Lecture 8: Functional Programming

UNIVERSITY OF
SAN FRANCISCO

James D. Wilson

BSDS 100 - Intro to Data Science with R

Functional Programming

The focus of the next few classes will be to turn your existing, informal knowledge of functions into a rigorous understanding of what functions are and how to write efficient code.

Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting:

- You can give a function an evocative name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste.

Outline

- Control Flow: if, while, for, repeat, and switch
- Functions
- Writing Robust Code
- Functionals
- Efficiency of Code

Reference: Chapters 19 and 21 in *R for Data Science* book

Part I: Control Flow

Control Flow

We begin talking about functional programming by first exploring *control flow*, which enables execution of statements repetitively, while only executing other statements if certain conditions are met. Control flow can be subdivided into two main topics:

- Repetition and Looping
  - `for()` loops
  - `while()` loops
  - `repeat()` loops
- Conditional Execution
  - `if()`
  - `if() {} else if() {} else {}`
  - `ifelse()`
  - `switch()`

The `for()` loop

- Executes a statement repetitively until a variable's value is no longer contained in the sequence `seq`
- The generic in-line syntax is

```
for (var in seq) {expression}
```

- A simple example which prints "BSDS 100" 5 times

```
for (i in 1:5) print("BSDS 100")
```

The `for()` loop

- It is possible to iterate over more complex sequences

```
> myVector <- factor(c("A", "A", "B", "C", "C", "C", "ZzZ"))

> for (k in levels(myVector)) print(k)
[1] "A"
[1] "B"
[1] "C"
[1] "ZzZ"
```

The `for()` loop

- The `seq` in a `for()` loop is evaluated at the start of the loop; changing it subsequently does not affect the loop
- You can make an assignment to the looping variable (e.g., *i*) within the body of the loop, but this will not affect the next iteration
- When the loop terminates, the looping variable contains its latest value

The `while()` loop

- Executes a statement repetitively until a condition is no longer true
- The generic in-line syntax is

```
while (condition) {expression}
```

- A simple example which prints “MSAN” 3 times

```
> n <- 3

> while (n > 0) {print("MSAN"); n <- n - 1}
```

- **Note**: the use of the semi-colon is only required when writing more than one in-line expression. When multiple lines are used, the semi-colon can be omitted.

The `repeat()` loop

- The `repeat()` loop can be used when the terminal condition does not apply at the top of the loop
- A `repeat()` loop must be terminated with a `break` command placed somewhere inside `repeat()` loop
- The `break` command immediately exists the innermost active `for()`, `while()` or `repeat()` loop

The `repeat()` loop

- Example

```
> x <- 7

> repeat{
+   print(x)
+   x <- x + 2
+   if (x > 10) break
+ }
[1] 7
[1] 9
```

The `if()` statement

- The `if()` control structure executes a statement if a given condition is true
- The generic in-line syntax is

```
if (condition) {expression}
```

- A simple example

```
> x <- 3

> if (x > 0) print(paste("x is: ", x, sep = ""))
[1] "x is: 3"
```

The `if()` statement

- The multi-line form for `if()` is

```
if (condition) {
    < expression 1 >
    ...
    < expression n >
}
```

The `if() {} else {}` statement

- The `if() {} else {}` control structure executes a statement if a given `condition` is true
- The generic in-line syntax is

```
if (condition) expression_01 else expression_02
```

- A simple example

```
> x <- -3

> if (x > 0) print("x is positive") else print("x is negative")
[1] "x is negative"
```

The `if() {} else {}` statement

- The multi-line form of `if() {} else {}` is

```
if (condition) {
    < expressions >
 } else {
    < alternate expressions >
 }
```

The above will run `expressions` if the `condition` is true, but will run `alternate expressions` if the `condition` is false.

Formatting Pitfalls

- This code snippet will run without error

```
x <- -3

if (x > 0) {
  print(paste("x is: ", x, sep = ""))
  } else {
    print("x is negative")
  }
```

```
[1] "x is negative
```

Formatting Pitfalls

- The code snippet will throw an error

```
x <- -3

if (x > 0) {
  print(paste("x is: ", x, sep = ""))
  }
else {
    print("x is negative")
  }
```

```
Error: unexpected '}' in "  }"
```

`if() {} else if() {} else {}`

- The multi-line form of `if() {} else if() {} else {}` is

```
if (condition_01) {
    < expressions 01 >
 } else if (condition_02) {
    < expressions 02 >
 } else {
    < expressions 03 >
```

- As many `else if () {}` clauses may be chained (sequenced) together as desired

`ifelse()` versus `if() {} else {}`

- If a vector $\mathbf{x} : |\mathbf{x}| > 1$ is passed to an `if()` statement, only the first element of the vector will be evaluated for conditional execution; moreover, `R` will throw a warning
- The `ifelse()` construct is a vectorized version of `if() {} else {}` which tests each element of a vector passed to it

`ifelse()` versus `if() {} else {}`

```
> x <- c(3, 2, 1)

> if ( x > 2) {print("first element in vector > 2")}
[1] "first element in vector > 2"
Warning message:
In if (x > 2) { :
  the condition has length > 1 and only the first element will be used

> ifelse(x > 2, ">2", "<=2")
[1] ">2"  "<=2" "<=2"
```

The `switch()` function

- `switch()` chooses statements from a vector based on the value on an expression
- The multi-line form of `switch()` is

```
switch(expression,
  condition_01 = command_01,
  condition_02 = command_02,
  ...
  condition_n = command_n,
)
```

