Statistics 360: Advanced R for Data Science

Lecture 3

Brad McNeney

Control Flow

R Functions

Control Flow

Control Flow

- Reading: text, chapter 5
- if/if-else, ifelse, switch
- for
- while
- break

if and if-else

- `if` tests a condition and executes code if the condition is true. Optionally, can couple with an `else` to specify code to execute when condition is false.

```
if("cat" == "dog") {
  print("cat is dog")
} else {
  print("cat is not dog")
}
```

```
## [1] "cat is not dog"
```

if returns a value

- The body of the if-else can evaluate expressions and store results, but note that if-else also returns a value.

```
cnd <- if("cat" == "dog")  "cat is dog" else "cat is not dog"
cnd
```

```
## [1] "cat is not dog"
```

`if` expects a single logical

- most other inputs will cause an error
- logical vectors will not throw an error, but `if` will only use the first element

```
try(if("cat") print("cat"))
```

```
## Error in if ("cat") print("cat") :
##   argument is not interpretable as logical
```

```
if(c("cat"=="dog","cat" == "cat")) print("hello world")
```

```
## Warning in if (c("cat" == "dog", "cat" == "cat")) print("hello world"): the
## condition has length > 1 and only the first element will be used
```

ifelse(): vectorized if

- ifelse() can handle logical vectors
- syntax is condition, what to return if expression true, what to return if expression false

```
x <- 1:10
ifelse(x %% 2 == 0, "even","odd")
```

```
##  [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

switch

- If you have multiple conditions to check, consider `switch` instead of repeated if-else; e.g.
  - `if(x==1) "cat" else if(x==2) "dog" else if (x==3) "mouse"`

```
x <- 2 # numeric argument
switch(x,"cat","dog","mouse") # evaluate the x'th element
```

```
## [1] "dog"
```

```
x <- "dog" #character argument
switch(x,cat="hi cat",dog="hi dog",mouse="hi mouse",
       warning("unknown animal")) # if we make it to the last condition
```

```
## [1] "hi dog"
```

```
switch("kangaroo",cat="hi cat",dog="hi dog",mouse="hi mouse",
       warning("unknown animal"))
```

```
## Warning: unknown animal
```

for loops

- Example:

```
n <- 10; nreps <- 100; x <- vector(mode="numeric",length=nreps)
for(i in 1:nreps) {
  # Code you want to repeat nreps times
  x[i] <- mean(rnorm(n))
}
summary(x)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -0.84286 -0.23118 -0.07301 -0.06175  0.12130  0.82242
```

```
print(i)
```

```
## [1] 100
```

for loop index set

- Index sets such as `1:n` are most common, but can be almost any atomic vector.

```
ind <- c("cat","dog","mouse")
for(i in ind) {
  print(paste("There is a",i,"in my house"))
}
```

```
## [1] "There is a cat in my house"
## [1] "There is a dog in my house"
## [1] "There is a mouse in my house"
```

seq_along

- A common use of for loops is to iterate over elements of a vector, say x.
- Creating the index set `1:length(x)` will not be what you expect when x has length 0 (e.g., x is NULL).
- Instead use `seq_along()`

```
x <- NULL
for(i in 1:length(x)) print(x[i])
```

```
## NULL
## NULL
```

```
for(i in seq_along(x)) print(x[i])
```

while loops

- Use a `while` loop when you want to continue until some logical condition is met.

```
set.seed(1)
# Number of coin tosses until first success (geometric distn)
p <- 0.1; counter <- 0; success <- FALSE
while(!success)  {
  success <- as.logical(rbinom(n=1,size=1,prob=p))
  counter <- counter + 1
}
counter
```

```
## [1] 4
```

break

- `break` can be used to break out of a `for` or `while` loop.

```
for(i in 1:100) {
  if(i>3) break
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

`repeat`

- `repeat` continues indefinitely until it encounters a `break`
- The text considers `repeat` to be the most flexible of `for`, `while` and `repeat`.

R Functions

R function fundamentals

- Reading: text sections 6.1 and 6.2
- In R, functions are objects with three essential components:
  - the code inside the function, or `body`,
  - the list of arguments to the function, or `formals`, and
  - an `environment` that contains all objects defined in the function.
- Functions can have other attributes, but the above three are essential.

Example function

```
f <- function(x) {
  return(x^2)
}
f
```

```
## function(x) {
##   return(x^2)
## }
```

The function body

- This is the code we want to execute.
- When the end of a function is reached without a call to `return()`, the value of the last line is returned.
  - So in our example function, we could replace `return(x^2)` with just `x^2.

```
body(f)
```

```
## {
##     return(x^2)
## }
```

The function formals

- These are the arguments to the function.
- Function arguments can have default values and/or be defined in terms of other arguments.

```
f <- function(x=0) { x^2}
f <- function(x=0,y=3*x) { x^2 + y^2 }
f()
```

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

```
f(x=1)
```

```
## [1] 10
```

```
f(y=1)
```

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

```
formals(f)
```

```
## $x
## [1] 0
##
## $y
## 3 * x
```

Argument matching when calling a function

- When you call a function, the arguments are matched first by name, then by "prefix" matching and finally by position:

```
f <- function(firstarg,secondarg) {
  firstarg^2 + secondarg
}
f(firstarg=1,secondarg=2)
```

```
## [1] 3
```

```
f(s=2,f=1)
```

```
## [1] 3
```

```
f(2,f=1)
```

```
## [1] 3
```

```
f(1,2)
```

```
## [1] 3
```

The function environment

- The environment within a function is like a map to the memory locations of all its variables.
- Variables created within the function are also store in its environment

```
f <- function(x) {
  y <- x^2
  ee <- environment() # Returns ID of environment w/in f
  print(ls(ee)) # list objects in ee
  ee
}
f(1) # function call
```

```
## [1] "ee" "x"  "y"
```

```
## <environment: 0x55698fe32378>
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

Enclosing environments

- Our function `f` was defined in the global environment, `.GlobalEnv`, which "encloses" the environment within `f`.
- If `f` needs a variable and can't find it within `f`'s environment, it will look for it in the enclosing environment, and then the enclosing environment of `.GlobalEnv`, and so on.
- The `search()` function lists the heirarchy of environments that enclose `.GlobalEnv`.

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics
## [4] "package:grDevices" "package:utils"     "package:datasets
## [7] "package:methods"   "Autoloads"         "package:base"
```

- To facilitate this search, each environment includes a pointer to its enclosing environment.

R packages and the search list

- Use the `library()` command to load packages.
- When we load a package it is inserted in position 2 of the search list, just after `.GlobalEnv`.

```
# install.packages("hapassoc")
library(hapassoc)
search()
```

```
##  [1] ".GlobalEnv"         "package:hapassoc"  "package:stats"
##  [4] "package:graphics"   "package:grDevices" "package:utils"
##  [7] "package:datasets"   "package:methods"   "Autoloads"
## [10] "package:base"
```

Detaching packages

- Detach a package from the search list with `detach()`

```
detach("package:hapassoc")
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

Package namespaces

- Package authors create a list of objects that will be visible to users when the package is loaded. This list is called the package namespace.
- You can access functions in a package's namespace without loading the package using the `::` operator.

```
set.seed(321)
n<-30; x<-(1:n)/n; y<-rnorm(n,mean=x); ff<-lm(y~x)
car::sigmaHat(ff)
```

```
## [1] 0.926726
```

- Doing so does not add the package to the search list.

Up next

- Reading: Text, sections 6.4-6.8