Fundamentals

Robert B. Gramacy
Virginia Tech Department of Statistics

bobby.gramacy.com

Expressions

R code is composed of a series of expressions.

- assignments
- conditional statements
- arithmetic

Here are a few example expressions.

```
> x <- 1
> if(1 > 2) "yes" else "no"
[1] "no"
> 127 %% 10
[1] 7
```

Expressions are composed of objects and functions, which may be separated by newlines or semicolons.

```
> "an expression"; 7+13; exp(0+1i*pi)
[1] "an expression"
[1] 20
[1] -1+0i
```

And, as we've seen, functions are themselves objects.

- All R code manipulates objects.

Symbols

Formally, variable names in R are called symbols.

When you assign an object a variable name you are actually assigning the object to a symbol in a certain environment or context.

For example `x <- 1` assigns the symbol "x" to the object "1".

```
> x <- 1
```

The environment, like the sequence of expressions, is defined by the programmer.

- more on this later

- *Almost* every symbol you can imagine creating can be assigned to any object,
- and can later be re-assigned to another, possibly different, object. *In fact, a symbol is technically an object.*

Some symbols are off-limits for assignment

- `NA`, `Inf`, `NaN`, `NULL`, `if`, `else`, `while`, `function`, `for`, ...

Others, which come pre-assigned with values, are not.

```
> pi
[1] 3.141593
> pi <- 1
> pi
[1] 1
```

- When the R interpreter evaluates an expression, it evaluates all symbols
- If you compose an object from a set of symbols, R will resolve the symbols at the time the object is constructed.

```
> x <- 1; y <- 2; z <- 3
> v <- c(x, y, z)
> v
[1] 1 2 3
> x <- 10
> v
[1] 1 2 3
```

It is possible to delay the evaluation of an expression so that symbols are not evaluated immediately.

```
> x <- 1
> v <- quote(c(x, y, z))
> eval(v)
[1] 1 2 3
> x <- 5
> eval(v)
[1] 5 2 3
```

Finally, you can create a *promise object* in R to delay the evaluation of a variable until it is (first) needed.

```
> x <- 1
> delayedAssign("v", c(x, y, z))
> x <- 5
> v
[1] 5 2 3
```

- Promise objects are used within packages to make objects available to users without loading them into memory.

Functions

A function is an object that takes some input objects (arguments) and returns an output object.

- Unlike MATLAB, functions cannot return multiple objects.
- They can, however, return a list of objects.

Technically, all work in R is done in functions.

Every statement in R—setting variables, doing arithmetic, repeating code in a loop—can be written as a function.

For example, suppose that you had defined a variable `animals` pointing to a character vector with four elements

```
> animals <- c("cow", "chicken", "pig", "tuba")
```

... and you wanted to change the fourth element

```
> animals[4] <- "duck"
```

That statement is *parsed* into a call to the `[<-` function.

```
> `[<-`(animals, 4, "duck")
[1] "cow"     "chicken" "pig"     "duck"
```

In practice you would probably never write this statement as a function call;

- the bracket notation is more intuitive and easier to read.

However it is helpful to know that every operation in R is a function.

Here is another example

```
> if(1 < 2) "this" else "that"
[1] "this"
> `if`(1 < 2, "this", "that")
[1] "this"
```

Immutable objects

In assignment statements, most objects are immutable.

- R will copy the object, not just the reference.

```
> u <- list(1)
> v <- u
> u[[1]] <- "hat"
> u
[[1]]
[1] "hat"

> v
[[1]]
[1] 1
```

This applies to vectors, lists, and most other primitive objects.

Including functions. E.g.,

```
> f <- function(x, i) { x[i] <- 4 }
> w <- c(10, 11, 12, 13)
> f(w, 1)
> w
[1] 10 11 12 13
```

- The vector `w` is copied when it is passed to the function.
- The value `x` is modified inside the context of the function.

Special Values

- `NA`: reserved for missing values

```
> v <- c(1,2,3)
> length(v) <- 4
> v
[1]  1  2  3 NA
```

- `Inf` and `-Inf`: for numbers too large for machine representation

```
> 2^1024
[1] Inf
> -2^1024
[1] -Inf
```

- `NaN`: for numerical results that make no sense (not a number)

```
> Inf - Inf
[1] NaN
> 0 / 0
[1] NaN
```

- `NULL` : a symbol representing no value.
  - Usually `NULL` is used as argument to a function to indicate that no value is being provided for that argument.
  - Sometimes function return-values are `NULL`.

Coercion

When you call a function with an argument of the wrong type, R will try to coerce the values to a different type so that the function will work.

With generic functions, R will look for a suitable method.

- If no suitable match exists, R will search for a coersion method that converts the object to a type for which a suitable method does exist.

For example ...

```
> x <- 1:5
> x
[1] 1 2 3 4 5
> class(x)
[1] "integer"
> x[4] <- "hat"
> x
[1] "1"   "2"   "3"   "hat" "5"
> class(x)
[1] "character"
```

An overview of conversion rules:

- Logical values become numbers: `TRUE` becomes `1` and `FALSE` becomes `0`.
- Values are converted to the simplest type required to represent all information.
- The ordering is roughly

logical $<$ integer $<$ numeric $<$ complex $<$ character $<$ list

- Objects of type `raw` are not converted to other types.
- Object attributes are dropped when an object is coerced from one type to another.

