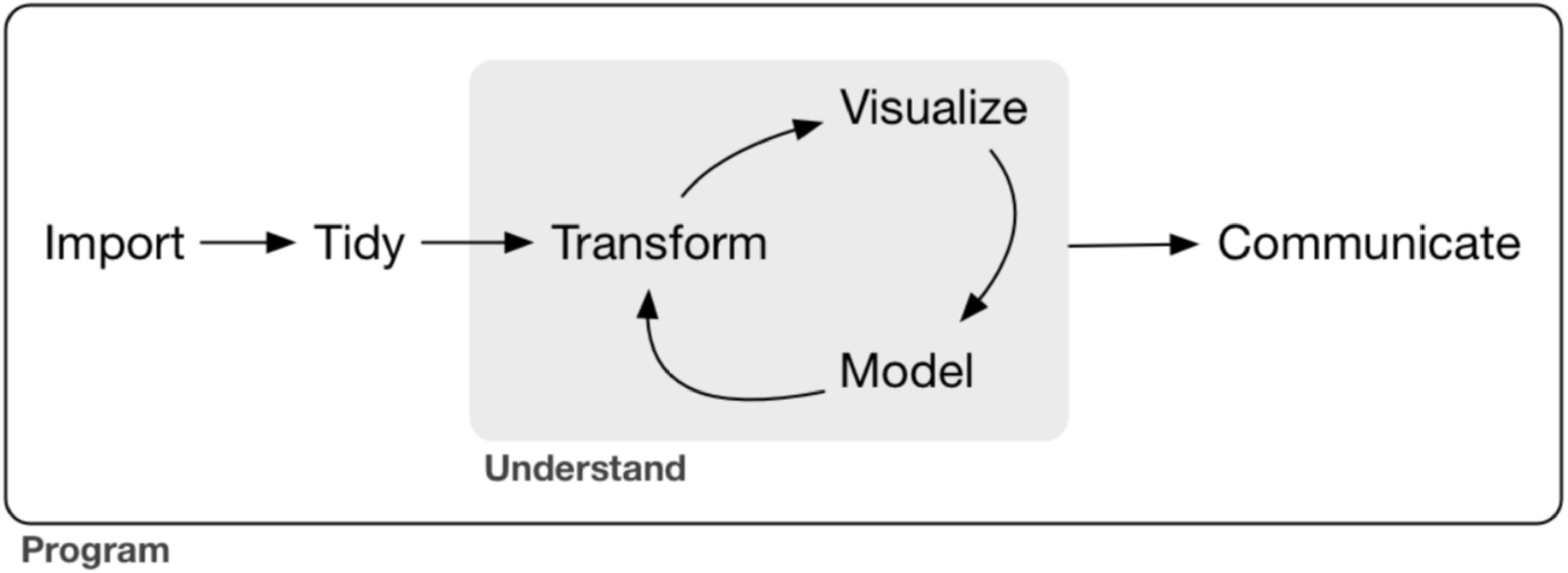
Programming in R

Elmer C. Peramo

Typical Data Processing Workflow in R



Demo

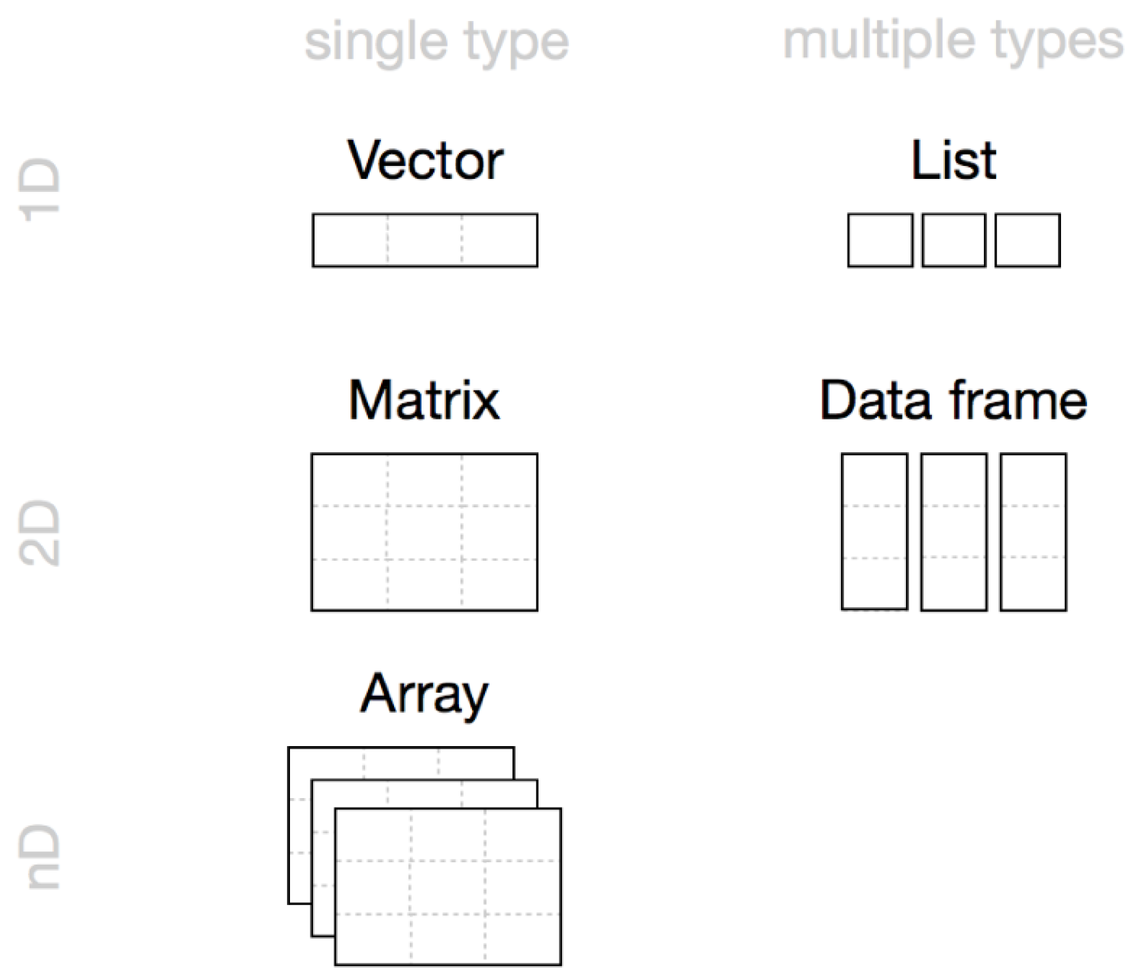
R Objects and Data Types

- R has 5 basic or "atomic" classes of objects:
  - Character
  - Numeric (Real Numbers)
  - Integer
  - Complex
  - Boolean

- R has 4 data types:
  - Vector
  - List
  - Matrix
  - Data Frame

A **vector** can only contain objects of the same class.
A **list** can represent different classes.
A **matrix** is a two-dimensional vector.
A **data frame** is a two-dimensional list.

R Objects and Data Types



Naming R Objects

- You can name an object in R almost anything you want, but there are a few rules. First, a name cannot start with a number. Second, a name cannot use some special symbols, like ^, !, $, @, +, -, /, or *:

Good names	Names that cause errors
a	1trial
b	$
FOO	^mean
my_var	2nd
.day	!bad

Numbers

- Numbers in R are generally treated as numeric objects (i.e., double precision real numbers). Append the L suffix if you want an integer. For example, entering **`3`** gives you a numeric object; entering **`3L`** gives you an integer.
- There is also a special number **`Inf`** which represents infinity; e.g., 1/0; **`Inf`** can be used in ordinary calculations; e.g., 1 / Inf = 0
- The value **`NaN`** represents an undefined value ("not a number"); e.g., 0/0; **`NaN`** can also be thought of as a missing value.

