# Week 2: Programming with R

Katherine Ramírez Cubillos

20/8/2020

## R Programming

This document contains all scripts developed in the second week of the R-Programming course of the John Hopkings University, teaching by Rogger D.Peng.

## Control Structures

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

- **If, else**: testing a condition
- **for**: execute a loop a fixed number of time
- **while**: execute a loop while a condition is true.
- **repeat**: execute an infinite 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 expresions.

## Control Structures- If-else

It allows you to test logical conditions, and let the r program do something, give whether or not, depending on whether that conditions is true or false.

```
# if(condition) {
  ## do something
# } else {
  ## do something else
# }

# if(condition1) {
  ## do something
# } else if(condition2) {
  ## do something different
# } else {
  ## do something different
# }
```

```r
## Tis is a valid id/else structure
x<- 2
y<-0
if(x>3){
  y<-10
}else{
  y<-0
}

## So is this one

y<-if(x>3){
  10
}else{
  0
}
```

Of course, the else clause is not necessary

```r
# if(<condition1>) {
# }
# if(<condition2>) {
# }
```

## Control Structures- For

**for** loops take an interator variable and assing it successive values from a sequence or vector. For loops are most commonly used for iterating over the elemens of an object (list, vector, etc.)

```r
for(i in 1:10){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

This loop takes the **i** variable and each iteration of the loop gives it values 1,2,3,. . . ,10, and then exits.

These three loops have the same behavior

```r
x<-c("a", "b", "c", "d")

for(i in 1:4){
  print(x[i])
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

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

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

```
for(letter in x){
  print(letter)
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

```
for(i in 1:4) print(x[i])
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

**seq_along** it takes a vector as an input and it creates an integer sequence that's equal to the length of that vector.

**for** loops can be nested

**seq_len** it takes an integer, which in this case happens to be the number of rows in x, and then it creates an integer sequence out of that. Do in this particular matrix has two rows, so it's going to create the sequence of 1 to 2.

```
x<-matrix(1:6,2,3)

for(i in seq_len(nrow(x))){
  for(j in seq_len(ncol(x))){
    print(x[i,j])
  }
}
```

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

Be careful with nesting through. Nesting beyond 2-3 levels is often very difficult to read/understand.

## Control Structures- While loops

While loops being by testing a condition. If it is true, then they execute the loop body. Once the loop body is execute, the condition is tested again, and so forth.

```
## Ths is an infinitive loop.

# count<-0
# while(count<-10){
#   print(count)
#   count<-count+1
# }
```

While loop can potentially result in infinite loops if not written properly. Use with care!

Sometimes there will be more than one condition in test.

```
z <- 5
while(z >= 3 && z <= 10) {
  print(z)
  coin <- rbinom(1, 1, 0.5)
  if(coin == 1) { ## random walk
    z <- z + 1
  } else {
    z <- z - 1
  }
}
```

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

Conditions are always evaluated from left to right.

## Control Structures- Repeat

Repeat initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a **repeat** loop is to call **break**

```
# x0<-1
# tol<-1e-8

# repeat{
#   x1<-computeEstimate()

#   if (abs(x1-x0)<tol){
#     break
#   }else{
#     x0<-x1
#   }
# }
```

The loop in the previous slide is a bit dangerous because there are no guarantee it will sto. Better to set a hard limit on the number of interation (e.g. using a for loop) and then report whether convergence was achived or not.

**Next, return**

**next** is used to skip an iteration of a loop

```
for(i in 1:100){
  if(i<=20){
    ## Skip the firts 20 iterations
    next
  }
  ## Do someting here
}
```

**return** signal that a function should exit and return a given value.

## Control Structures Summary

- Control structures like **if, while,** and **for** allow you to control the flow of R program.
- Infinite loops should generally be avoided, even if the are theoretically correct.
- Control structures mentiond here are primarily useful for writing programs; for command-line interactive work, the *apply functions are more useful.

## My First R Funciton

```
add2 <-function(x,y){
  x+y
}

add2(3,5)
```

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

This function is going to take a vector of numbers, it's going to return the subset of the vector, that's above the vector value ten. So any number that's bigger then ten, it's going to return those numbers for you.

```
above10<-function(x){
  use<-x>10
  x[use]
}

x<-1:20
above10(x)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

This function allows people to sub, to kind of extract the elements of a vector.

