Functions

Statistical Computing, STA3005

Tuesday Mar 6, 2024

Last chapter: Plotting tools

- `plot()` : generic plotting function
- `points()` : add points to an existing plot
- `lines()` , `abline()` : add lines to an existing plot
- `text()` , `legend()` : add text to an existing plot
- `rect()` , `polygon()` : add shapes to an existing plot
- `hist()` , `image()` : histogram and heatmap
- `heat.colors()` , `topo.colors()` , etc: create a color vector
- `density()` : estimate density, which can be plotted
- `contour()` : draw contours, or add to existing plot
- `curve()` : draw a curve, or add to existing plot

Part I: *Function basics*

Why do we need 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

Remember those commands you typed over and over?

From our lectures on text manipulation and regular expressions:

```
# Get King's word counts
king.lines = readLines("king.txt")
king.text = paste(king.lines, collapse=" ")
king.text = gsub(pattern = "[[:punct:]]", replacement = "", king.text)
king.words = strsplit(king.text, split="[[:space:]]")[[1]]
king.words = king.words[king.words != ""]
king.wordtab = table(king.words)
```

Creating your own function

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

```
# get.wordtab.king: get a word table from King's "I Have A Dream" speech
# Input: none
# Output: word table, i.e., vector with counts as entries and associated
#   words as names

get.wordtab.king = function() {
  lines = readLines("king.txt")
  text = paste(lines, collapse=" ")
  text = gsub(pattern = "[[:punct:]]", replacement = "", text)
  words = strsplit(text, split="[[:space:]]")[[1]]
  words = words[words != ""]
  wordtab = table(words)
```

```
  return(wordtab)
}

king.wordtab.call <- get.wordtab.king()
all(king.wordtab == king.wordtab.call)
```

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

Much better: create a word table function that takes a file name

```
# get.wordtab.from.file: get a word table from a txt file
# Input:
# - fname: string, specifying the name of a file to be split
# Output: word table, i.e., vector with counts as entries and associated
#   words as names

get.wordtab.from.file = function(fname) {
  lines = readLines(fname)
  text = paste(lines, collapse=" ")
  text = gsub(pattern = "[[:punct:]]", replacement = "", text)
  words = strsplit(text, split="[[:space:]]")[[1]]
  words = words[words != ""]
  wordtab = table(words)
  return(wordtab)
}
```

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 allow your function to have multiple outputs, but you can return a list

Using (calling) your created function

Our created functions can be used or called just like the built-in ones

```
# Using our function
king.wordtab.new = get.wordtab.from.file("king.txt")
all(king.wordtab.new == king.wordtab)
```

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

```
# Revealing our function's definition
get.wordtab.from.file    不加括号
```

```
## function(fname) {
##   lines = readLines(fname)
##   text = paste(lines, collapse=" ")
##   text = gsub(pattern = "[[:punct:]]", replacement = "", text)
##   words = strsplit(text, split="[[:space:]]")[[1]]
##   words = words[words != ""]
##   wordtab = table(words)
##   return(wordtab)
## }
```

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

```
get.wordtab.from.file = function(fname) {
  lines = readLines(fname)
  text = paste(lines, collapse=" ")
  text = gsub(pattern = "[[:punct:]]", replacement = "", text)
  words = strsplit(text, split="[[:space:]]")[[1]]
  words = words[words != ""]
  table(words)
}
```

Multiple inputs

Our function can take more than one input

```
# get.wordtab.from.file: get a word table from a txt file
# Inputs:
# - fname: string, specifying the name of a file
# - split: string, specifying what to split on
# Output: word table, i.e., vector with counts as entries and associated
#   words as names

get.wordtab.from.file = function(fname, split) {
  lines = readLines(fname)
  text = paste(lines, collapse=" ")
  text = gsub(pattern = "[[:punct:]]", replacement = "", text)
  words = strsplit(text, split=split)[[1]]
  words = words[words != ""]
  table(words)
}
```