Mathematical Operators

- Arithmetic Operators

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	integer division 5%/%2 is 2

- Logical Operators

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE

Creating Vectors

- The `c()` function can be used to create vectors of object.

```
> x <- c(0.5, 0.6)          ## numeric
> x <- c(TRUE, FALSE)       ## logical
> x <- c(T, F)              ## logical
> x <- c("a", "b", "c")     ## character
> x <- 9:29                 ## integer
> x <- c(1+0i, 2+4i)        ## complex
```

- Using the vector() function

```
> x <- vector("numeric", length = 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
```

Mixing Objects and Explicit Coercion

- There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident; it can also be done on purpose.

```
> y <- c(1.7, "a")    ## character
> y <- c(TRUE, 2)     ## numeric
> y <- c("a", TRUE)   ## character
```

- Objects can be explicitly coerced from one class to another using the `as.*` functions.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
```

```
> as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Lists

- List can be explicitly created using the `list()` function

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE
```

- An empty list with a prespecified length can be created using the `vector()` function.

```
> x <- vector("list", length = 5)
> x
[[1]]
NULL
```

Matrices

- Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (nrow,ncol)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices

- Matrices are constructed column-wise

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices

- Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
 [1]  1  2  3  4  5  6  7  8  9 10
> dim(m) <- c(2, 5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Matrices

- Matrices can be created by column-binding or row-binding with the `cbind()` and `rbind()` functions.
-

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
```

```
> rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

Missing Values

- Missing values are denoted by `NA` or `NaN` for undefined mathematical operations.
  - `is.na()` is used to test objects if they are NA.
  - is.nan() is used to test for NaN.
  - NA values have a class also, so there are integer NA, character NA, etc.
  - A NaN value is also NA but the converse is not true.

Missing Values

```
> ## Create a vector with NAs in it
> x <- c(1, 2, NA, 10, 3)
> ## Return a logical vector indicating which elements are NA
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> ## Return a logical vector indicating which elements are NaN
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE

> ## Now create a vector with both NA and NaN values
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
```

Data Frames

- Data frames are used to store tabular data.
- The are represented as a special type of list where every element of the list has to have the same length.
- Each element of the list can be thought of as a column and the length of each of the list is the number of rows.
- Unlike matrices, data frames can store different classes of objects in each column.
- Data frames also have a special attribute called **`row.names`**
- Data frames are usually created by calling **`read.table()`** or **`read.csv()`**
- Can be converted to a matrix by calling **`data.matrix()`**

Data Frames

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

Getting Help

- If you want to know something about an R command or a function, just type '?' followed by the name of the function. You can also type **`help`**`("name of the function")` on the console.

Console ~/

```
> ?dget
> help("read.csv")
> ?write.table
```

read.table {utils} R Documentation

Data Input

Description

Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

Usage

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),
           row.names, col.names, as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors(),
           fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)

read.csv(file, header = TRUE, sep = ",", quote = "\"",
         dec = ".", fill = TRUE, comment.char = "", ...)
```

- It is also helpful to use the **`example()`** function

Reading Lines of a Text File

- **`readLines`** can be useful for reading in lines of web pages.

```
> con <- url("http://www.uap.asia", "r")
> x <- readLines(con)
> head(x)
[1] "<!DOCTYPE html>"
[2] "<html lang=\"en-US\">"
[3] "<head itemscope itemtype=\"https://schema.org/WebSite\">"
[4] "<meta charset=\"UTF-8\" />"
[5] "<title>UA&amp;P – Where Dragons Live</title><meta name=\"des
[6] "\t\t<meta name=\"robots\" content=\"noodp,noydir\" />"
```

Reading and Writing Lines of Text

```
> ## Open connection to gz-compressed text file
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
 [1] "1080"      "10-point" "10th"      "11-point" "12-point" "16-point"
 [7] "18-point" "1st"       "2"         "20-point"
```

- `writeLines` takes a character vector and writes each element one line at a time to a text file.

Getting to Know Your Dataset

- After reading in a dataset, here are some common functions to run:
  - **`head()`**, **`tail()`** shows the first or the last six values
  - **`summary()`** shows result summaries of the data
  - **`str()`** displays the structure of the data frame
  - **`names()`** displays the column names of a data frame
  - **`nrow()`** shows the number of rows or observations
  - **`ncol()`** shows the number of columns or observations
  - **`sum()`** returns the sum of the arguments
  - **`min()`**, **`max()`** - lowest and highest values
  - **`mean()`**, **`sd()`** - average and standard deviation

Subsetting

- There are a number of operators that can be used to extract subsets of R objects.
- `[` always returns an object of the same class as the original; can be used to select more than one element
- `[[` is used to extract elements of a list or a data frame; it can only be used to extract a single element and a class of the returned object will not necessarily be a list or data frame.
- `$` is used to extract elements of a list or data frame by name; semantics are similar to `[[`.