```
above <-function(x,n){
  use <-x>n
  x[use]
}
above(x,12)
```

```
## [1] 13 14 15 16 17 18 19 20
```

This is the same functions, but with a default value n=10.

```
above <-function(x,n=10){
  use <-x>n
  x[use]
}
above(x)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

This funciton is going to take a matrix or a data frame and calculate the mean of each column. This is going to involve using a for-loop.

```
columnmean <- function(y){
  nc <- ncol(y)
  means <- numeric(nc)
  for(i in 1:nc){
    means[i] <- mean(y[, i])
  }
  means
}

columnmean(airquality)
```

```
## [1]        NA        NA  9.957516 77.882353  6.993464 15.803922
```

This is the same function but remove the NA values.

```
columnmean <- function(y, removeNA=TRUE){
  nc <- ncol(y)
  means <- numeric(nc)
  for(i in 1:nc){
    means[i] <- mean(y[, i], na.rm = removeNA)
  }
  means
}

columnmean(airquality)
```

```
## [1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

## Functions

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

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

Functions in R are "first calss 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 funciton inside of another funcitons.
- The return value of a function is the last expression in the fucniton body to be evaluated.

### Functions Arguments

Functions have named arguments which potentialy have default values.

- The formal arguments are the arguments included in the function definition.
- The **formal** function returns a list of all formal arguments of a function
- No every function call in R makes use of all the formal arguments.
- Function arguments can be *missing* or might have default values.

### Arguments Matching

R functions arguments can be matches positionally or by name. So the following to **sd** are all equivalent.

```
mydata<-rnorm(100)
sd(mydata)
```

```
## [1] 1.096707
```

```
sd(x=mydata)
```

```
## [1] 1.096707
```

```
sd(x=mydata, na.rm=FALSE)
```

```
## [1] 1.096707
```

```
sd(na.rm=FALSE, x=mydata)
```

```
## [1] 1.096707
```

```r
sd(na.rm=FALSE, mydata)
```

```
## [1] 1.096707
```

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.

You can mix positional matching with marching 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.

```r
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, ...)
## NULL
```

The following lm calls are equivalent

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

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

Named arguments also help if you can remember the name of the arguments and no its position on the argument list (plotting is a good example).

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

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

### 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.**

### 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 matches to **a**.

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

f(45,35)
```

```
## [1] 45
## [1] 35
```

If we don't give a value to b. We would get:

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

This is because **b** didn't have to be evaluated until after **print(a)**. Once the function tried to evaluate **print(b)**, it had to throw an error.

### The "..." argument

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

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

```
myplot<-function(x,y, type="1", ...){
  plot(x,y, type,...)
}
```

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

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x000000000091ca4f0>
## <environment: namespace:base>
```

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)
## NULL
```

```
args(cat)
```

```
## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## NULL
```

One chatch with ... is that any arguments that appear *after* ... on the argument list must be named explicity and cannont be partially matched.

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL)
## NULL
```

```
paste("a", "b", sep=":")
```

```
## [1] "a:b"
```

```
paste("a", "b", se=":")
```

```
## [1] "a b :"
```

## Scoping Rules

### A diversion on binding values to symbol

How does know which value to asign 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 the value of **lm** that is in the *stats* package?

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

1. Seach the global environmental for a symbol name the one requested.
2. Seach the namespaces of each of the packages on the search list.

The search list can be found by using the **search** funciton.

```r
search()
```

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

- 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 matter!
- User's can configure which packages get loaded on starup so you cannot assume that will be a set list of packages available.
- When a user loads a packages with **library** the namespaces of that packages gets put in position 2 of the search list (by default) and everything else get 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 name c.

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 *statical scoping*. A common alternative is *dynamic scoping*.
- Lexical scoping turns out to be particulary useful for simplifying statistical computations.

Consider the following function:

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

z=3
f(4,8)
```

```
## [1] 18.66667
```

This function has 2 formal arguments **x** y **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 derermine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned inside the function body).

