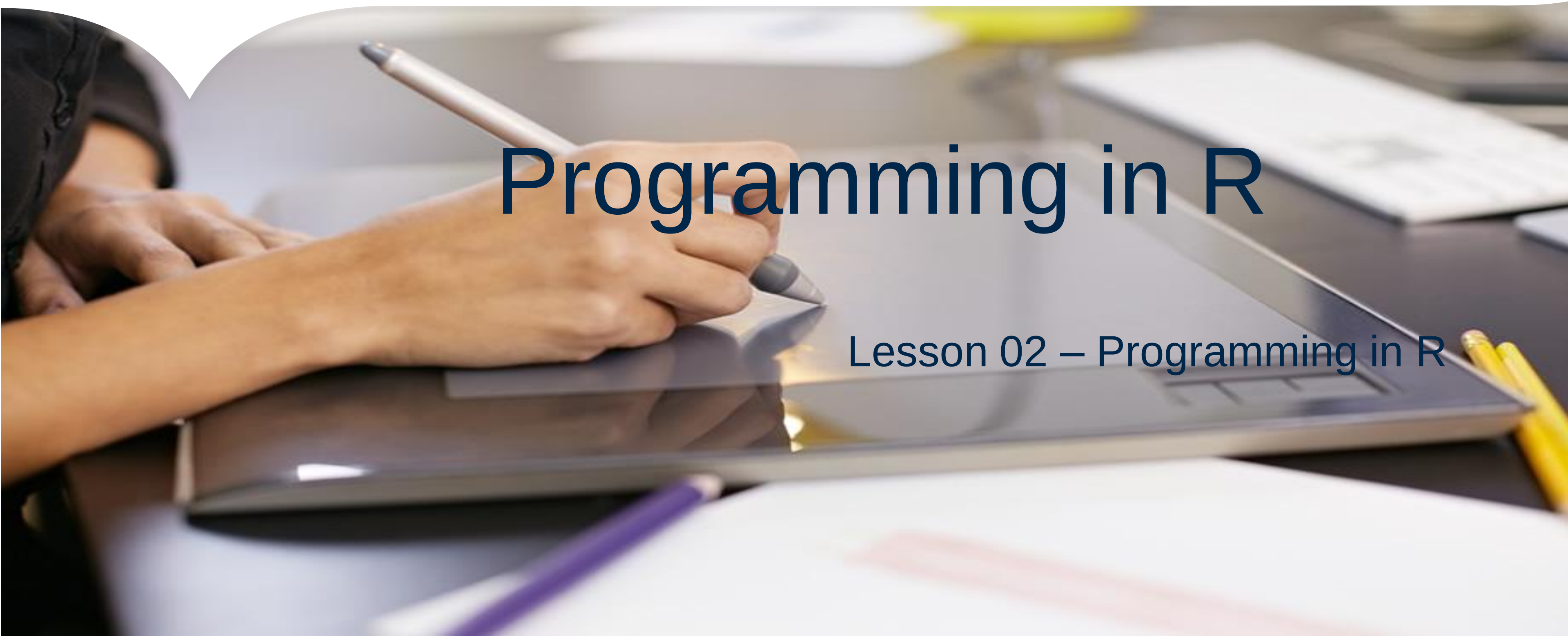

Programming in R

Lesson 02 – Programming in R

Lesson Objectives

- R Control Structures
  - if-else expression
  - for loop, a while loop, and a repeat loop
- Functions in R
  - function in R and specify its return value
  - How R binds a value to a symbol via the search list
  - Lexical scoping w.r.t. resolution of free variables in R
  - Lexical scoping scoping rules
  - Convert character string representing a date/time into an R datetime object

Copyright © Capgemini 2016. All Rights Reserved

R Control Structures

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are

- `if`, `else`: testing a condition
- `for`: execute a loop a fixed number of times
- `while`: execute a loop *while* a condition is true
- `repeat`: execute an infinite loop
- `break`: break the execution of a loop
- `next`: skip an interation of a loop
- `return`: exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expresisons.

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2016. All Rights Reserved

Functions in R

Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```
f <- function(<arguments>) {
        ## Do something interesting
}
```

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated.

Copyright © Capgemini 2016. All Rights Reserved

Functions in R : Function Arguments

Functions have *named arguments* which potentially have *default values*.

- The *formal arguments* are the arguments included in the function definition
- The `formals` function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be *missing* or might have default values

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

Copyright © Capgemini 2016. All Rights Reserved

Functions in R : Argument Matching

R functions arguments can be matched positionally or by name. So the following calls to `sd` are all equivalent

```
> mydata <- rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2016. All Rights Reserved

Functions in R : Argument Matching

You can mix positional matching with matching by name. When an argument is matched by name, it is “taken out” of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> args(lm)
function (formula, data, subset, weights, na.action,
          method = "qr", model = TRUE, x = FALSE,
          y = FALSE, qr = TRUE, singular.ok = TRUE,
          contrasts = NULL, offset, ...)
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Copyright © Capgemini 2016. All Rights Reserved

Functions in R : Argument Matching

Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list

Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

Copyright © Capgemini 2016. All Rights Reserved

Functions in R : Argument Matching

Function arguments can also be *partially* matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Copyright © Capgemini 2016. All Rights Reserved

Functions in R: Defining a Function

```r
f <- function(a, b = 1, c = 2, d = NULL) {

}
```

In addition to not specifying a default value, you can also set an argument value to `NULL`.