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)

```
# get.wordtab.from.file: get a word table from a txt file
# Inputs:
# - fname: string, specifying the name of a file
# - split: string, specifying what to split on. Default is the regex pattern
#   "[[:space:]]"
# - tolower: Boolean, TRUE if words should be converted to lower case before
#   the word table is computed. Default is TRUE
# Output: word table, i.e., vector with counts as entries and associated
#   words as names

get.wordtab.from.file = function(fname, split="[[:space:]]",
                                 tolower=TRUE) {
  lines = readLines(fname)
  text = paste(lines, collapse=" ")
  text = gsub(pattern = "[[:punct:]]", replacement = "", text)
  words = strsplit(text, split=split)[[1]]
  words = words[words != ""]

  # Convert to lower case, if we're asked to
  if (tolower) words = tolower(words)
```

```
  table(words)
}
```

Examples of function calls

Inputs can be called by name, or without names

```
king.wordtab1 = get.wordtab.from.file(fname="king.txt",
  split="[[:space:]]", tolower=TRUE)
king.wordtab2 = get.wordtab.from.file("king.txt",
  "[[:space:]]", TRUE)
all(king.wordtab2 == king.wordtab1)
```

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

Inputs can be called by partial names (if uniquely identifying)

```
king.wordtab3 = get.wordtab.from.file(fn="king.txt",
  spl="[[:space:]]", tolower=TRUE)
all(king.wordtab3 == king.wordtab1)
```

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

When inputs aren't specified, default values are used

```
king.wordtab4 = get.wordtab.from.file(
  fname="king.txt",
  split="[[:space:]]")
all(king.wordtab4 == king.wordtab1)
```

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

Named inputs can go in any order

```
king.wordtab5 = get.wordtab.from.file(
  tolower=TRUE, split="[[:space:]]",
  fname="king.txt")
all(king.wordtab5 == king.wordtab1)
```

```
## [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). E.g., the following code would throw an error:

```
king.wordtab6 = get.wordtab.from.file("[[:space:]]",
  "king.txt",
  tolower=FALSE)
```

```
## Warning in file(con, "r"): cannot open file '[[:space:]]': Invalid argument
```

```
## Error in file(con, "r"): cannot open the connection
```

because our function would regard `[[:space:]]` as the name of a file and try to open up

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

Part II: *Return values and side effects*

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

```
# get.wordtab.from.file: get a word table from a txt file
# Inputs:
# - fname: string, specifying the name of a file
# - split: string, specifying what to split on. Default is the regex pattern
#   "[[:space:]]"
# - tolower: Boolean, TRUE if words should be converted to lower case before
#   the word table is computed. Default is TRUE
# - keep.nums: Boolean, TRUE if words containing numbers should be kept in the
#   word table. Default is FALSE
# Output: list, containing word table, and then some basic numeric summaries

get.wordtab.from.file = function(fname, split="[[:space:]]",
                                 tolower=TRUE, keep.nums=FALSE) {
  lines = readLines(fname)
  text = paste(lines, collapse=" ")
  text = gsub(pattern = "[[:punct:]]", replacement = "", text)
  words = strsplit(text, split=split)[[1]]
  words = words[words != ""]

  # Convert to lower case, if we're asked to
  if (tolower) words = tolower(words)

  # Get rid of words with numbers, if we're asked to
  if (!keep.nums)
    words = grep("[0-9]", words, inv=TRUE, val=TRUE)

  # Compute the word table
  wordtab = table(words)

  return(list(wordtab=wordtab,
              number.unique.words=length(wordtab),
              number.total.words=sum(wordtab),
              longest.word=words[which.max(nchar(words))]))
}
```

- If the `inv` argument of `grep` is `TRUE`, we extract strings not matching the regular expression.

```
# King's "I Have A Dream" speech
king.wordtab = get.wordtab.from.file("king.txt")
lapply(king.wordtab, head)
```

```
## $wordtab
## words
##       a    able   again     ago   ahead alabama
##      37       8       2       1       1       3
##
## $number.unique.words
## [1] 529
##
```

```
## $number.total.words
## [1] 1621
##
## $longest.word
## [1] "discrimination"
```

```
# Lincoln's Gettysburg address
lincoln.wordtab = get.wordtab.from.file("lincoln.txt")
lapply(lincoln.wordtab, head)
```