  - If the expression passed to `switch()` is not a character, it is coerced to `integer`
  - If the expression passed to `switch()` is a character string, then the string is matched exactly (with some small edge cases, see documentation)

The `switch()` function

```
grades <- c("A", "D", "F")

for (i in grades) {
  print(
    switch(i,
           A = "Well Done",
           B = "Alright",
           C = "C's get Degrees!",
           D = "Meh",
           F = "Uh-Oh"
           )
  )
}

[1] "Well Done"
[1] "Meh"
[1] "Uh-Oh"
```

In-Class Lab

`titanic.csv`

1. Using a `for()` loop, recode the entries in the `Survived` variable with `"Survived"` and `"Perished"`
2. Using the `if()` command and loop, create a new variable of type `ordered factor` in the data frame called `ageClass`, and map `Age` to: `"Minor"` if less than 18 yrs; 18 yrs ≤ `"Adult"` ≤ 65 yrs; and `"Senior"` if older than 65 yrs
3. Ordering the passengers in descending order by last name, use a `while()` loop to identify the name of the 100th surviving passenger
4. Using a `switch()` statement, identify each passenger class, `Pclass`, as either `"First Class"`, `"Business Class"` or `"Economy"`, and print the results to the console

In-Class Lab - on your own

`titanic.csv`

- Iterate through the data frame, and for variables that are numeric, create a histogram, for categorical variables create a bar chart, and skip over all others
  1. Be sure to correct and clean the variable types before you run code (e.g., there are only two truly numeric variables)
  2. After creating each graph, be sure to include a pop-up a message that says `"Press Enter for next Graph"` to add a pause in the sequential execution

Part II: Functions

Function Components

All `R` functions have three parts

- the `body()`, the code inside the function
- the `formals()`, the list of arguments which controls how you can call the function
- the `environment()`, the *map* of the location of the function's variables

Function Components

```
> myFunc <- function(x) x^2

> myFunc
function(x) x^2

> formals(myFunc)
$x

> body(myFunc)
x^2

> environment(myFunc)
<environment: R_GlobalEnv>
```

Lab

1. Write a function that takes two arguments, `a` and `b`, and returns rows `a` through `b` of `mtcars`
2. Write a function that takes a numeric vector as an input, squares every value in the vector, appends the squared vector to the original vector in the form of a data frame, and prints the first 10 rows of the data frame to the console
3. Write a function that takes a numeric vector as an input, squares every value in the vector, appends the squared vector to the original vector in the form of a data frame, and then returns and stores the data frame **over** the original vector, i.e., replace the old vector (which was input) with the new data frame (which is returned)

Scoping

- Scoping is the set of rules that govern how `R` looks up the value of a symbol
- There are four basic principles behind `R`'s implementation of lexical scoping:
    1. name masking
    2. functions vs. variables
    3. a fresh start
    4. dynamic lookup

Name Masking

```
rm(list=ls())

myFunc_01 <- function() {
  x <- 1
  y <- 2
  c(x,y)
}

myFunc_01()
```

What does the preceding code return?

```
[1] 1 2
```

The function searches inside itself for `x` and `y`

Name Masking

```
rm(list=ls())

myFunc_01 <- function() {
  x <- 1
  y <- 2
  c(x,y)
}

myFunc_01()
```

What does the preceding code return?

```
[1] 1 2
```

The function searches inside itself for `x` and `y`

Name Masking

If a name isn't defined inside a function, R will look one level up

```
x <- 2

myFunc_02 <- function() {
  y <- 1
  c(x,y)
}

myFunc_02()
```

What does the preceding code return?

```
[1] 2 1
```

- What if you omitted `x <- 2`?

Name Masking

If a name isn't defined inside a function, `R` will look one level up

```
x <- 2

myFunc_02 <- function() {
  y <- 1
  c(x,y)
}

myFunc_02()
```

What does the preceding code return?

```
[1] 2 1
```

- What if you omitted `x <- 2`?

Name Masking

```
rm(list=ls())

x <- 1
myFunc_03 <- function() {
  y <- 2
  myFunc_04 <- function() {
    z <- 3
    c(x,y,z)
  }
  myFunc_04()
}
myFunc_03()
```

What does the preceding code return?

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

Search begins inside the function, then where that function was

Name Masking

```
rm(list=ls())

x <- 1
myFunc_03 <- function() {
  y <- 2
  myFunc_04 <- function() {
    z <- 3
    c(x,y,z)
  }
  myFunc_04()
}
myFunc_03()
```

What does the preceding code return?

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

Search begins inside the function, then where that function was

Name Masking

```
rm(list=ls())

x <- 1
myFunc_03 <- function() {
  x <- 1000
  y <- 2
  myFunc_04 <- function() {
    x <- 99
    z <- 3
    c(x,y,z)
  }
  myFunc_04()
}
myFunc_03()
```

What does the preceding code return?

```
[1] 99 2 3
```

Name Masking

```
rm(list=ls())

x <- 1
myFunc_03 <- function() {
  x <- 1000
  y <- 2
  myFunc_04 <- function() {
    x <- 99
    z <- 3
    c(x,y,z)
  }
  myFunc_04()
}
myFunc_03()
```

What does the preceding code return?

```
[1] 99 2 3
```

Functions vs. Variables

The same principles apply for finding functions just as they do for finding variables

```
rm(list=ls())

myFunc_04 <- function(x) x + 99

myFunc_05 <- function() {
  myFunc_04 <- function(x) x * 2
  myFunc_04(20)
  }

myFunc_05()
```

