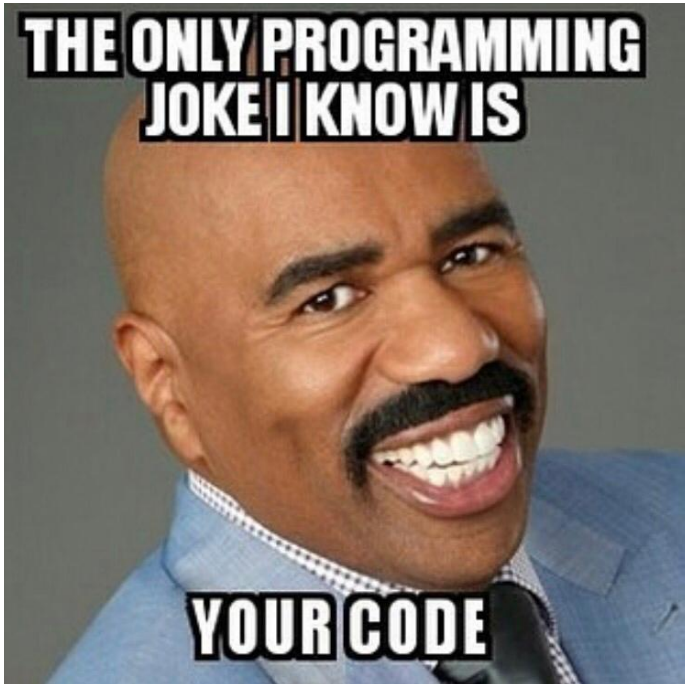


Lecture 4: R continued

Moving beyond basic R statements

Outline

First half: programming in R

- Functions
- Control structures
- Vectorized operations, loop functions

Second half: R under the hood

- Environments
- Scoping
- Dynamic typing

Functions

Take in some input, perform some calculations, and return a value.

Function example

Here is a simple R function:

```
# This function returns the square of a number
# Args:
#    x: A number
# Returns:
#    The square of the number
square <- function(x) {
  x * x
}
```

Now we can **call** the function on a particular input:

```
square(4)
```

```
## [1] 16
```

4/57

Anatomy of an R function

```
# Comments that explain the function
#   - What it does
#   - The arguments
#   - The return value
function_name <- function(arg1, arg2, arg3) {
  # Do some calculations
  # The function returns the last value calculated
}
```

Default argument values

Arguments can have default values. These arguments go at the end of the argument list.

```
# Prints a friendly greeting
# Args:
#   - name: Your name
#   - greeting: A greeting
greet <- function(name, greeting = "Hello") { # greeting has a default value
  print(paste(greeting, name)) # We call two built-in functions here
}
```

```
greet("Pam", "Hey")
```

```
## [1] "Hey Pam"
```

```
greet("Pam") # Defer to the default value of greeting
```

```
## [1] "Hello Pam"
```

Calling functions

Function arguments are recognized by order or by name.

```
greet("Pam", "Hey") # Arguments recognized by order
```

```
## [1] "Hey Pam"
```

```
greet("Pam", greeting = "What's up") # Pass an argument by name
```

```
## [1] "What's up Pam"
```

Why include argument names?