You can inhibit coersion when passing arguments to functions by using the `I()` function.

Operators

We already discussed how operators (`+`, `^`, `*`, ...) are really just functions.

- These are some of the the built-in *binary* operators.

You can define your own operators. E.g.,

```
> `%myop%` <- function(a, b) { 2*a + 2*b }
> 1 %myop% 1
[1] 4
> 1 %myop% 2
[1] 6
```

There are *unary* operators too, e.g., `-`.

Order of operations:

1. function calls and grouping expressions: `{}`, `()`
2. index and lookup operators: `[]`
3. arithmetic: `+/-*`, with the usual precedence
4. comparison: `<=>`
5. formulas: `~`
6. assignment: `<-`
7. help: `?`

Special assignments

Most assignments simply assign an object to a symbol.

```
> y <- list(shoes="loafers", hat="ball cap",
+       shirt="white")
> z <- function(a, b, c) { a^b/c }
> v <- 1:8
```

R also allows assignments with a function on the left-hand side of the assignment operator:

```
> dim(v) <- c(2,4)
> v[2,2] <- 10
> formals(z) <- alist(a=1, b=2, c=3)
```

Grouping expressions

We already saw that you can place multiple expressions on the same line with separating semicolons.

Parentheses elevate the precedence of an operation to that of a function call.

- In fact, a parenthetical grouping is equivalent to evaluating an identity function.

```
> 2 * (5 + 1)
[1] 12
> f <- function(x) x
> 2 * f(5 + 1)
[1] 12
```

Curly braces are used to evaluate a series of expressions (separated by newlines or semicolons) and return only the last expression.

A common use is to group operations in the body of a function.

```
> f <- function() { x <- 1; y <- 2; x + y }
> f()
[1] 3
```

But they can also be used in other contexts.

```
> { x <- 1; y <- 2; x + y }
[1] 3
```

Code in curly braces are evaluated in the current environment.

- A new environment is created by a function call but *not* by the use of curly braces.

```
> f <- function() { u <- 1; v <- 2; u + v }
> f()
[1] 3
> u
Error: object 'u' not found
> { u <- 1; v <-2; u + v }
[1] 3
> u
[1] 1
```

Control Structures

Conditional statements take the form

`if(`*condition*`)` *true_expression* `else` *false_expression*

or, alternatively

`if(`*condition*`)` *true_expression*

Some examples:

```
> if(FALSE) "not printed"
> if(FALSE) "not printed" else "printed"
[1] "printed"
```

Conditional statements are not treated as vector operations.

If the *condition* statement is a vector of more than one `logical` value, then only the first term will be used. E.g.,

```
> x <- 10
> y <- c(8, 10, 12, 3, 17)
> if(x < y) x else y
[1]  8 10 12  3 17
Warning message:
In if (x < y) x else y :
  the condition has length > 1 and only
  the first element will be used
```

`ifelse` is vectorized if you need it:

```
> a <- rep("a", 5)
> b <- rep("b", 5)
> tf <- (1:5) %% 2 > 0
> ifelse(tf, a, b)
[1] "a" "b" "a" "b" "a"
```

Often, it is convenient to return different values (or call different functions) depending on a single input value.

```
> switcheroo.if.then <- function(x) {
+   if(x == "a") "alligator"
+   else if(x == "b") "bear"
+   else if(x == "c") "camel"
+   else "moose"
+ }

> switcheroo.if.then("b")
[1] "bear"
```

Its cleaner to use the `switch` function.

```
> switcheroo.switch <- function(x) {
+   switch(x,
+          a="alligator",
+          b="bear",
+          c="camel",
+          "moose")
+ }

> switcheroo.switch("b")
[1] "bear"
```

There are three different looping constructs in R.

The simplest is `repeat`,

> `repeat` *expression*

which repeats the same *expression* until a `break` keyword is executed therein.

- If you don't include a `break` command, the R code will be in an infinite loop.
- A `next` command allows you skip the rest of the loop iteration without evaluating the remaining *expressions* in the loop body.

As an example, the following code prints out multiples of 5 up to 25.

```
> i <- 5
> repeat {
+   if(i > 25) break
+   else { print(i); i <- i + 5 }
+ }
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

Another useful construction is `while` loops, which repeat an expression while a condition is true:

`while(`*condition*`)` *expression*

```
> i <- 5
> while(i <= 25) { print(i); i <- i + 5 }
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

- You can use `break` and `next` inside `while` loops.