What does the preceding code return? **Key point**: **don't** give identical names to functions and variables

New Functional Environments for Each Execution

- Every time a function is called, a new environment is called to host execution; each invocation is completely independent
- The following function returns a value of `999` every time

```
# NOTE  rm(list=ls()) is deleted

myFunc_06 <- function() {
  if(!exists("myAtomicVector")){
    myAtomicVector <- 999
  } else {
    myAtomicVector <- myAtomicVector + 1
  }
  print(myAtomicVector)
}

myFunc_06()
```

Real-Time Variable Lookup

A function will search for a value when it's run, **not** when it's created

```
> rm(list=ls())

> myFunc_07 <- function() x

> x <- 15

> myFunc_07()
[1] 15

> x <- 20

 > myFunc_07()
[1] 20
```

Self-Contained Functions

- Variables internal to a function, i.e., variables which are not passed to a function, should be locally scoped to ensure that a function is self-contained
- A function that is not self-contained can cause a pernicious error that can be difficult to identify
- Use the `findGlobals` function from the `codetools` package to identify global variables in a function

Self-Contained Functions

```
> rm(list=ls())

> myFunc_08 <- function() x + 1

# NOTE myFunc_08 is not self-contained

> codetools::findGlobals(myFunc_08)
[1] "+" "x"
```

Lab

- Write any function with locally-scoped variables, confirming there are locally scoped using the `codetools` package

Formal Arguments of a Function

- It is important to distinguish between the formal and actual arguments of a function
- **Formal arguments** are a property of the function

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`na.rm`	a logical value indicating whether `NA` values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Calling Arguments of a Function

- It is important to distinguish between the formal and actual arguments of a function
- **Actual or calling arguments** can vary each time you call a function

```
> mean(x = 1:10)
[1] 5.5

> mean(x = 99:999)
[1] 549
```

- In the above examples, the calling arguments are `1:10` and `99:999` respectively

Calling Arguments of a Function

- When calling a function you can specify arguments by position, by complete name, or by partial name
- Arguments are matched in the following order
  1. Exact name (perfect matching)
  2. Prefix matching (imperfect/partial matching)
  3. Position

```
myFunc_09 <- function(arg1, my_arg2, my_arg3){
  list(a = arg1, m1 = my_arg2, m2 = my_arg3)
}
```

Calling Arguments of a Function [CONT'D]

```
# positional
> str(myFunc_09(1, 2, 3))
List of 3
 $ a : num 1
 $ m1: num 2
 $ m2: num 3
```

```
# joint partial matching and positional
> str(myFunc_09(2, 3, a = 1))
List of 3
 $ a : num 1
 $ m1: num 2
 $ m2: num 3
```

```
# exact matching and positional
> str(myFunc_09(2, 3, arg1 = 1))
List of 3
 $ a : num 1
 $ m1: num 2
 $ m2: num 3
```

Best Practices for Calling Arguments

- You only want to use positional matching for the first one or two arguments of a function call, i.e., the most commonly used arguments
- Avoid using positional matching for infrequently used arguments
- If a function uses `...` (ellipsis), you can only specify arguments listed after the `...` with their full name, i.e., exact matching
- If you are writing code for a package to be published to CRAN, you are not permitted to use partial matching

Using a List of Arguments to Call a Function

- If you wish call a function with a list of arguments, use the following code

```
> funcArguments <- list(1:10, na.rm = TRUE)

> do.call(mean, funcArguments)
[1] 5.5

# equivalent to
> mean(1:10, na.rm = TRUE)
[1] 5.5
```

Default Arguments

```
# w/o default values
myFunc_10 <- function(a, b) {
  c(a, b)
}

> myFunc_10()
Error in myFunc_10() : argument "a" is missing, with no default

# with default values
myFunc_11 <- function(a = 1, b = 2) {
  c(a, b)
}

> myFunc_11()
[1] 1 2
```

Default Arguments

Function arguments in `R` can be defined in terms of other arguments

```
myFunc_12 <- function(a = 1, b = a * 2) {
  c(a, b)
}

> myFunc_12()
[1] 1 2

> myFunc_12(111)
[1] 111 222

> myFunc_12(99, 100)
[1]  99 100
```

Missing Arguments

Two common approaches to determine whether or not an argument was supplied to a function:

1) missing()

```
myFunc_13 <- function(arg1, arg2) {
  c(missing(arg1), missing(arg2))
}

> myFunc_13()
[1] TRUE TRUE

> myFunc_13(arg1 = 1)
[1] FALSE  TRUE

> myFunc_13(arg2 = 99)
[1]  TRUE FALSE
```

Missing Arguments

2) Set default argument values to `NULL` and subsequently test if the argument is supplied using `is.null()`

```
> myFunc_14 <- function(arg1 = NULL, arg2 = NULL) {
  c(is.null(arg1), is.null(arg2))
}

> myFunc_14()
[1] TRUE TRUE

> myFunc_14(arg1 = 1)
[1] FALSE  TRUE

> myFunc_14(arg2 = 99)
[1]  TRUE FALSE
```