Subsetting

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]     ## Extract the first element
[1] "a"
> x[2]     ## Extract the second element
[1] "b"
> x[1:4]
[1] "a" "b" "c" "c"
> x[c(1, 3, 4)]
[1] "a" "c" "c"
```

Subsetting

```
> u <- x > "a"
> u
[1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"

> x[x > "a"]
[1] "b" "c" "c" "d"
```

Subsetting a List

```
> x <- list(foo = 1:4, bar = 0.6)
> x
$foo
[1] 1 2 3 4

$bar
[1] 0.6
> x[[1]]
[1] 1 2 3 4
> x[["bar"]]
[1] 0.6
> x$bar
[1] 0.6
```

Subsetting a List

- One thing that differentiates the `[[` operator from the `$` is that the `[[` operator can be used with computed indices. The `$` operator can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
>
> ## computed index for "foo"
> x[[name]]
[1] 1 2 3 4
>
> ## element "name" doesnâ€™t exist! (but no error here)
> x$name
NULL
>
> ## element "foo" does exist
> x$foo
[1] 1 2 3 4
```

Subsetting a List

- The [[operator can take an integer sequence if you want to extract a nested element of a list.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
>
> ## Get the 3rd element of the 1st element
> x[[c(1, 3)]]
[1] 14
>
> ## Same as above
> x[[1]][[3]]
[1] 14
>
> ## 1st element of the 2nd element
> x[[c(2, 1)]]
[1] 3.14
```

Subsetting a Matrix

- Matrices can be subsetted in a usual way with (*i*,*j*) type indices.

```
> x <- matrix(1:6, 2, 3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

```
> x[1, ]   ## Extract the first row
[1] 1 3 5
> x[, 2]   ## Extract the second column
[1] 3 4
```

Dropping Matrix Dimensions

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[1, 2, drop = FALSE]
     [,1]
[1,]    3
```

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5
> x[1, , drop = FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

- By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1x1 matrix.
- Similarly, subsetting a single column or a single row will give you a vector, not a matrix (by default).

Vectorized Operations

- Many operations in R are vectorized, making code more efficient, concise, and easier to read.

```
> x <- 1:4
> y <- 6:9
> z <- x + y
> z
[1]  7  9 11 13
```

```
> x
[1] 1 2 3 4
> x > 2
[1] FALSE FALSE  TRUE  TRUE
```

Vectorized Operations

```
> x >= 2
[1] FALSE  TRUE  TRUE  TRUE
> x < 3
[1]  TRUE  TRUE FALSE FALSE
> y == 8
[1] FALSE FALSE  TRUE FALSE
```

```
> x - y
[1] -5 -5 -5 -5
> x * y
[1]  6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Vectorized Operations

```
> x <- matrix(1:4, 2, 2)
> y <- matrix(rep(10, 4), 2, 2)
>
> ## element-wise multiplication
> x * y
     [,1] [,2]
[1,]   10   30
[2,]   20   40
>
```

```
> ## element-wise division
> x / y
     [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
>
> ## true matrix multiplication
> x %*% y
     [,1] [,2]
[1,]   40   40
[2,]   60   60
```

Sorting

- Use the order() function to sort a vector or dataframe in R

```
# sorting examples using the mtcars dataset
attach(mtcars)

# sort by mpg
newdata <- mtcars[order(mpg),]

# sort by mpg and cyl
newdata <- mtcars[order(mpg, cyl),]

#sort by mpg (ascending) and cyl (descending)
newdata <- mtcars[order(mpg, -cyl),]

detach(mtcars)
```

```
> a <- c(100, 10, 1000)
> order(a)
[1] 2 1 3
```

```
> a[order(a)]
[1]   10  100 1000
```

Exercises

- 1. WHO dataset
  - a. load and run Session1.R
  - b. country with the biggest population
  - c. population of Malaysia
  - d. country with the lowest literacy
  - e. Richest country in Europe based on GNI
  - f. Mean Life expectancy of countries in Africa
  - g. Number of countries with population greater than 10,000
  - h. Top 5 countries in the Americas with the highest child mortality

