arguments can go in any order when explicitly tagged:

```
identical(psi.2(x=z,c=2), psi.2(c=2,x=z))
```

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

Checking Arguments

Problem: Odd behavior when arguments aren't as we expect

```
psi.2(x=z,c=c(1,1,1,10))
```

```
## [1]  0.25  9.00  0.81 81.00
```

```
psi.2(x=z,c=-1)
```

```
## [1]   0.25 -11.00   0.81 -19.00
```

Solution: Put little sanity checks into the code

```
# "Robust" loss function, for outlier-resistant regression
# Inputs: vector of numbers (x), scale for crossover (c)
# Outputs: vector with x^2 for small entries, 2c|x|-c^2 for large ones
psi.3 <- function(x,c=1) {
  # Scale should be a single positive number
  stopifnot(length(c) == 1,c>0)
  psi <- ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(psi)
}
```

Arguments to `stopifnot()` are a series of expressions which should all be TRUE; execution halts, with error message, at *first* FALSE (try it!)

Returning more than one thing

When creating a function in R, though you cannot return more than one output, you can **return a list**. This (by definition) can contain an arbitrary number of arbitrary objects

```
# "Robust" loss function, for outlier-resistant regression
# Inputs: vector of numbers (x), scale for crossover (c)
# Outputs: vector with x^2 for small entries, 2c|x|-c^2 for large ones
psi.4 <- function(x,c=1) {
  # Scale should be a single positive number
  stopifnot(length(c) == 1,c>0)
  psi <- ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(list(psi = psi, x = x, c = c))
}
psi.4(z)
```

```
## $psi
## [1]  0.25  9.00  0.81 17.00
##
## $x
## [1] -0.5 -5.0  0.9  9.0
##
## $c
## [1] 1
```

Size effects

A **side effect** of a function is something that happens as a result of the function's body, but is not returned. Examples:

- Printing something out to the console
- Plotting something on the display
- Saving an R data file, or a PDF, etc.

```
# "Robust" loss function, for outlier-resistant regression
# Inputs: vector of numbers (x), scale for crossover (c)
# Outputs: vector with x^2 for small entries, 2c|x|-c^2 for large ones
psi.5 <- function(x,c = 1) {
  # Scale should be a single positive number
  stopifnot(length(c) == 1,c>0)
  cat(paste0("x = ", x, ", c = ", c, "\n"))
  psi <- ifelse(x^2 > c^2, 2*c*abs(x)-c^2, x^2)
  return(list(psi = psi, x = x, c = c))
}
psi.5(z)
```

```
## x = -0.5, c = 1
##  x = -5, c = 1
##  x = 0.9, c = 1
##  x = 9, c = 1
```

```
## $psi
## [1]  0.25  9.00  0.81 17.00
##
## $x
## [1] -0.5 -5.0  0.9  9.0
##
## $c
## [1] 1
```

Environment: what the function can see and do

- Each function has its own environment

- Names here over-ride names in the global environment

- Internal environment starts with the named arguments

- Assignments inside the function only change the internal environment (There *are* ways around this, but they are difficult and best avoided)
- Names undefined in the function are looked for in the environment the function gets called from *not* the environment of definition

Internal environment examples

```
x <- 7
y <- c("A","C","G","T","U")
adder <- function(y) { x<- x+y; return(x) }
adder(1)
```

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

```
x
```

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

```
y
```

```
## [1] "A" "C" "G" "T" "U"
```

```
circle.area <- function(r) { return(pi*r^2) }
circle.area(c(1,2,3))
```

```
## [1]  3.141593 12.566371 28.274334
```

```
truepi <- pi
pi <- 3
circle.area(c(1,2,3))
```

```
## [1]  3 12 27
```

```
pi <- truepi      # Restore sanity
circle.area(c(1,2,3))
```

```
## [1]  3.141593 12.566371 28.274334
```

Relying on variables outside of the function's environment

- Generally OK for built-in constants like `pi`, `letters`, `month.names`, etc.
- Generally not OK for user-defined variables outside of the function
- For the latter, pass these as input arguments to your function

Bad side effects

Not all side effects are desirable. One particularly **bad side effect** is if the function's body changes the value of some variable outside of the function's environment

- Not easy to do (we won't even tell you how)
- But can be done and should be avoided at all costs!

Top-down function design

1. Start with the big-picture view of the task
2. Break the task into a few big parts
3. Figure out how to fit the parts together
4. Repeat this for each part

Start off with a code sketch

You can write top-level code, right away, for your function's design:

```
# Not actual code
big.job = function(lots.of.arguments) {
  first.result = first.step(some.of.the.args)
  second.result = second.step(first.result, more.of.the.args)
  final.result = third.step(second.result, rest.of.the.args)
  return(final.result)
}
```

After you write down your design, go ahead and write the sub-functions (here `first.step()`, `second.step()`, `third.step()`). The process may be iterative, in that you may write these sub-functions, then go back and change the design a bit, etc.

With practice, this design strategy should become natural

Summary

- Function: formal encapsulation of a block of code; generally makes your code easier to understand, to work with, and to modify
- Functions are absolutely critical for writing (good) code for medium or large projects
- A function's structure consists of three main parts: inputs, body, and output
- R allows the function designer to specify default values for any of the inputs
- R doesn't allow the designer to return multiple outputs, but can return a list
- Side effects are things that happen as a result of a function call, but that aren't returned as an output
- Top-down design means breaking a big task into small parts, implementing each of these parts, and then putting them together