Introduction to Functions

Data Wrangling and Husbandry

2/17/2020

R functions by example

Although R functions can be quite complicated, the basics are pretty simple.

1. Give a name to the function. Everyone you know will be happier if you choosed informative names instead of names such as `f`.
2. Write the arguments to the function inside the parentheses of `function()`
3. Write the body of the function inside curly braces `{ }`

```r
raise_to_power <- function(x){
  x^2
}

raise_to_power(-1.414)
```

```
## [1] 1.999396
```

```r
raise_to_power(1:4)
```

```
## [1]  1  4  9 16
```

```r
raise_to_power("k")
```

```
## Error in x^2: non-numeric argument to binary operator
```

Functions can have several arguments:

```
raise_to_power <- function(x, exponent){
  x^exponent
}

raise_to_power(5, 2)
```

```
## [1] 25
```

```
raise_to_power(-1.414, exponent = 4)
```

```
## [1] 3.997584
```

```
raise_to_power(1:4, exp = 2:3)
```

```
## [1]  1  8  9 64
```

```
# R uses partial matching for arguments
```

Those arguments can have defaults.

```
raise_to_power <- function(x, exponent = 2){
  x^exponent
}

raise_to_power(5)
```

```
## [1] 25
```

```
raise_to_power(-1.414, 4)
```

```
## [1] 3.997584
```

This is particularly useful if you are adding arguments to a function that you've defined and used before—old code that calls the function will still work.

- Objects that are inside the body don't survive once the function is finished; they also take precedence over objects with the same names outside the function.
- Functions can have multiple lines. Any object produced or named on the last line is returned (there is also a `return()` function that can be used in special cases).

```
raise_to_power <- function(x, exponent = 2, verbose = FALSE
  result <- x^exponent
  if (verbose) cat("result:", result, "\n")
  result
}
result <- 5
raise_to_power(0, verbose = TRUE)
```

```
## result: 0
```

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

Be very, very careful about using objects—other than functions—that you've defined outside of the function. Otherwise your function implicitly depends on things that may change without your being aware of it. Pass constants via arguments instead.

(Objects undefined in the function are looked for in the environment R uses when the function is called, not when it's defined)

Why write a function?

- Saves work
- Makes code easier to understand
- Makes code easier to update
- Makes code less error prone

Example

```
reformat_phone_number <- function(x, sep = "-"){
  library(stringr)
  str_replace_all(x, "\\(?([2-9][0-9]{2})\\)?[- .]?([0-9]{3
}

reformat_phone_number(c("3214567890", "(321) 456-7890"))
```

```
## [1] "321-456-7890" "321-456-7890"
```

```
reformat_phone_number(c("3214567890", "(321) 456-7890"), se
```

```
## [1] "321.456.7890" "321.456.7890"
```

Conditionals

Conditionals can be used anywhere in R, but are particularly useful in functions. The basic pattern is

```
if (condition) {code that is executed if condition is true}
```

or

```
if (condition){
  code that is executed if condition is true
} else {
  code that is executed if condition is false
}
```

```r
generate_random_variables <- function(n, distribution = "no
  if (!(distribution %in% c("norm", "t"))) stop("Distributi
  if (distribution == "norm") {
    return(rnorm(n))
  } else {
    return(rt(n, df = df))
  }
}
generate_random_variables(2)
```

```
## [1]  0.8107244 -0.1656866
```

```r
generate_random_variables(2, distribution = "t", df = 1)
```

```
## [1] -0.2491007  4.9881491
```

```r
generate_random_variables(2, distribution = "unif")
```

```
## Error in generate_random_variables(2, distribution = "un
```

...

You can use ... in your function to pass unspecified arguments to a function within your function.

```
generate_random_variables <- function(n, distribution = "no
  if (!(distribution %in% c("norm", "t"))) stop("Distributi
  if (distribution == "norm") {
    return(rnorm(n, ...))
  } else {
    return(rt(n, df = df, ...))
  }
}
mean(generate_random_variables(1000, mean = -5))
```

```
## [1] -4.989132
```

Debugging

- If you write small functions, debugging may be obvious
- `traceback()` will list the sequence of calls that lead to the error
- RStudio has a "Rerun with Debug" tool that will step you through the function
- You can use a command like `debug(generate_random_variables)` (and its companion `undebug()`) to step through a particular function

Final points

- Keep your functions small—get your work done by combining many functions
- If you use your function beyond the immediate project, include error checking code

In class exercises

1. Create a function that takes an `lm` object and returns the R^2 of the model (and nothing else).
2. Modify your function so that you can optionally get the adjusted R^2_{adj} instead—the function should still given R^2 by default.
3. Modify your function so that if the first argument is not of class "lm" the function gives an error message
4. Create a new function that, given an `lm` object, returns the 5 residuals with the largest absolute values (but return the residuals, not the absolute value of the residuals).
5. Modify that function so that the number returned can be varied.
6. Modify that function to give a clear error message if the argument for number is larger than the number of residuals