Copyright © Capgemini 2016. All Rights Reserved

Functions in R : Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed.

```
f <- function(a, b) {
    a^2
}
f(2)
```

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

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the 2 gets positionally matched to `a`.

Copyright © Capgemini 2016. All Rights Reserved

Functions in R : Lazy Evaluation

```
f <- function(a, b) {
    print(a)
    print(b)
}
f(45)
```

```
## [1] 45
```

```
## Error: argument "b" is missing, with no default
```

Notice that “45” got printed first before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` it had to throw an error.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Copyright © Capgemini 2016. All Rights Reserved

Functions in R : The "..." Argument

The ... argument indicate a variable number of arguments that are usually passed on to other functions.

- ... is often used when extending another function and you don't want to copy the entire argument list of the original function

```
myplot <- function(x, y, type = "l", ...) {
        plot(x, y, type = type, ...)
}
```

- Generic functions use ... so that extra arguments can be passed to methods (more on this later).

```
> mean
function (x, ...)
UseMethod("mean")
```

Copyright © Capgemini 2016. All Rights Reserved

Functions in R : The "..." Argument

The ... argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> args(cat)
function (..., file = "", sep = " ", fill = FALSE,
    labels = NULL, append = FALSE)
```


Copyright © Capgemini 2016. All Rights Reserved

Functions in R : Arguments following "..."

One catch with ... is that any arguments that appear *after* ... on the argument list must be named explicitly and cannot be partially matched.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> paste("a", "b", sep = ":")
[1] "a:b"

> paste("a", "b", se = ":")
[1] "a b :"
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Copyright © Capgemini 2016. All Rights Reserved

R Scoping Rules : Binding Values to Symbol

How does R know which value to assign to which symbol? When I type

```
> lm <- function(x) { x * x }
> lm
function(x) { x * x }
```

how does R know what value to assign to the symbol `lm`? Why doesn't it give it the value of `lm` that is in the *stats* package?

Copyright © Capgemini 2016. All Rights Reserved

R Scoping Rules : Binding Values to Symbol

When R tries to bind a value to a symbol, it searches through a series of `environments` to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly

1. Search the global environment for a symbol name matching the one requested.
2. Search the namespaces of each of the packages on the search list

The search list can be found by using the `search` function.

```
> search()
[1] ".GlobalEnv"         "package:stats"      "package:graphics"
[4] "package:grDevices"  "package:utils"      "package:datasets"
[7] "package:methods"    "Autoloads"          "package:base"
```


Copyright © Capgemini 2016. All Rights Reserved

R Scoping Rules : Binding Values to Symbol

- The *global environment* or the user's workspace is always the first element of the search list and the *base* package is always the last.
- The order of the packages on the search list matters!
- User's can configure which packages get loaded on startup so you cannot assume that there will be a set list of packages available.
- When a user loads a package with `library` the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.
- Note that R has separate namespaces for functions and non-functions so it's possible to have an object named c and a function named c.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2016. All Rights Reserved

R Scoping Rules

The scoping rules for R are the main feature that make it different from the original S language.

- The scoping rules determine how a value is associated with a free variable in a function
- R uses *lexical scoping* or *static scoping*. A common alternative is *dynamic scoping*.
- Related to the scoping rules is how R uses the search *list* to bind a value to a symbol
- Lexical scoping turns out to be particularly useful for simplifying statistical computations

Copyright © Capgemini 2016. All Rights Reserved

R Scoping Rules : Lexical Scoping

Consider the following function.

```
f <- function(x, y) {
        x^2 + y / z
}
```

This function has 2 formal arguments `x` and `y`. In the body of the function there is another symbol `z`. In this case `z` is called a *free variable*. The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned insided the function body).

Copyright © Capgemini 2016. All Rights Reserved

R Scoping Rules : Lexical Scoping

Consider the following function.