Exercises

- 2. NBA dataset (Historical NBA Performance.xlsx)
  - a. The year Bulls has the highest winning percentage
  - b. Teams with an even win-loss record in a year
- 3. Seasons_Stats.csv
  - a. Player with the highest 3-pt attempt rate in a season.
  - b. Player with the highest free throw rate in a season.
  - c. What year/season does Lebron James scored the highest?
  - d. What year/season does Michael Jordan scored the highest?
  - e. Player efficiency rating of Kobe Bryant in the year where his MP is the lowest?
- 4. National Universities Rankings.csv
  - a. University with the most number of undergrads
  - b. Average Tuition in the Top 10 University

Other Resources

- Datacamp.com
- R-bloggers.com
- Statmethods.net (Quick-R)
- Rstudio.com
- and of course https://cran.r-project.org/doc/manuals/r-release/R-intro.html
- Swirl   (package.install(swirl)

Quiz #1

Dates in R

- The two standard date-time classes in R are **`POSIXct`** and **`POSIXlt`**.
- ct is short for "calendar time," and the **`POSIXct`** class stores dates as the number of seconds since the start of 1970, in the Coordinated Universal Time (UTC) zone.
- **`POSIXlt`** stores dates as a list, with components for seconds, minutes, hours, day of month, etc.
- **`POSIXct`** is best for storing dates and calculating with them, whereas **`POSIXlt`** is best for extracting specific parts of a date.
- The function **`Sys.time`** returns the current date and time in **`POSIXct`** form:

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

## [1] "2013-07-17 22:47:01 BST"
```

- Now try taking the **`class()`** and **`unclass()`** functions of **`now_ct`**.

Dates in R

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

```
> ## Coerce a 'Date' object from character
> x <- as.Date("1970-01-01")
> x
[1] "1970-01-01"
```

- You can see the internal representation of a Date object by using the unclass() function.

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

Time in R

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

```
> x <- Sys.time()
> x
[1] "2015-04-13 10:09:17 EDT"
> class(x)    ## 'POSIXct' object
[1] "POSIXct" "POSIXt"
```

- The POSIXlt object contains some useful metadata.

```
> p <- as.POSIXlt(x)
> names(unclass(p))
 [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"
 [8] "yday"   "isdst"  "zone"   "gmtoff"
> p$wday      ## day of the week
[1] 1
```

Time in R

- You can also use the `POSIXct` format.

```
> x <- Sys.time()
> x                 ## Already in ‘POSIXct’ format
[1] "2015-04-13 10:09:17 EDT"
> unclass(x)        ## Internal representation
[1] 1428934157
> x$sec             ## Can't do this with 'POSIXct'!
Error in x$sec: $ operator is invalid for atomic vectors
> p <- as.POSIXlt(x)
> p$sec             ## That's better
[1] 17.16238
```

Operations on Dates and Times

- You can use the + and – operators on dates and times. You can do comparisons (==, >, <) too.

```
> 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 in x - y: non-numeric argument to binary operator
> x <- as.POSIXlt(x)
> x-y
Time difference of 356.3095 days
```

```
> x <- as.Date("2012-03-01")
> y <- as.Date("2012-02-28")
> x-y
Time difference of 2 days
```

Operations on Dates and Times

- You can use the + and – operators on dates and times. You can do comparisons (==, >, <) too.

```
> 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 in x - y: non-numeric argument to binary operator
> x <- as.POSIXlt(x)
> x-y
Time difference of 356.3095 days
```

```
> x <- as.Date("2012-03-01")
> y <- as.Date("2012-02-28")
> x-y
Time difference of 2 days
```

Control Structures

- Commonly used control structures are
  - **`if`** and **`else`**: testing a condition and acting on it
  - **`for`**: execute a loop a fixed number of times
  - **`while`**: execute a loop while a condition is true
  - **`repeat`**: execute an infinite loop
  - **`break:`** stop the execution of the loop
  - **`next`** : skip an iteration of a loop

if … else

```
if(<condition>) {
        ## do something
}
## Continue with rest of code
```

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

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