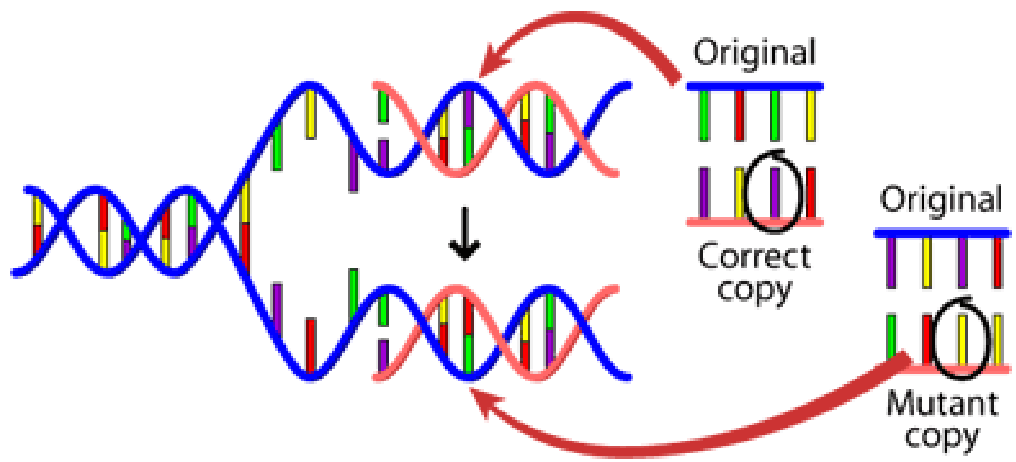
- Better clarity when there are many arguments, less chance of error
- Skip some arguments when there are multiple arguments with defaults
- Technically you can pass named arguments in any order, but it's confusing

DRY (Don't Repeat Yourself)

If you will perform a similar operation more than once, encapsulate it in a function.

Why DRY?

Because copies will mutate separately!



https://evolution.berkeley.edu/evolibrary/article/evo_20

10/57

DRY do's and don'ts

Don't do this!!

```
principal <- 100000
rate <- 0.16
ncompound <- 4
balance_5yr <- principal * (1 + rate / ncompound) ^ (ncompound * 5)

# A bunch of other code...
# ...
# Forget that we calculated interest before...

balance_10yr <- principal * (1 + 0.16 / 4) ^ (4 * 10)
```

DRY do’s and don’ts

Do this instead: write a function!

```
# Calculates the balance of an account after accrued interest
# Args:
#    orig_principal: Original balance
#    interest_rate: Interest rate
#    n_compound: Number of times compounded per year
#    n_year: Number of years
# Returns:
#    Ending balance
acct_balance <- function(orig_principal, interest_rate, n_compound, n_year) {
  orig_principal * (1 + interest_rate / n_compound) ^ (n_compound * n_year)
}
```

DRY do’s and don’ts

Only type each parameter once and call the function

```
principal <- 100000
rate <- 0.16
ncompound <- 4

balance_5yr <- acct_balance(principal, rate, ncompound, 5)
balance_10yr <- acct_balance(principal, rate, ncompound, 10)
```

Now if the interest rate changes…

```
rate <- 0.13 # Only change one line of code
```

14/57

DRY vs. WET

WET code

- Write Everything Twice
- We Enjoy Typing
- Waste Everyone's Time

https://www.reddit.com/r/AnimalsBeingDerps/comments/5j0f48/dry_blop_vs_wet_blop_xpost_rblop/

15/57

Control structures

Decisions about which statements to execute, and in what order, can be made based on runtime conditions.

Repeatedly execute the same behavior

```
fruits <- c("apple", "banana", "pear", "grape")
print(fruits[1])
```

```
## [1] "apple"
```

```
print(fruits[2]) # We are repeating code
```

```
## [1] "banana"
```

```
print(fruits[3]) # Ugghhh
```

```
## [1] "pear"
```

```
print(fruits[4]) # Uggggghhhhhh
```

```
## [1] "grape"
```

17/57

Control structure = freedom

Use a control structure to repeat the same line of code over and over:

```
for (fruit in fruits) {
  print(fruit)
}
```

```
## [1] "apple"
## [1] "banana"
## [1] "pear"
## [1] "grape"
```

The simplest control structure: `if/else`

```
today <- weekdays(Sys.Date()) # The day of the week today
today # Let's see the value
```

```
## [1] "Sunday"
```

```
if(today == "Wednesday") {
  print("Today is Wednesday")
} else {
  print("Today is not Wednesday")
}
```

```
## [1] "Today is not Wednesday"
```

Something different will be printed depending on when we run this code. The decision of what to print is made **at runtime**.