Lazy Functional Evaluation of Calling Arguments

- R function arguments are only evalued when they are used
- If you want to ensure that an argument is evaluated you can use `force()`

```
myFunc_15 <- function(x){
   10
 }

 > myFunc_15()
 [1] 10

 > myFunc_15(thisIsNonsense)
 [1] 10

 > myFunc_15("nonsense")
 [1] 10
```

```
myFunc_16 <- function(x){
  force(x)
  10
}

> myFunc_16(thisIsNonsense)
Error in force(x) : object
  'thisIsNonsense' not found
```

Lazy Evaluation, Default & Missing Arguments

- Default arguments are evaluated inside the function
- If the expression depends on the current environment the results will differ depending on whether you use the default value or explicitly provide one

```
myFunc_17 <- function(a = ls()){
  z <- 10
  a
}

> myFunc_17()
[1] "a" "z"

> myFunc_17(ls())
[1] "i"          "j"          "myFunc_13"
[4] "myFunc_15" "myFunc_17"
```

Return Values

The last expression evaluated in a function becomes the return value

```
myFunc_18 <- function(xyz) {
  if (xyz < 10) {
    0
  } else {
    10
  }
}

> myFunc_18(5)
[1] 0

> myFunc_18(10)
[1] 10
```

To `return()` or not to `return()`

- The last expression evaluated in a function is the return value
- You can always wrap the final expression in `return()` if you choose
- Calling `return()` is an additional call and will add to the execution time of your function, albeit minuscule for a single call
- In simplistic functions, `R` programmers will typically omit `return()`
- In longer, more complicated functions, `return()` is often used to distinguish “leaves” of code
- In sum, for the purposes of this class, I require the use of `return()` to make the code more legible for any functions with “leaves” of code

To `return()` or not to `return()`

```
# simple function, does not require a return()

myFunc_15 <- function(x){
  10
}



# a more complex function benefits visually from having return()
#   but does not require return()

myFunc_18 <- function(xyz) {
  if (xyz < 10) {
    return(0)
  } else {
    return(10)
  }
}
```

Lab - on your own

1. Write a function that takes two arguments, `firstRow` and `lastRow`, and returns rows `firstRow` through `lastRow` of `iris`, and subsequently call the function with values firstRow = 1 and lastRow = 3, using both positional matching and exact matching
2. In the question above, what are the formal and calling arguments of the function?
3. Is this function self-contained? Why or why not?
4. Rewrite the above function to include a data frame `myDataFrame` as an additional argument, such that it returns rows `firstRow` through `lastRow` of `myDataFrame`
5. Rewrite the function to use default arguments `firstRow = 1` and `lastRow = 10`, and evaluate all 3 arguments at the beginning of the function using force

Part III: Writing Robust Code

Writing Robust R Code

Debugging

How to fix unanticipated problems

Condition Handling

How functions communicate problems and how actions can be taken based on those communications

Defensive Programming

How to avoid common problems before they occur

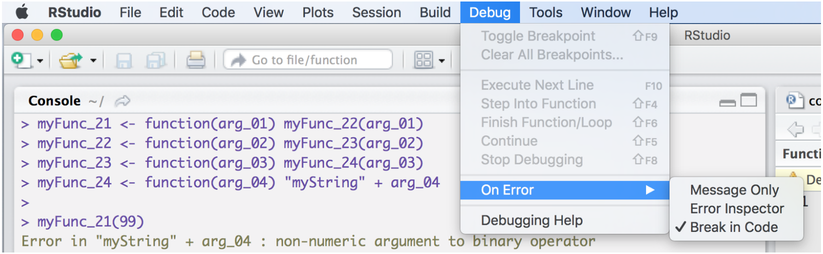
Debugging Tools

There are three key debugging tools

1. Error inspector and `traceback()` which lists a sequence of calls that lead to the error
2. "Return with Debug" tool and `options(error = browser)` which open an interactive session where the error occurred
3. Breakpoints and `browser()` which open an interactive session at an arbitrary location in the code

Depending on your selection in the menu bar, different actions will occur when R throws an error

- Selecting **Message Only** will simply print an error message to the console
- Selecting **Error Inspector** additionally provides links to *Show Traceback* and *Rerun with Debug*
- Selecting **Break in Code** additionally launches *Browse on Error*



Traceback & The Call Stack

- The call stack is the sequence of calls that lead up to an error
- For example, if we run the following code...

```
> rm(list=ls())

> myFunc_21 <- function(arg_01) myFunc_22(arg_01)
> myFunc_22 <- function(arg_02) myFunc_23(arg_02)
> myFunc_23 <- function(arg_03) myFunc_24(arg_03)
> myFunc_24 <- function(arg_04) "myString" + arg_04
```

- ... and then call `myFunc_21()`, we see the following error message

```
> myFunc_21(99)
Error in "myString" + arg_04 : non-numeric argument to binary operator
```

Traceback & The Call Stack

- The call stack is the sequence of calls that lead up to an error
- For example, if we run the following code...

```
> rm(list=ls())

> myFunc_21 <- function(arg_01) myFunc_22(arg_01)
> myFunc_22 <- function(arg_02) myFunc_23(arg_02)
> myFunc_23 <- function(arg_03) myFunc_24(arg_03)
> myFunc_24 <- function(arg_04) "myString" + arg_04
```

- ... and then call `myFunc_21()`, we see the following error message

```
> myFunc_21(99)
Error in "myString" + arg_04 : non-numeric argument to binary operator
```