```
if(<condition1>) {

}

if(<condition2>) {

}
```

if … else

- Here are examples of a valid **`if … else`** structure:

```r
## Generate a uniform random number
x <- runif(1, 0, 10)
if(x > 3) {
        y <- 10
} else {
        y <- 0
}
```

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

for Loops

- Here are examples of a valid **`for`** loop structure:

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

```
> x <- c("a", "b", "c", "d")
>
> for(i in 1:4) {
+         ## Print out each element of 'x'
+         print(x[i])
+ }
```

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

```
> ## Generate a sequence based on length of 'x'
> for(i in seq_along(x)) {
+         print(x[i])
+ }
```

```
> for(i in 1:4) print(x[i])
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Nested `for` Loops

- `for` loops can be nested inside each other:

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

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

while Loops

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

```
> z <- 5
> set.seed(1)
>
> while(z >= 3 && z <= 10) {
+         coin <- rbinom(1, 1, 0.5)
+
+         if(coin == 1) {  ## random walk
+                 z <- z + 1
+         } else {
+                 z <- z - 1
+         }
+ }
> print(z)
[1] 2
```

repeat Loops

- **`repeat`** loops initiate an infinite loop

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

repeat {
        x1 <- computeEstimate()

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

next, break

- **`next`** is used to skip an iteration of a loop.

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

- **`break`** is used to exit a loop immediately

```
for(i in 1:100) {
      print(i)

      if(i > 20) {
              ## Stop loop after 20 iterations
              break
      }
}
```

…, break

Functions

- Functions are defined using the **`function()`** directive and are stored as R objects.

```
> f <- function() {
+           ## This is an empty function
+ }
> ## Functions have their own class
> class(f)
[1] "function"
> ## Execute this function
> f()
NULL
```

Functions

- A simple function

```
> f <- function() {
+         cat("Hello, world!\n")
+ }
> f()
Hello, world!
```

Functions

- A function with argument

```
> f <- function(num) {
+         for(i in seq_len(num)) {
+                 cat("Hello, world!\n")
+         }
+ }
> f(3)
Hello, world!
Hello, world!
Hello, world!
```

Function Arguments with Default Values

```
> f <- function(num = 1) {
+         hello <- "Hello, world!\n"
+         for(i in seq_len(num)) {
+                 cat(hello)
+         }
+         chars <- nchar(hello) * num
+         chars
+ }
> f()     ## Use default value for 'num'
Hello, world!
[1] 14
> f(2)    ## Use user-specified value
Hello, world!
Hello, world!
[1] 28
```

Argument Matching

- Arguments can be matched positionally or by name

```
> str(rnorm)
function (n, mean = 0, sd = 1)
> mydata <- rnorm(100, 2, 1)              ## Generate some data

> ## Positional match first argument, default for 'na.rm'
> sd(mydata)
[1] 0.9033251
> ## Specify 'x' argument by name, default for 'na.rm'
> sd(x = mydata)
[1] 0.9033251
> ## Specify both arguments by name
> sd(x = mydata, na.rm = FALSE)
[1] 0.9033251
```

Argument Matching

- Arguments can be matched positionally or by name

```
> ## Specify both arguments by name
> sd(na.rm = FALSE, x = mydata)
[1] 0.9033251

> sd(na.rm = FALSE, mydata)
[1] 0.9033251
```

Lazy Evaluation

```
> f <- function(a, b) {
+           a^2
+ }
> f(2)
[1] 4

> f <- function(a, b) {
+           print(a)
+           print(b)
+ }
> f(45)
[1] 45
Error in print(b): argument "b" is missing, with no default
```

Scoping Rules

- What if there's a conflict in naming your functions. Which one will R follow?