`else` is optional

```
x <- 5
if (x > 4) {print("x is greater than 4")}
```

```
## [1] "x is greater than 4"
```

```
if (x > 6) {print("x is greater than 6")} # No else clause; nothing happens
```

`for` loop

Execute the same block of code for each element of a sequence/list/vector

```
for (fruit in fruits) {
  print(fruit)
}
```

```
## [1] "apple"
## [1] "banana"
## [1] "pear"
## [1] "grape"
```

```
for (i in 1:3) {
  print(i)
}
```

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

Nested `for` loops

You can nest `for` loops.

```
for (fruit in fruits) {
  for (n in c("two", "three", "four")) {
    print(paste(n, " ", fruit, "s", sep = ""))
  }
}
```

```
## [1] "two apples"
## [1] "three apples"
## [1] "four apples"
## [1] "two bananas"
## [1] "three bananas"
## [1] "four bananas"
## [1] "two pears"
## [1] "three pears"
## [1] "four pears"
## [1] "two grapes"
## [1] "three grapes"
## [1] "four grapes"
```

Be careful: nested loops can be inefficient and hard to read.

`while` loop

Keep executing a block of code as long as a condition is true

```
x <- 5
while(x < 8) {
  print(x)
  x <- x + 1 # increment x
}
```

```
## [1] 5
## [1] 6
## [1] 7
```

```
y <- 5
while(y <= 8) { # note the '<='
  print (y)
  y <- y + 1 # increment y
}
```

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
```

23/57

Watch out for infinite loops

This loop will never terminate because `z` will always be less than `8`:

```
z <- 5
while(z < 8) {
  print(z)
}
```

24/57

Applying functions to composite objects

- Vectorized operations
- Loop functions

Vectorized operations

Function applied to a vector is applied individually to each element

```
x <- 1:5
y <- 11:15
x + y
```

```
## [1] 12 14 16 18 20
```

```
x > 3
```

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

26/57

Some vectorized operations

Arithmetic:

`+`
`-`
`*`
`/`

Comparison:

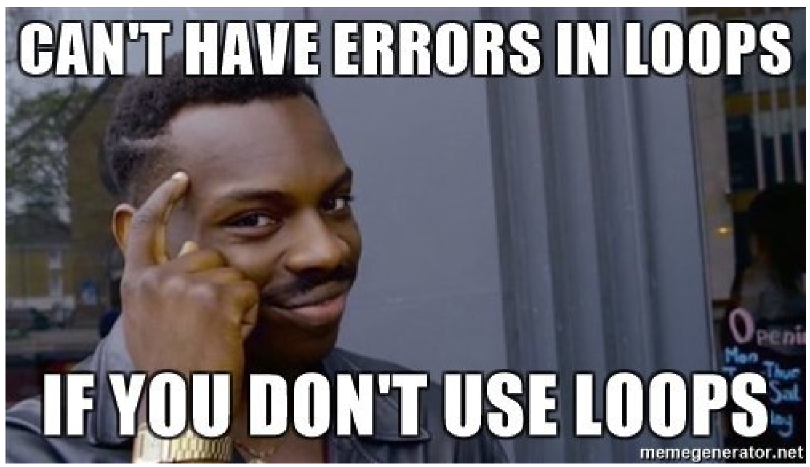
`<`
`>`
`<=`
`>=`
`==`

27/57

Loop functions

Apply a function to each element of a composite object and return a new composite object

Accomplishes the same thing as a `for` loop in a simple line of code



28/57

`lapply`: apply a function to each element of a vector or list

```
# Arguments to lapply: (1) vector or list, (2) function to apply to each element
lapply(c(2,3,4), sqrt)
```

```
## [[1]]
## [1] 1.414214
##
## [[2]]
## [1] 1.732051
##
## [[3]]
## [1] 2
```

`lapply` always returns a list.

`sapply`: simplified `lapply`

`sapply` tries to simplify the return value, returning either a vector or matrix if possible, or a list if it can't figure it out.