Traceback & The Call Stack

- Looking at the Console pane, you should see the following

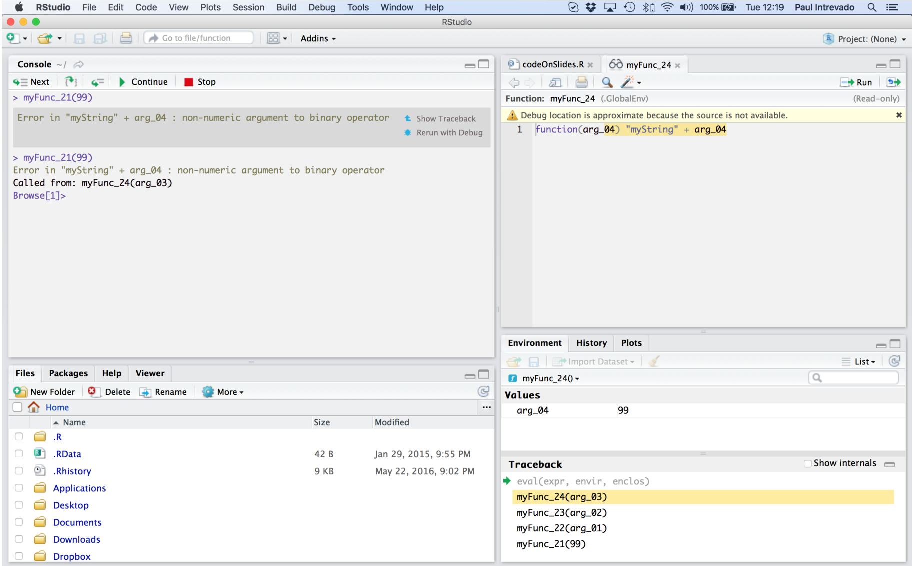
```
> myFunc_21(99)
Error in "myString" + arg_04 : non-numeric argument to binary operator      Hide Traceback
                                                                             Rerun with Debug

  4 myFunc_24(arg_03)
  3 myFunc_23(arg_02)
  2 myFunc_22(arg_01)
  1 myFunc_21(99)
```

- The call stack is to be read from bottom to top:
  - The initial call is to `myFunc_21()`
  - `myFunc_21()` calls `myFunc_22()`
  - `myFunc_22()` calls `myFunc_23()`
  - `myFunc_23()` calls `myFunc_24()` which triggers the error
- The Traceback window shows you where the error occurred, **not** why it occurred

Browsing on Error

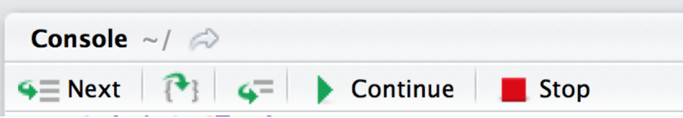
- Selecting *Rerun with Debug* allows you to enter the interactive debugger
- This reruns the command that create the error, pausing the execution where the error occurred
- This puts you in an interactive state inside the function, and you can interact with any objects defined there
- You will observe
  1. A Traceback pane with the call stack
  2. An Environment pane with all objects in the current environment
  3. A Code Browser pane (icon of glasses) listing the statement that will be run next highlighted in yellow
  4. A `Browse[1]>` prompt in the console window which allows you to run arbitrary code

Browsing on Error



Browsing on Error

A few special commands can be accessed in the toolbar on the Console pane (from left to right)



- `Next` executes the next step in the function
- `Step Into` works similarly to to `Next`, except if the next line is a function, it will also step into that function
- `Finish` completes execution of current loop or function
- `Continue` leaves interactive debugging and continues regular execution of the function
- `Stop` stops debugging, terminates function, and returns to the global workspace

Condition Handling

- The task of handling expected errors, e.g., when your function is expecting an atomic vector as an argument but is passed a data frame
- In `R`, there are two main tools for handling conditions (including errors) programatically
  1. `try()` gives you the ability to continue execution even when an error occurs
  2. `tryCatch()` lets you specify *handler* functions that control what happens when a condition is signaled

Wrapping code in the statement `try()` results in an error message printing **but** execution will continue

```
> rm(list=ls())

myFunc_25 <- function(z){
  log(z)
  print("Made it here")
}

> myFunc_25("abc")
Error in log(z) : non-numeric argument to mathematical function
```

Ignoring Errors with `try()`

```
myFunc_26 <- function(z){
try(log(z))
print("Made it here")
}

> myFunc_26("abc")
Error in log(z) : non-numeric argument to mathematical function
[1] "Made it here"
```

Ignoring Errors with `try()`

- If you prefer, you can suppress the error message with `try(..., silent = TRUE )`
- The output of `try()` can also be captured
  1. If the execution of code within `try()` is successful, the result will be the last result evaluated (just as in a function)
  2. If unsuccessful, the (invisible) result will be of class `try-error`

```
> successful <-  try(1 + 99)

> class(successful)
[1] "numeric"

> unsuccessful <- try("a" + "b")
Error in "a" + "b" : non-numeric argument to binary operator

> class(unsuccessful)
[1] "try-error"
```

Handling Conditions with `tryCatch()`

- `tryCatch()` is a general tool for handling conditions
- `tryCatch()` can handle
  1. errors (made by `stop()`)
  2. warnings (`warning()`)
  3. message (`message()`)
  4. interrupts (user-terminated code execution, e.g., `ctrl + C`)
- `tryCatch()` maps conditions to **handlers**, i.e., named functions that are called with the condition as an argument
- If a condition is signaled, `tryCatch()` will call the first handles whose name matches one of the classes of the condition