### Lexical scoping

Lexical scoping in R means that: *The values of free variables are searched for in the environmet in which the function was defined*

### What is an environment?

- An environment is a collection of (sybol, value) pairs, ie., **x** is a symbol and **3,14** might be it's value.
- Every environment has a parent environment; it is possible for an environment to have multiple "children".
- The only environment whitout a parent is the empty environment.
- A function + an environment= a *closure or funciton closure*.

Searching for the value for a free variable:

- If the value of the 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 seacrh list until we hit the *empty envrionment*. If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.

**Why does all ths is matter?**

- Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace.
- This behavior is logical for most people and is usually the "right thing" to do.
- However, in R you can have functions defined *inside other functions*
- Languages like C don't let you do this.
- Now things get interesting - in this case the environment in which a funciton is defined is the body of another function!

```
make.power<-function(n){
  pow<-function(x){
    x^n
  }
  pow
}
```

This function returns another function as its value

```
cube<-make.power(3)
square<-make.power(2)
cube(3)
```

```
## [1] 27
```

```
square(2)
```

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

What's in a funciton's environment?

```
ls(environment(cube))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(cube))
```

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

```
ls(environment(square))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(square))
```

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

### Lexical VS.Dynamic Scoping

```
y<-10
f<-function(x){
  y<-2
  y^2 + g(x)
}

g<-function(x){
  x*y
}
```

What is the value of **f(3)**

```
f(3)
```

```
## [1] 34
```

With lexical scoping the value of **y** in the funciton **g** is looked up in the environment in which the function was defined, in this case the global environment, so the value of **y** is 10.

With dynamic scoping, the value of **y** is looked up in the environment from which the function was *called* (sometimes referred to as the *calling environment*)

- In R the calling environment is known as the *parent frame*.

So the value of **y** would be 2.

When a function is *defined* in the gloal environment and is subsequently *called* from the global environmentm then the defining environment and the calling environment are the same. This can sometimes give the apperance of dynamic scoping.

```
g <- function(x){
  a <- 3
  x+a+y
}
## g(2)
## Erroe in g(2): object "y" not found
y<-3
g(2)
```

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

### Other Languages

Other languages that support lexical scopping:

- Scheme
- Perl
- Python
- Common Lisp (all languages coverge to Lisp)

### Consequences of Lexical Scoping

- In R, all objects must be stored in memory.
- All functions must carry a pointer to their respective defining environments, which could be anywhere.
- In S-PLUS, free variables are always looked up in the global workspace, so everything can be store on the disk because the "defining environment" of all functions is the same.

## Application: Optimization

Why is any of this information useful?

- Optimization routines in R like **optim, lm**, and **optimize** require you to pass a function whose argument is a vector of parameters (e.g. a log-likelihood).
- However, an object function might depend on a host of other things besides it parameters (*like data*)
- When writing software which does optimization, it may be desariable to allow the user to hold certain parameters fixed.

## Maximizing a Normal Likehood

Write a "constructor" function

```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {
 params <- fixed
 function(p) {
 params[!fixed] <- p
 mu <- params[1]
 sigma <- params[2]
 a <- -0.5*length(data)*log(2*pi*sigma^2)
 b <- -0.5*sum((data-mu)^2) / (sigma^2)
 -(a + b)
 }
}
```

Note: Opotimization function in R *minimize* functions, so you need to use the negative log-likelihood.

```
set.seed(1); normals <- rnorm(100, 1, 2)
nLL <- make.NegLogLik(normals)
nLL
```

```
## function(p) {
##  params[!fixed] <- p
```

```
##  mu <- params[1]
##  sigma <- params[2]
##  a <- -0.5*length(data)*log(2*pi*sigma^2)
##  b <- -0.5*sum((data-mu)^2) / (sigma^2)
##  -(a + b)
##  }
## <bytecode: 0x0000000008c50018>
## <environment: 0x00000000094f5a80>
```

```
ls(environment(nLL))
```

```
## [1] "data"   "fixed"  "params"
```

### Estimating Parameters

```
optim(c(mu = 0, sigma = 1), nLL)$par
```

```
##       mu    sigma
## 1.218239 1.787343
```