```
# Returns a vector instead of a list
sapply(c(2,3,4), sqrt)
```

```
## [1] 1.414214 1.732051 2.000000
```

30/57

Custom functions as arguments

Functions can be passed around as arguments: by name or anonymously.

```
# Pass an anonymous function as an argument to sapply
sapply(c(2,3,4), function(x) {x + 5})
```

```
## [1] 7 8 9
```

```
# Declare a function with a name and pass it as an argument
plusfive <- function(x) {x + 5}
sapply(c(2,3,4), plusfive)
```

```
## [1] 7 8 9
```

31/57

apply

Apply a function to the rows or columns of a matrix

```
# Create a matrix of random numbers
set.seed(1614)
mat <- matrix(data = runif(25), nrow = 5, ncol = 5)
mat
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.5370039 0.6466657 0.2933027 0.5355449 0.7486944
## [2,] 0.3000032 0.9947813 0.5919862 0.8680835 0.4470040
## [3,] 0.4821635 0.3397505 0.9549776 0.7287408 0.6452801
## [4,] 0.4972801 0.5636399 0.7512045 0.1076171 0.2122665
## [5,] 0.8917704 0.4963248 0.8377595 0.1124934 0.8043411
```

apply

Get information about the `apply` function:

```
str(apply)
```

```
## function (X, MARGIN, FUN, ...)
```

`X` is the matrix, `MARGIN` is `1` for rows or `2` for columns, `FUN` is the function to apply

```
# Get the mean of each row
apply(mat, 1, mean)
```

```
## [1] 0.5522423 0.6403717 0.6301825 0.4264016 0.6285378
```

```
# Get the max of each column
apply(mat, 2, max)
```

```
## [1] 0.8917704 0.9947813 0.9549776 0.8680835 0.8043411
```

`apply` with an anonymous function

```
# Print mat again for convenience
mat
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.5370039 0.6466657 0.2933027 0.5355449 0.7486944
## [2,] 0.3000032 0.9947813 0.5919862 0.8680835 0.4470040
## [3,] 0.4821635 0.3397505 0.9549776 0.7287408 0.6452801
## [4,] 0.4972801 0.5636399 0.7512045 0.1076171 0.2122665
## [5,] 0.8917704 0.4963248 0.8377595 0.1124934 0.8043411
```

```
# Get the second largest number in each column
apply(mat, 2, function(col) {
  col_order <- order(col, decreasing = TRUE)
  col_reordered <- col[col_order]
  col_reordered[2]
  # Note: a more concise formulation would be col[order(col, decreasing = TRUE)][2]
})
```

```
## [1] 0.5370039 0.6466657 0.8377595 0.7287408 0.7486944
```

Think twice about writing loops in R

Can your goal be accomplished more consisely with an `apply`?

35/57

Think about the language

How do things work under the hood?

Ultimately your code will be easier to trust and debug.

Functions are first class objects

Can do anything with functions that you can do with normal variables

- Assign them to variables
- Pass them as arguments to functions
- Return them from functions
- Create lists of functions

Functions are first class objects

```
# This function takes a number and returns a new function that
# adds that number to its argument
#
# Args:
#   x: A number
# Returns:
#   A function that takes one argument and adds x to the argument value
f <- function(x) {
  function(y) {
    x + y
  }
}
```

38/57

Return a function from a function

```
f(5) # f returns a function, so f(5) is a function
```

```
## function(y) {
##     x + y
##   }
## <environment: 0x7fd284e3add0>
```

```
# Now call that function on an input
f(5)(3)
```

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

```
# Or give that function a name
g <- f(5)
g(3)
```

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

Environment

Environment is the collection of objects that are available to use.

Environment available at the command prompt is the *global environment*.