Handling Conditions with `tryCatch()`

```
show_condition <- function(code) {
  tryCatch(code,
           error = function(x) "myError",
           warning = function(x) "myWarning",
           message = function(x) "myMessage"
  )
}

> show_condition(stop("!"))
[1] "myError"

> show_condition(warning("?!"))
[1] "myWarning"

> show_condition(message("?"))
[1] "myMessage"
```

Handling Conditions with `tryCatch()`

Let's follow the execution of the function `show_condition()`

1. `show_condition(stop("!"))` calls the function `show_condition()`, passing `stop("!")` as the argument, represented in the function as `code`
2. `code` is executed in the `tryCatch()` block, where `code` == `stop("!")`
3. the function `stop()` "*stops execution of the current expression and executes an error action*"
4. when `stop()` executes an error action, `tryCatch()` maps the **error** condition to a function `error = function(x) "myError"`, which prints the word `myError` to the console
5. execution of the function terminates

Handling Conditions with `tryCatch()`

- When a condition in mapped to a function, what is being passed to that function?
- Let's modify the previous code and explore the inner workings of condition handling

```
show_condition <- function(code) {
  tryCatch(code,
           error = function(x) y <<- x
  )
}

> show_condition(stop("!"))
  ## this call generates no message in the console
```

- This is the first time we observe the «- operator, which makes an assignment to a *global* variable

Handling Conditions with `tryCatch()`

```
> y
<simpleError in doTryCatch(return(expr), name, parentenv, handler): !>

> str(y)
List of 2
 $ message: chr "!"
 $ call   : language doTryCatch(return(expr), name, parentenv, handler)
 - attr(*, "class")= chr [1:3] "simpleError" "error" "condition"

> attributes(y)
$names
[1] "message" "call"

$class
[1] "simpleError" "error"       "condition"

> y$message
[1] "!"
```

Handling Conditions with `tryCatch()`

- `tryCatch()` can be customized:

```
show_condition <- function(code) {
  tryCatch(code,
           error = function(x) {
             print(x$message)
             print(x$call)
             writeLines("\nSilly error!")
           }
  )
}

> show_condition(stop("!"))
[1] "!"
doTryCatch(return(expr), name, parentenv, handler)

Silly error!
```

Lab

Write a function employing error handling techniques that takes a single vector as input, take the natural log of each element in that vector, and print the result of each to the console

Defensive Programming

- Defensive programming is the art of making code fail in a well-defined manner even when something unexpected occurs
- A key principle of defensive programming is to *fail fast*: as soon as something wrong is discovered, signal an error
- This *fail fast* behavior is more work up front for the programmer, but results in easier debugging for the user, as they receive errors earlier rather than later, before the error has been potentially digested by multiple functions

Implementing the *Fail Fast* Principle

- Be strict about what a function accepts
  - If a function is not vectorized in inputs but uses functions that are, build in a check to ensure that inputs are scalars
  - Use `stopifnot()` or the `assertthat` package
- Avoid functions that use non-standard evaluation such as `subset`, `transform` and `with`
  - These functions save time when working with R interactively, but they typically fail uninformatively
  - Non-standard evaluation is the ability of a computing language to access not only the value(s) of a function's argument but also the code used to compute them (Advanced R, Chapter 13)

Implementing the *Fail Fast* Principle

- Be strict about what a function accepts
  - If a function is not vectorized in inputs but uses functions that are, build in a check to ensure that inputs are scalars
  - Use `stopifnot()` or the `assertthat` package
- Avoid functions that use non-standard evaluation such as `subset`, `transform` and `with`
  - These functions save time when working with `R` interactively, but they typically fail uninformatively
  - Non-standard evaluation is the ability of a computing language to access not only the value(s) of a function's argument but also the code used to compute them (Advanced `R`, Chapter 13)

Implementing the *Fail Fast* Principle

- Avoid functions that return different of output depending on their input
  - Two big offenders are `[` and `sapply()`
    - Whenever subsetting a data frame in a function, **always** use the option `drop = F` to maintain the data structure, e.g., to avoid converting a one-column data frame to an atomic vector

stop() versus stopifnot()

```
> myVec <- c("a", "bcd", "efgh")

> if(length(unique(nchar(myVec))) != 1) {
    stop("Error: Elements of your input vector do not have the
  same length!")
      }
Error: Error: Elements of your input vector do not have the same length!

> stopifnot(length(unique(nchar(myVec))) != 1,
    "Error: Elements of your input vector HAVE the same length!")
Error: "Error: Elements of your input vector HAVE the same length!"
is not TRUE

> stopifnot(1 == 1, all.equal(pi, 3.14159265), 1 < 2) # all TRUE

> stopifnot(1 == 2, all.equal(pi, 3.14159265), 1 < 2) # all first
#is FALSE
Error: 1 == 2 is not TRUE
```

Functional Programming

- Assume you are given the following data frame

```
> myDataFrame_01
   A  B C   D   E F
1  1  6 1   5 -99 1
2 10  4 4 -99   9 3
3  7  9 5   4   1 4
4  2  9 3   8   6 8
5  1 10 5   9   8 6
6  6  2 1   3   8 5
```

- Your objective is to replace all of the `-99`s with `NA`s

- You could—but shouldn't—iterate through each column manually, e.g.

```
myDataFrame_01$A[myDataFrame_01$A == -99] <- NA
myDataFrame_01$B[myDataFrame_01$B == -99] <- NA
...
myDataFrame_01$F[myDataFrame_01$F == -99] <- NA
```

Problems with Brute-Force Approaches

1. It's easy to make copy-paste mistakes
2. It makes bugs more likely
3. It makes updating code a HUGE pain in the arse
4. etc.