```
f <- function(x, y) {
        x^2 + y / z
}
```

This function has 2 formal arguments `x` and `y`. In the body of the function there is another symbol `z`. In this case `z` is called a *free variable*. The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned insided the function body).

Copyright © Capgemini 2016. All Rights Reserved

R Scoping Rules : Lexical Scoping

Lexical scoping in R means that

the values of free variables are searched for in the environment in which the function was defined.

What is an environment?

- An *environment* is a collection of (symbol, value) pairs, i.e. `x` is a symbol and `3.14` might be its value.
- Every environment has a parent environment; it is possible for an environment to have multiple "children"
- the only environment without a parent is the empty environment
- A function + an environment = a *closure* or *function closure*.

Copyright © Capgemini 2016. All Rights Reserved

R Scoping Rules : Lexical Scoping

Searching for the value for a free variable:

- If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the *parent environment*.
- The search continues down the sequence of parent environments until we hit the *top-level environment*; this usually the global environment (workspace) or the namespace of a package.
- After the top-level environment, the search continues down the search list until we hit the *empty environment*. If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.

Copyright © Capgemini 2016. All Rights Reserved

Dates and Times in R

R has developed a special representation of dates and times

- Dates are represented by the `Date` class
- Times are represented by the `POSIXct` or the `POSIXlt` class
- Dates are stored internally as the number of days since 1970-01-01
- Tmes are stored internally as the number of seconds since 1970-01-01

Copyright © Capgemini 2016. All Rights Reserved

Dates in R

Dates are represented by the Date class and can be coerced from a character string using the `as.Date()` function.

```
x <- as.Date("1970-01-01")
x
## [1] "1970-01-01"
unclass(x)
## [1] 0
unclass(as.Date("1970-01-02"))
## [1] 1
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Copyright © Capgemini 2016. All Rights Reserved

Times in R

Times are represented using the `POSIXct` or the `POSIXlt` class

- `POSIXct` is just a very large integer under the hood; it use a useful class when you want to store times in something like a data frame
- `POSIXlt` is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month

There are a number of generic functions that work on dates and times

- `weekdays`: give the day of the week
- `months`: give the month name
- `quarters`: give the quarter number (“Q1”, “Q2”, “Q3”, or “Q4”)

Copyright © Capgemini 2016. All Rights Reserved

Times in R

Times can be coerced from a character string using the `as.POSIXlt` or `as.POSIXct` function.

```
x <- Sys.time()
x
## [1] "2013-01-24 22:04:14 EST"
p <- as.POSIXlt(x)
names(unclass(p))
## [1] "sec"    "min"    "hour"   "mday"   "mon"
## [6] "year"   "wday"   "yday"   "isdst"
p$sec
## [1] 14.34
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2016. All Rights Reserved

Times in R

You can also use the `POSIXct` format.

```
x <- Sys.time()
x  ## Already in 'POSIXct' format
## [1] "2013-01-24 22:04:14 EST"
unclass(x)
## [1] 1359083054
x$sec
## Error: $ operator is invalid for atomic vectors
p <- as.POSIXlt(x)
p$sec
## [1] 14.37
```

Copyright © Capgemini 2016. All Rights Reserved

Times in R

Finally, there is the `strptime` function in case your dates are written in a different format

```
datestring <- c("January 10, 2012 10:40", "December 9, 2011 9:10")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x
```

```
## [1] "2012-01-10 10:40:00 EST" "2011-12-09 09:10:00 EST"
```

```
class(x)
```

```
## [1] "POSIXlt" "POSIXt"
```

I can *never* remember the formatting strings. Check `?strptime` for details.

Copyright © Capgemini 2016. All Rights Reserved

Operations on Dates and Times

You can use mathematical operations on dates and times. Well, really just + and -. You can do comparisons too (i.e. ==, <=)

```
x <- as.Date("2012-01-01")
y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
x-y
## Warning: Incompatible methods ("-.Date",
## "-.POSIXt") for "-"
## Error: non-numeric argument to binary operator
x <- as.POSIXlt(x)
x-y
## Time difference of 356.3 days
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2016. All Rights Reserved

Operations on Dates and Times

Even keeps track of leap years, leap seconds, daylight savings, and time zones.

```
x <- as.Date("2012-03-01") y <- as.Date("2012-02-28")
x-y
## Time difference of 2 days
x <- as.POSIXct("2012-10-25 01:00:00")
y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
y-x
## Time difference of 1 hours
```


Copyright © Capgemini 2016. All Rights Reserved

Dates and Times in R : Summary

- Dates and times have special classes in R that allow for numerical and statistical calculations
- Dates use the `Date` class
- Times use the `POSIXct` and `POSIXlt` class
- Character strings can be coerced to Date/Time classes using the `strptime` function or the `as.Date`, `as.POSIXlt`, or `as.POSIXct`

Copyright © Capgemini 2016. All Rights Reserved

Coding Standards in R

- Always use text files / text editor
- Indent your code
- Limit the width of your code (80 columns?)
- Limit the length of individual functions

Copyright © Capgemini 2016. All Rights Reserved

Lesson Summary

- R Control Structures
  - if-else expression
  - for loop, a while loop, and a repeat loop
- Functions in R
  - function in R and specify its return value
  - How R binds a value to a symbol via the search list
  - Lexical scoping w.r.t. resolution of free variables in R
  - Lexical scoping rules
  - Convert character string representing a date/time into an R datetime object

Copyright © Capgemini 2016. All Rights Reserved