```
## $wordtab
## words
##        a    above      add advanced      ago      all
##        7        1        1        1        1        1
##
## $number.unique.words
## [1] 138
##
## $number.total.words
## [1] 271
##
## $longest.word
## [1] "proposition"
```

Side effects

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

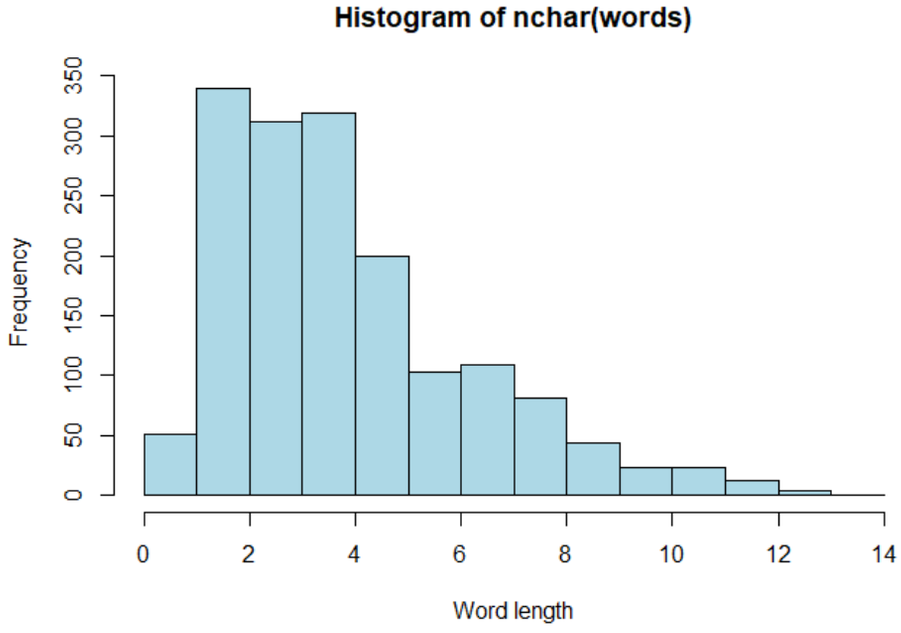
- Printing something out to the console
- Plotting something on the display
- Saving an R data file, or a PDF, etc.

会运行，但除非 specify，否则不会 return

```
# get.wordtab.from.file: get a word table from a txt file
# Inputs:
# - fname: string, specifying the name of a file
# - split: string, specifying what to split on. Default is the regex pattern
#   "[[:space:]]"
# - tolower: Boolean, TRUE if words should be converted to lower case before
#   the word table is computed. Default is TRUE
# - keep.nums: Boolean, TRUE if words containing numbers should be kept in the
#   word table. Default is FALSE
# - hist: Boolean, TRUE if a histogram of word lengths should be plotted as a
#   side effect. Default is FALSE
# Output: list, containing word table, and then some basic numeric summaries

get.wordtab.from.file = function(fname, split="[[:space:]]",
                                 tolower=TRUE, keep.nums=FALSE, hist=FALSE) {
  lines = readLines(fname)
  text = paste(lines, collapse=" ")
  text = gsub(pattern = "[[:punct:]]", replacement = "", text)
  words = strsplit(text, split=split)[[1]]
  words = words[words != ""]

  # Convert to lower case, if we're asked to
  if (tolower) words = tolower(words)

  # Get rid of words with numbers, if we're asked to
  if (!keep.nums)
    words = grep("[0-9]", words, inv=TRUE, val=TRUE)
```

```
  # Plot the histogram of the word lengths, if we're asked to
  if (hist)
    hist(nchar(words), col="lightblue", breaks=0:max(nchar(words)),
         xlab="Word length")

  # Compute the word table
  wordtab = table(words)

  return(list(wordtab=wordtab,
              number.unique.words=length(wordtab),
              number.total.words=sum(wordtab),
              longest.word=words[which.max(nchar(words))]))
}
```