- Employ the Do Not Repeat Yourself (DRY) Principle

  "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [Thomas & Hunt, http://pragprog.com]

Functional Programming [EXAMPLE 1]

Let's write a function with the objective of replacing all `-99`s in a single column with `NA`s

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
}
```

- Will the code above work as intended? Hint: **no**. Why not?

Let's write a function with the objective of replacing all `-99`s in a single column with `NA`s

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
}
```

- Will the code above work as intended? Hint: **no**. Why not?

Functional Programming [EXAMPLE 1]

The following **does** work as intended:

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
  myCol
}

myDataFrame_01$A <- fix99s_byCol(myDataFrame_01$A)
...
myDataFrame_01$F <- fix99s_byCol(myDataFrame_01$F)
```

- This reduces but doesn't eliminate the potential for errors
- There is no gain in efficiency (repetitive code is still required)

Functional Programming [EXAMPLE 1 REVISITED]

- What if there were a function that could iterate not only across all rows of a column checking for `NAs`, but also across all columns of a data frame?
- `lapply()`—from the generic family of `apply()` functionals—takes three inputs
  1. A list
  2. A function (applied to each element of the list)
  3. `...` (other arguments to pass to the function)

- `lapply()` applies the function to each element of a list a returns the new list

n.b. We can employ `lapply()` here because data frames are lists

Definition `lapply()` returns a list of the same length as (the list) X, each element of which is the result of applying a function to the corresponding element of X.

- `lapply()` applies the function to each element of a list a returns the new list

n.b. We can employ `lapply()` here because data frames are lists

Definition `lapply()` returns a list of the same length as (the list) X, each element of which is the result of applying a function to the corresponding element of X.

Functional Programming [EXAMPLE 1 REVISITED]

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
  myCol
}


> myDataFrame_01 <- lapply(myDataFrame_01, fix99s_byCol)


> str(myDataFrame_01)
List of 6
 $ A: num [1:6] 1 10 7 2 1 6
 $ B: num [1:6] 6 4 9 9 10 2
 $ C: num [1:6] 1 4 5 3 5 1
 $ D: num [1:6] 5 NA 4 8 9 3
 $ E: num [1:6] NA 9 1 6 8 8
 $ F: num [1:6] 1 3 4 8 6 5
```

- This almost worked...but not quite

Functional Programming [EXAMPLE 1 REVISITED]

Here are two ways to correct the previous function call so that it returns a data frame

1. ```
   > myDataFrame_01 <-
   as.data.frame(lapply(myDataFrame_01,
   fix99s_byCol))
   ```
2. ```
   > myDataFrame_01[] <- lapply(myDataFrame_01,
   fix99s_byCol)
   ```

Functional Programming [EXAMPLE 1 REVISITED]

Employing functional programming, as in the previous example, has many advantages

1. It is very compact
2. If the code for a missing value changes, it only needs to be updated in a single location
3. It works for any number of columns, so you don't need to specify the number of columns, therefore avoiding potential mistakes
4. All columns are evaluated uniformly
5. You can generalize the technique to a subset of columns if preferred

```
> myDataFrame_01[1:3] <- lapply(myDataFrame_01[1:3], fix99s_byCol)
```

Adding Arguments

- What if different columns employed different coding schemes for missing values, e.g., `-99`, `-999` and `-8888888`?
- You could end up copy/pasting the function

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
  myCol
}
```

and replacing the `-99` in `myCol[myCol == -99] <- NA`, with a updated values in each copy/paste so that you end up with three different functions (but this is not efficient)

Adding Arguments

We can simply add an argument to the previous code as follows

```
fixMising <- function(myCol, myValue) {
  myCol[myCol == myValue] <- NA
  myCol
}
```

Anonymous Functions

- The following code is equivalent and permissible

```
myFunc_26 <- function(myCol) {
  myCol <- myCol + 3
  myCol
}

lapply(myDataFrame_01, myFunc_26)

#########################

lapply(myDataFrame_01, function(x) x + 3)
```

- What you are observing in the lower half of the code in an anonymous function, i.e., a function that does not have a name

Lab

- Create a compact and robust function which, when passed an $n \times m$ numeric data frame, returns, for each column, the
  1. Mean
  2. Median
  3. Standard Deviation
  4. Variance
  5. Quantiles
  6. IQR
- NOT THE BEST SOLUTION (but it works)

```
mySummaryFunc <- function(myCols) {
  c(mean(myCols), median(myCols), sd(myCols), var(myCols),
    quantile(myCols), IQR(myCols))
}

lapply(myDataFrame_01, mySummaryFunc)
```

Part IV: Functionals

Functionals

A functional is a function that takes a function as an input and returns a vector as an output

```
myFunctional_01 <- function(myFuncArg) myFuncArg(runif(1000), na.rm = TR
```

This works as follows

1. We create a functional named `myFunctional_01`
2. We pass the argument `myFuncArg` to `myFunctional_01`, **where the argument is itself a function**
3. The argument `myFuncArg` is then called using the parameters defined in the inline, anonymous function.

When we call the functional

```
myFunctional_01 <- function(myFuncArg) myFuncArg(runif(1000), na.rm = TR
```

we get the following results