Fixing $\omega = 2$

```
nLL <- make.NegLogLik(normals, c(FALSE,2))
optimize(nLL, c(-1,3))$minimum
```

```
## [1] 1.217775
```

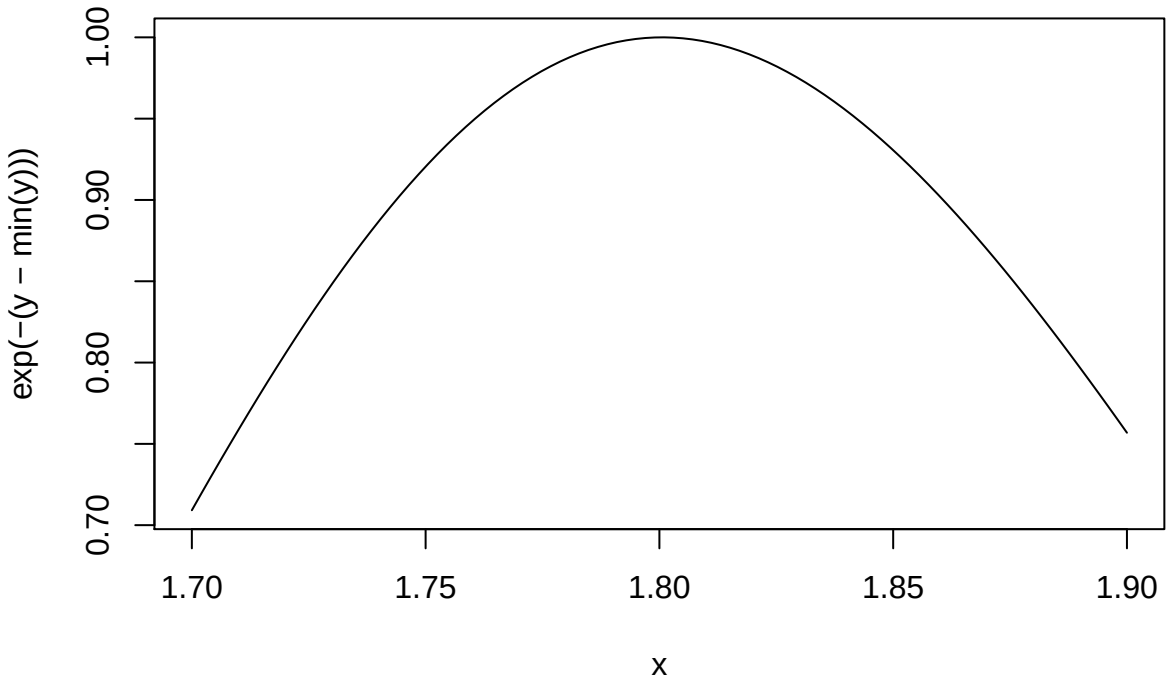
Fixing $\mu = 1$

```
nLL <- make.NegLogLik(normals, c(1, FALSE))
optimize(nLL, c(1e-6, 10))$ninimum
```

