Session 4: Custom Functions, Iteration/Looping, & Branching

Foundations of Quantitative Ecology (EEOB 8896.11)

Paul J. Hurtado
(hurtado.10@mbi.osu.edu)

Mathematical Biosciences Institute (MBI)
The Ohio State University

Last compile: September 11, 2013

Functions

Often, it's helpful to create custom functions for tasks we'll run more than once.

```
## See ?function for details.
MyFuncName <- function(arg1, arg2, arg3 = default.value) {
    ## code that does something with the inputs
    return(output)
}
## To allow vector inputs, not just single-valued inputs
MyNewFunc <- Vectorize(MyFuncName, c("arg1", "arg2"))
## Speed things up!
library(compiler)
MyNewFunc <- cmpfun(MyFuncName)
```

Notice we're passing functions as arguments to functions!

Functions

Often, it's helpful to create custom functions for tasks we'll run more than once.

```
## See ?function for details.
MyFuncName <- function(arg1, arg2, arg3 = default.value) {
    ## code that does something with the inputs
    return(output)
}
## To allow vector inputs, not just single-valued inputs
MyNewFunc <- Vectorize(MyFuncName, c("arg1", "arg2"))
## Speed things up!
library(compiler)
MyNewFunc <- cmpfun(MyFuncName)
```

Notice we're passing functions as arguments to functions!
Q: What happens if we call a function without parentheses?

Iteration and Looping

There are multiple ways to repeat procedures in R.

Iteration and Looping

There are multiple ways to repeat procedures in R.

Some are common to nearly all programming languages:

```
## Like for loops, that iterate over a list
for (i in 1:100) {
    Output[i] <- Do.Something(i)
}
## or while loops, which repeat until a condition is met
i = 0
while (times[i] < end_time) {
    i <- i + 1
    Output[i] <- Do.Something(i)
    times[i] <- Update.time(i)
}
```

Iteration and Looping

Others are specific to R, e.g., the `apply()` family of functions.

```
## This is standard
for (i in 1:100) {
    Output[i] <- Do.Something(i)
}
## Equivalently, in R
Output <- lapply(1:100, Do.Something)
## See ?sapply, ?apply, ?mapply for details.  To replicate something n times
## without varying input values:
replicate(n = 5, {
    xydata = data.frame(x = 1:100, y = rnorm(100, 1:100, 1))
    fit = lm(y ~ x, xydata)
    return(fit$coefficients)
})
```

Iteration and Looping

`for` and `while` loops are fundamental programming tools that help us with "computational thinking", **BUT** in practice, write **vectorized** code!

```
## Not vectorized!
Output[1] <- Do.Something(1)  ## Horrible.
Output[2] <- Do.Something(2)  ## Coding.
Output[3] <- Do.Something(3)  ## Style.
...
Output[100] <- Do.Something(100)  ## Ugh!
## Still not vectorized, but better!
for (i in 1:100) {
    Output[i] <- Do.Something(i)
}
## Better, but on par with the for loop
Output <- lapply(1:100, Do.Something)
## Yes!
Output <- Do.Something(1:100)  ## See ?Vectorize
```

Iteration and Looping

`for` and `while` loops are fundamental programming tools that help us with "computational thinking", **BUT** in practice, write **vectorized** code!

```
## Not vectorized!
Output[1] <- Do.Something(1)  ## Horrible.
Output[2] <- Do.Something(2)  ## Coding.
Output[3] <- Do.Something(3)  ## Style.
...
Output[100] <- Do.Something(100)  ## Ugh!
## Still not vectorized, but better!
for (i in 1:100) {
    Output[i] <- Do.Something(i)
}
## Better, but on par with the for loop
Output <- lapply(1:100, Do.Something)
## Yes!
Output <- Do.Something(1:100)  ## See ?Vectorize
```

Vectorized code is (1) easier to write/read, **AND** (2) computationally efficient. `Vectorize()` may not always be the solution!

Vectorize!

```
## Function to benchmark
Do.OneThing <- function(x=20,seed=1) { set.seed(seed)
  svd(matrix(rnorm(x),x))$d
}
N=5000;
## How fast in a for loop?
system.time({ dummy=c(); for(i in 1:N) dummy[i] <- Do.OneThing(i) })

##    user  system elapsed
##    4.23    0.00    4.87

## Using an apply function
system.time(Output <- lapply(1:N, Do.OneThing))

##    user  system elapsed
##    4.37    0.00    4.37

## Or vectorize (literally!)
Do.Something <- Vectorize(Do.OneThing,"x") ## See ?Vectorize
system.time(Output <- Do.Something(1:N))

##    user  system elapsed
##    4.21    0.00    4.87
```

Exercises

Ex. 1: Write a while loop that draws exponentially distributed random variables (rate=1) until they sum up to 100. Output: the vector of values.
Ex. 2: Write something similar that draws 100 such exponentially distributed values using a for loop.
Ex. 4: Do the same, but using `replicate()`.
Bonus: Plot the liklihood of those data for rates in the range (0, 5).

Branching with `if/else`

Branching refers to doing different things depending on the input. For example:

```
## See ?Control, ?ifelse for details
if (condition == TRUE) {
    Do.This()
} else {
    Do.That()
}
## or
if (condition == TRUE) {
    Do.This()
} else if (otherthing == TRUE) {
    Do.That()
} else {
    Do.nothing()
}
## or
ifelse(logical.vector, val.if.true, val.if.false)
## Ex: color points by sign of y values:
plot(x, y, col = ifelse(yval >= 0, "red", "blue"))
```

Regular Expressions

`if/else` conditions are often based on strings. Regular expressions help with parsing for such purposes.

```
my.strs = c("Peter Piper    picked", "a peck of    pickled peppers.")
gsub("\\s+", ".", my.strs)  # see ?regex

## [1] "Peter.Piper.picked"         "a.peck.of.pickled.peppers."

grep("pickle", my.strs, value = TRUE)  # see ?grep

## [1] "a peck of    pickled peppers."

grepl("pickle", my.strs)

## [1] FALSE  TRUE

strsplit(paste(my.strs, collapse = " "), "\\s+")

## [[1]]
## [1] "Peter"    "Piper"    "picked"   "a"          "peck"      "of"
## [7] "pickled"  "peppers."
```

Project

Exercise:

Download the BBS Data from the course website, and start a new project directory.

Write code that iterates through bird species, and if the name contains "Warbler", plot the trend data.

Modify your code to write the plots to files.