```
> myFunctional_01(mean)
[1] 0.5194186

> myFunctional_01(mean)
[1] 0.5038302

> myFunctional_01(min)
[1] 0.000956069

> myFunctional_01(sd)
[1] 0.2846844
```

Why use Functionals?

- A common use of functionals is as an alternative to *for* loops
- *For* loops have a reputation for being slow in `R`
- The real advantage of using functionals is the ability to express a clear, specific objective in a single statement
- The probability of generating bugs in your code decreases

The `apply()` Family of Functionals

The `apply()` family of functionals are often used in lieu of *for* loops, coming in a variety of flavors (not exhaustive)

Functional	Input	Output
`apply()`	Array/Matrix	Vector/Array
`lapply()`	Vector/List	List
`sapply()`	Vector/List	List
`vapply()`	Vector/List	Vector

Let's examine a few examples to convince ourselves that the `apply()` family of functionals are truly useful

Lab

Using X

1. Write code that **does not contain** functionals that computes the mean of each column of data
2. Employ your functional of choice to write code that computes the mean of each column of data

Lab Solution

1. Write code that **does not contain** functionals that computes the mean of each column of data

```
> myColMeans_01 <- numeric(ncol(X))

for (i in 1:3) {
  myColMeans_01[i] <- mean(X[ , i])
}
```

2. Employ your functional of choice to write code that computes the mean of each column of data

```
> (myColMeans_02 <- apply(X, 2, mean))
```

The `apply()` function

- `apply()` coerces input to either a matrix (in 2 dimensions) or an array (in $>$ 2 dimensions), therefore the second argument indicates the dimension over which to apply the function
- The `apply()` function outputs a numeric vector

How `lapply()` Works

`lapply()` is a wrapper for a common loop pattern

- It creates a container for output
- Applies the function `f()` to each element of a list
- Fills the container with the results
- Returns a list
- Use `unlist()` to convert the list to a vector

Note `lapply()` is particularly useful for working with data frames as data frames are lists

`sapply()` and `vapply()`

- Both operate similarly to `lapply()`, taking similar inputs, but they differ on output
- `sapply()` will guess at what type of output it should generate
  - `sapply()` is good for interactive coding as it minimizes typing and the coder is able to observe and rectify and unexpected output types
  - **do not** bury an `sapply()` in a function where it can generate an odd and difficult to trace error

`sapply()` and `vapply()`

- `vapply()` requires an additional argument, specifying the output type
  - More verbose than `sapply()`, it always generates consistent output based on argument specification, gives more informative error messages, and never fails silently, and is therefore more appropriate for use inside functions

Final Notes on Functionals

- For multiple varying arguments, use `Map()`
- Leveraging the fact that each iterations of `apply()` functionals is isolated from all others, this lends themselves well to parallelisation using `mclapply()` and `mcMap()` from the `parallel` package
- May also want to read up on the `purrr` package

Lab

Using `state.x77`

1. Use `lapply()` to return the correlation of each numeric variable with population. What type of output is returned?
2. Repeat with `sapply()`. What type of output is returned?
3. Find the sum of area by region (*hint*: search for an `apply()` that we have not yet used)

Lab

Using `state.x77`

1. Use `lapply()` to return the correlation of each numeric variable with population. What type of output is returned?

```
c <- state.x77
lapplyResult <- lapply(3:8, function(i) return(cor(s[ ,2], s[ ,i])))
str(lapplyResult)
```

2. Repeat with `sapply()`. What type of output is returned?

```
sapplyResult = sapply(3:8, function(i) return(cor(s[, 2], s[, i])))
str(sapplyResult)
```

3. Find the sum of area by region (*hint*: search for an `apply()` that we have not yet used)

```
tapply(s[, "Area"], state.region, sum)
```

Part V: Efficiency of Code

How to Quantify Code Efficiency

- A precise way to measure the speed of small blocks of code is microbenchmarking
- `R` has a package called `microbenchmark` which provides a range of tools for evaluating code efficiency

```
> microbenchmark(sqrt(x), x^0.5, times = 1000)
Unit: microseconds
    expr    min      lq      mean median      uq     max neval
 sqrt(x)  3.959  4.2420  6.507298  6.607  7.3975  64.095  1000
   x^0.5 24.730 26.7105 32.004310 28.484 35.3140 110.297  1000
```

- By default, `neval = 100`

Some Context on Code Efficiency

It is useful to think about how many times a function needs to run before it takes one second

Microbenchmark	Interpretation
1 ms	1,000 calls takes one second
1 μs	1,000,000 calls takes one second
1 ns	1,000,000,000 calls takes one second

Practical Interpretation

It takes roughly 800 ns to compute the square root of 100 numbers using `sqrt()`. That means that if you repeated that operation a million times, i.e., compute the square root on 10,000,000 numbers, it would take 0.8 seconds.

system.time()

Wrapping code in `system.time()` will also give you the system time required to process code, but

1. `microbenchmark()` is far more precise
2. `system.time()` only runs the block of code once, therefore you need to mnually wrap `system.time()` in a loop to generate meaningful statistics

```
> microbenchmark(sqrt(x), x^0.5, times = 1000)
Unit: microseconds
    expr    min     lq      mean median      uq     max neval
 sqrt(x)  3.834  3.971  6.823777  4.213  7.7275 639.492  1000
   x^0.5 24.094 24.804 30.690782 26.232 31.9885 100.101  1000
>
> system.time(for (i in 1:1000) x^0.5) / 1000
   user  system elapsed
2.7e-05 1.0e-06 2.8e-05
```