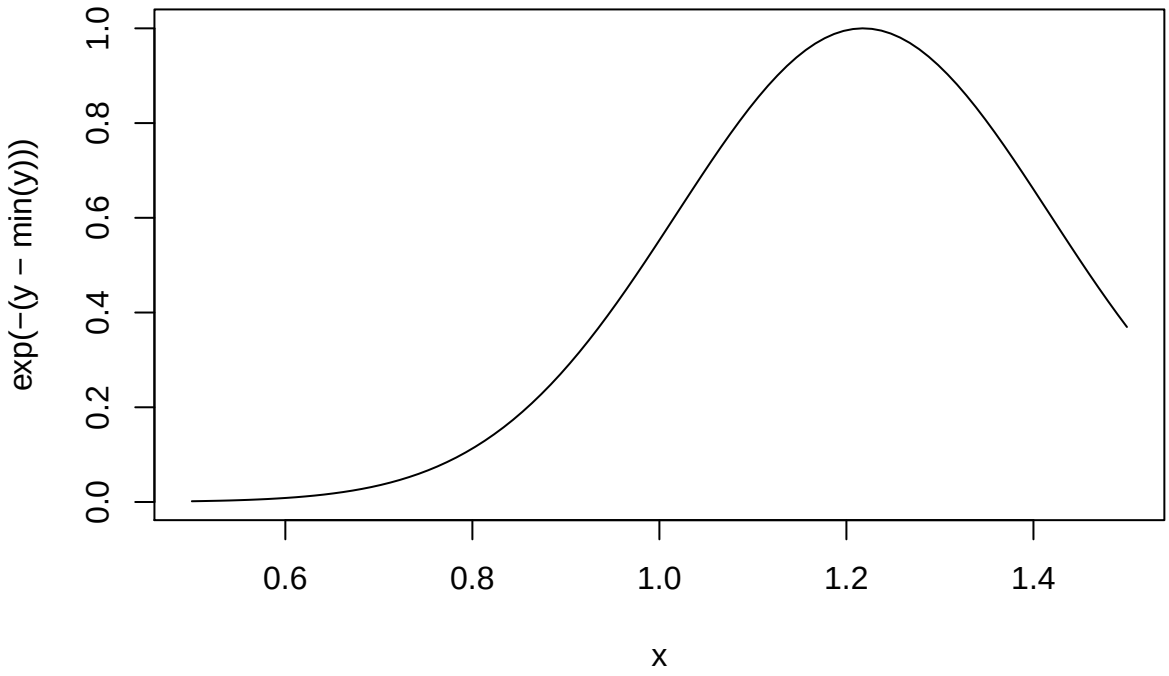
```
## NULL
```

### Plotting the Likelihood

```
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")
```




```
nLL <- make.NegLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")
```




## Lexical Scoping Summary

- Objective funcitons can be "built" which contain all of the necessary data for evaluating the function.
- No need to carry arround long argument lists - useful for interactive and exploratory work.
- Code can be simplifed and cleand up.
- Reference: Robert Gentleman and Ross Ihaka (2000). "Lexical Scope and Statistical Computing," JCGS, 9, 491–508.

## Coding Standards for R

**1. Always use text files / text editor:** Text file is a kind of basic standard. The basic idea is that a text format, can be read by pretty much any basic aditing program. When you're writing code you should always try to use a text editor, becasue it makes it that every will be able you acces your code and improve upon it.

**2. Ident your code:** Identing is the idea that different blocks of code should be space over the right a little bit more than other blocks of code so you can see kind of how the control flow, how the low of the program goes based on the indenting alone.

**3. Limiti the width of your code (80 columns?)**

**4. Limit the lenght of individual functions**

# Dates and Times in R

R has develop 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
- Times are stored internally as the number of seconds since 1970-01-01.

## Dates in R

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

```
x <- as.Date("1970-01-01")
x
```

```
## [1] "1970-01-01"
```

```
unclass(x)
```

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

```
unclass(as.Date("1970-01-02"))
```

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

## Times in R

Times are represented using 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 week, day of 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").

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

```
x <- Sys.time()
x
```

```
## [1] "2020-08-24 21:39:50 -05"
```

```
p <- as.POSIXlt(x)
names(unclass(p))
```

```
##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"
##  [9] "isdst"  "zone"   "gmtoff"
```

```
p$sec
```

```
## [1] 50.66311
```

You can also use the **POSIXct** format

```
x<- Sys.time()
x  ## Alreaady in 'POSIXct' format
```

```
## [1] "2020-08-24 21:39:50 -05"
```

```
unclass(x)
```

```
## [1] 1598323191
```

```
# x$sec
# Error: $ operator is invalid for atomic
p$sec
```

```
## [1] 50.66311
```

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] NA NA
```

```
class(x)
```

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

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

## Operation on Dates and Times

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

```r
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 NA secs
```

```r
## Time difference of 356.3 days
```

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

```r
x <- as.Date("2012-03-01")
y <- as.Date("2012-02-28")
x-y
```

```
## Time difference of 2 days
```

```r
## 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 0 secs
```

```r
## Time difference of 1 hours
```

## Summary

- Dates and times have special classes in R that allow for numerical nad statistical calculations.
- Dates use the **Date** class
- Time 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**