```
# Print everything in global environment
# Note: includes functions we have defined
ls()
```

```
##  [1] "acct_balance"  "balance_10yr"  "balance_5yr"   "f"
##  [5] "fruit"         "fruits"        "function_name" "g"
##  [9] "greet"         "i"             "mat"           "n"
## [13] "ncompound"     "plusfive"      "principal"     "rate"
## [17] "square"        "today"         "x"             "y"
```

Removing objects from environment

```
# Remove specific objects
rm(x, y, i, n)
ls()
```

```
##  [1] "acct_balance"  "balance_10yr"  "balance_5yr"   "f"
##  [5] "fruit"         "fruits"        "function_name" "g"
##  [9] "greet"         "mat"           "ncompound"     "plusfive"
## [13] "principal"     "rate"          "square"        "today"
```

```
# Clear entire environment
rm(list=ls())
ls()
```

```
## character(0)
```

Function environments

Functions have access to at least two environments.

- Parent (enclosing) environment where function was defined
- Temporary environment within function body; created each time function is called

Function’s enclosing environment

Environment in which function was defined

- When function is defined, it is bound to the environment in which it was defined
- Often defined in global environment but can be defined inside another function
- Function “remembers” which environment it was defined in

Nested environments

Temporary function environment is nested inside function's enclosing environment

44/57

Name conflicts

Objects inside same environment can’t share same name.

How does R handle objects in nested environments with the same name?

Scoping

Finding the value associated with a variable name

R uses lexical scoping:

- Names defined inside function mask names defined outside function
- If name doesn't exist inside function, looks one level up
- Next level up may be global environment, or another function if function was called inside a function

46/57

Scoping: intuitive behavior

```
x <- 5      # Assign new variable in global environment
f1 <- function() {
  x <- 10   # Assign new variable in temporary environment
  x         # Look for x in current environment first; will find value 10
}
f1()
```

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

47/57

Scoping: modifying enclosing environment

```
y <- 5       # Assign new variable in global environment
f2 <- function() {
  y <- 7     # Assign new variable in temporary environment
  y <<- 10   # Reassign y in enclosing environment (global in this case)
  y          # Look for y in current environment first; will find value 7
}
f2()
```

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

48/57

Scoping: function remembers its parent environment

```
z <- 5      # Assign new variable in global environment
f3 <- function() {
  z <<- 10  # Reassign z in enclosing environment (global)
  z
  # Look for z in current environment first
  # z is not found; go to enclosing environment
  # We have just reassigned z in the enclosing environment
}
f3()
```

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

Function remembers its parent environment

```
a <- 1
b <- 2

# We are defining f in the global environment
# f will always look for a and b in the global environment
f <- function(x) {a * x + b}

g <- function(x) {
  a <- 3
  b <- 4
  f(x)
}

g(2)
```

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

50/57

Function remembers its parent environment

```
a <- 1
b <- 2

# We are defining f in the global environment
# f will always look for a and b in the global environment
f <- function(x) {a * x + b}

g <- function(x) {
  a <- 3
  b <- 4
  f(x)
}

# Reassign a and b in the global environment
a <- 5
b <- 6

g(2)
```

```
## [1] 16
```

51/57

When in doubt…

Don’t use the same variable names in different environments.

Can be useful in certain situations but probably don’t need them.

52/57

Assignment operators

- `<-` Normal assignment of a value to a variable
- `=` Equivalent to `<-` almost always
- `<<-` If a definition exists in parent environment, reassign. Keep going up environments; if no assignment exists, define new variable in global environment.
- Rule of thumb to stay out of trouble: use `<-` for assignment and `=` for argument binding in function calls

Dynamic typing

Object types are checked at runtime.

Interpreter decides which version of function to use at runtime depending on object type.

Dynamic typing pros and cons

Pros:

- Same code can be used for inputs of different types

Cons:

- Errors not caught until runtime
- Often need to write tests for type correctness

Convenience of dynamic typing

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
head(1:10)
```

```
## [1] 1 2 3 4 5 6
```

Both types provide an implementation of `head`.

R decides which version of `head` to call at runtime.

Errors caught late

```
f <- function(x) {
  sqrt(x) + 5
}

f("hello") # Problem won't be caught until we run this
```

```
## Error in sqrt(x): non-numeric argument to mathematical function
```