```
# King's speech
king.wordtab = get.wordtab.from.file(
  fname="king.txt",
  hist=TRUE)
```

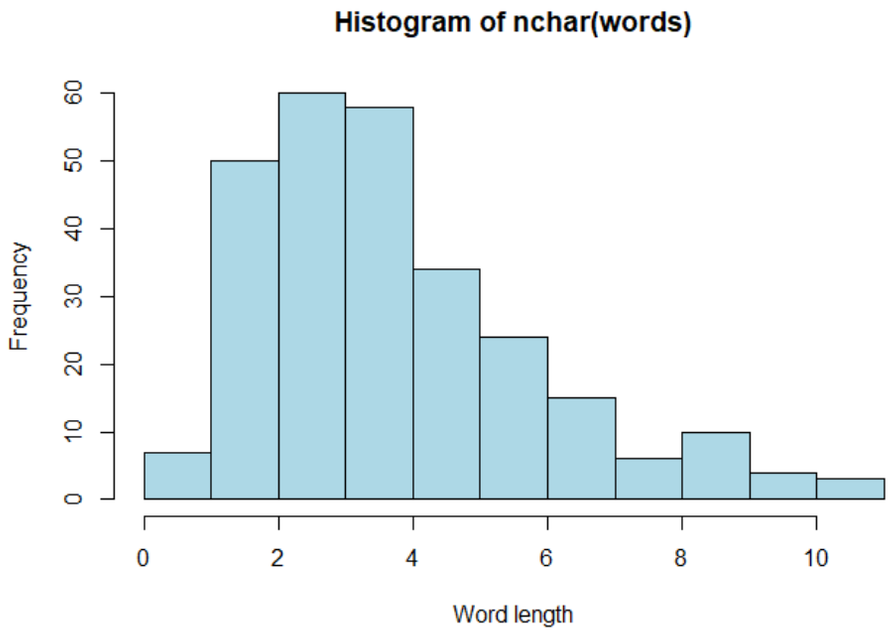



```
lapply(king.wordtab, head)
```

```
## $wordtab
## words
##       a    able   again     ago   ahead alabama
##      37       8       2       1       1       3
##
## $number.unique.words
## [1] 529
##
## $number.total.words
## [1] 1621
##
## $longest.word
## [1] "discrimination"
```

```
# Lincoln's speech
lincoln.wordtab = get.wordtab.from.file(
```

```
  fname="lincoln.txt",
  hist=TRUE)
```




```
lapply(lincoln.wordtab, head)
```

```
## $wordtab
## words
##        a    above      add advanced      ago      all
##        7        1        1        1        1        1
##
## $number.unique.words
## [1] 138
##
## $number.total.words
## [1] 271
##
## $longest.word
## [1] "proposition"
```

Part III: *Environments and design*

Environment: what the function can see and do

Environment is the place where data objects and functions are stored.

- Each function has its own environment, called internal environment.
- Internal environment starts with the named arguments.
- Functions can also have access to the global environment.

`ls()` returns all the variables and functions in the global environment (not include built-in variables and functions as well as functions in packages).

```
ls()
```

```
##  [1] "get.wordtab.from.file" "get.wordtab.king"      "king.lines"
##  [4] "king.text"             "king.words"            "king.wordtab"
##  [7] "king.wordtab.call"     "king.wordtab.new"      "king.wordtab1"
## [10] "king.wordtab2"         "king.wordtab3"         "king.wordtab4"
## [13] "king.wordtab5"         "lincoln.wordtab"
```

Scope of variables

Variables (arguments) defined inside the function only exist in the internal environment, while variables defined in the global environment are available in the internal environment. The scope of a variable is the places where the variable can be referenced and used. To understand the variable scope, we consider the following three cases:

- Variables only defined inside the function: Assignments inside the function only change the internal environment
- Variables only defined outside the function: Names undefined in the function are looked for in the global environment
- Variables with the same name inside and outside the function: Names in the internal environment override names in the global environment

不会改变global variable

```
# area only defined in circle.area
# pi defined in global
circle.area = function(r) {
  S = pi*r^2
  print(S)
}
circle.area(1:3)
```

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

```
S
```

defined inside the function

```
## Error in eval(expr, envir, enclos): object 'S' not found
```

```
true.pi = pi
pi = 3 # Valid in 1800s Indiana
circle.area(1:3)
```

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

```
pi = true.pi # Restore sanity
circle.area(1:3)
```

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

```
# x and y defined in both
x = 7
y = c("A","C","G","T","U")
adder = function(y) { x = x+y; x }
adder(1)
```

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

```
x
```

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

```
y
```

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

Replying on variables in the global environment

If a function replies on variables defined in the global 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