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

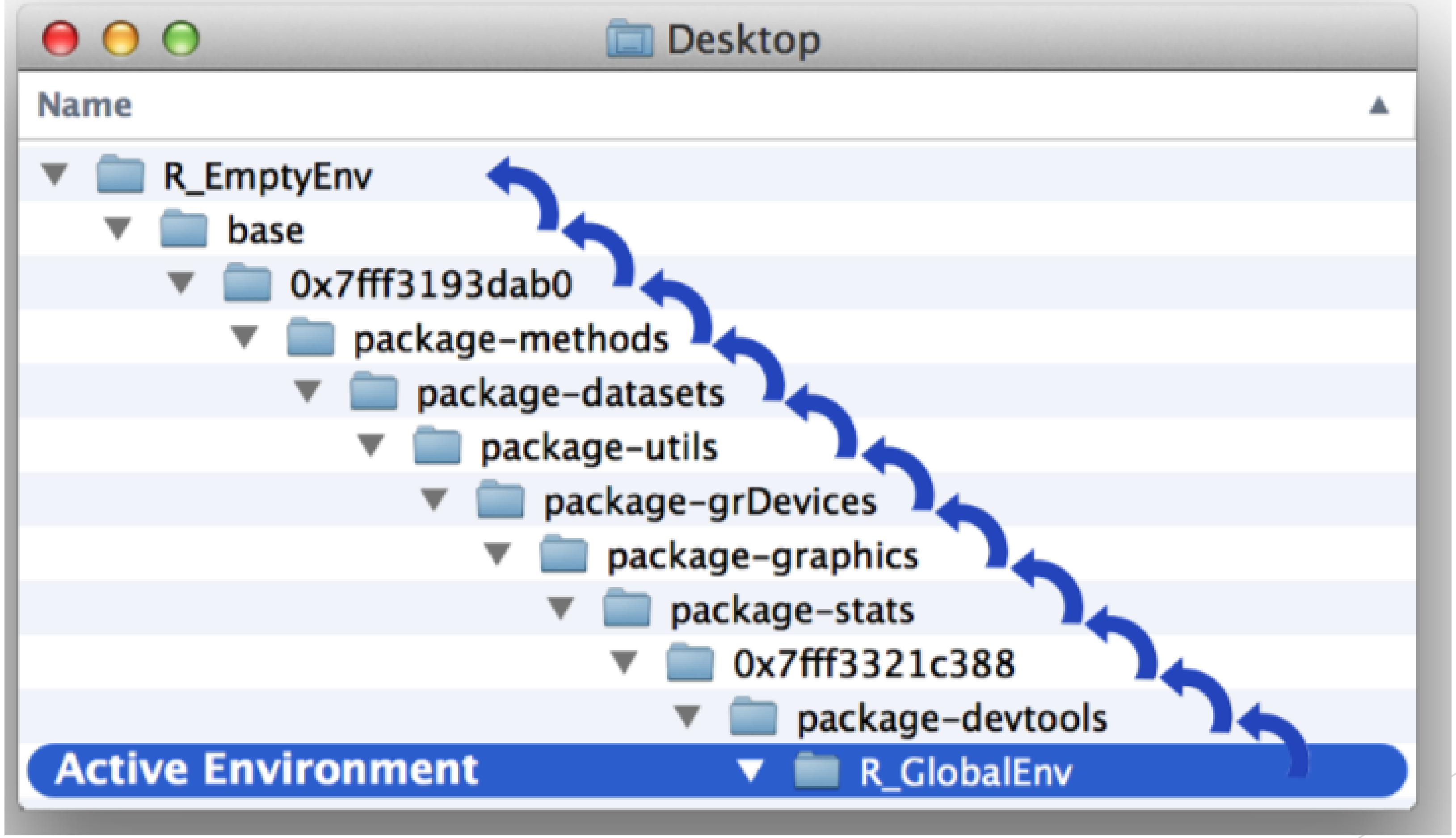
1. Search the global environment (i.e. your workspace) for a symbol name matching the one requested.

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:knitr"     "package:stats"
[4] "package:graphics"  "package:grDevices" "package:utils"
[7] "package:datasets"  "Autoloads"         "package:base"
```

Scoping Rules



Lexical Scoping

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

- Lexical scoping in R means that the values of free variables are searched for in the environment in which the function was defined.
- 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.

Lexical Scoping 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)`?

```
> g <- function(x) {
+         a <- 3
+         x+a+y
+         ## 'y' is a free variable
+ }
> g(2)
Error in g(2): object 'y' not found
> y <- 3
> g(2)
[1] 8
```

Coding Standards

- Always use text files / text editor.
- Indent your code
- Limit the width of your code.
- Limit the length of individual functions.
- Organize your files within a project
- Use version control systems like Git