Finally, R provides `for` loops, which iterate through each item in a vector (or a list):

`for(`*var* `in` *list*`)` *expression*

```
> for(i in seq(5, 25, by=5)) print(i)
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

- Again, `break` and `next` work with `for` loops.

There are two important properties of (all) looping statements to keep in mind.

1. Results are not printed unless you explicitly call `print`.

   ```
   > for(i in seq(5, 25, by=5)) i
   ```

2. The variable *var* that is set in a `for` loop is changed in the calling environment.

   ```
   > i <- 1
   > for(i in seq(5, 25, by=5)) i
   > i
   [1] 25
   ```

Accessing Data Structures

We've seen three ways of accessing information from vectors, lists, arrays, matrices, and data frames.

`x[i]` `x[[i]]` `x$n`

About brackets:

- `[[]]` always returns a single element, whereas `[]` may return multiple elements.
- When an element is referred to by name, as opposed to by element, `[]` requires exact matches whereas `[[]]` allows partial ones.
- With lists, `[]` gives a list whereas `[[]]` gives a vector.

In the last lecture we saw lots of indexing examples; here are some more advanced features/details.

You can use negative indices to return a vector consisting of all elements *except* those indexed.

```
> v <- 100:119
> v
 [1] 100 101 102 103 104 105 106 107 108 109
[11] 110 111 112 113 114 115 116 117 118 119
> v[-15:-1]
[1] 115 116 117 118 119
```

The same applies to lists.

```
> l <- list(a=1, b=2, c=3, d=4, e=5,
+           f=6, g=7, h=8, i=9, j=10)
> l[-7:-1]
$h
[1] 8

$i
[1] 9

$j
[1] 10
```

When selecting a subset of a larger dimensional object, R will automatically coerce the results to an object appropriate for the new dimensions.

```
> class(a)
[1] "array"
> class(a[1,,])
[1] "matrix"
> class(a[1,1,])
[1] "integer"
> class(a[1:2,1:2,1:2])
[1] "array"
> class(a[1,1,1, drop=FALSE])
[1] "array"
```

You can replace elements of a vector, matrix, or array using brackets.

```
> a[1,,]
     [,1] [,2] [,3] [,4]
[1,]  101  107  113  119
[2,]  103  109  115  121
[3,]  105  111  117  123
> a[1,1:2,1:2] <- 1:4
> a[1,,]
     [,1] [,2] [,3] [,4]
[1,]    1    3  113  119
[2,]    2    4  115  121
[3,]  105  111  117  123
```

Lists can be indexed by name using `$`, but you can also use `[]`.

```
> l[c("a", "b", "c")]
$a
[1] 1

$b
[1] 2

$c
[1] 3
```

You can index by name with `[[]]` for single elements.

```
> dairy <- list(milk="1 gallon", butter="1 pound",
+               eggs=12)
> dairy$milk
[1] "1 gallon"
> dairy[["milk"]]
[1] "1 gallon"
```

You can also use partial names with `exact = FALSE`.

```
> diary[["mil"]]
NULL
> dairy[["mil", exact=FALSE]]
[1] "1 gallon"
```

Sometimes an object is a list of lists, in which case you can use `[[]]` with a vector index of names or numbers.

```
> fruit <- list(apples=6, oranges=3, bananas=10)
> shopping.list <- list(dairy=dairy, fruit=fruit)
> shopping.list[[c("dairy", "milk")]]
[1] "1 gallon"
> shopping.list[[c(1,2)]]
[1] "1 pound"
```

Factors

When analyzing data, it is quite common to encounter categorical values.

For example, suppose you have a set of observations about people that includes eye color.

You could represent eye colors as a character array.

```
> ecc <- c("brown", "blue", "blue", "green",
+          "brown", "brown", "brown")
```

This is a perfectly valid way to represent information, but it can become inefficient if you are working with large names or a large number of observations.

- R provides a better way to represent categorical values, by using factors.

A factor is an ordered (potentially) collection of items.

```
> ecf <- factor(ecc)
> ecf
[1] brown blue  blue  green brown brown brown
Levels: blue brown green
```

In this example, *order* probably does not matter. But you can use factors to represent ordered categories. E.g.,

```
> ecf <- factor(ecc, levels=c("green", "brown",
+                             "blue"), ordered=TRUE)
> ecf
[1] brown blue  blue  green brown brown brown
Levels: green < brown < blue
```

- By default, the order is alphanumeric.

- Factors are implemented internally using integers.
- The levels attribute maps integer to factor level.
- Integers take up a small, fixed amount of storage space.

```
> unclass(ecf)
[1] 2 3 3 1 2 2 2
attr(,"levels")
[1] "green" "brown" "blue"
> as.integer(ecf)
[1] 2 3 3 1 2 2 2
```

Data Frames

It is worth repeating that data frames are a useful way to represent data in R.

- Many experiments consist of individual observations, each of which involves several different measurements.
- Often, those measurements have different dimensions and sometimes there are qualitative, not quantitative.

A data frame provides a structure for representing such data in a tabular (row/column) format.

- Each column may be a different type, but each row must have the same length.

The row and column names of a data frame (and a matrix) can be accessed as follows.

```
> colnames(trees)
[1] "Girth"  "Height" "Volume"
> names(trees)
[1] "Girth"  "Height" "Volume"
> rownames(trees)[1:6]
[1] "1" "2" "3" "4" "5" "6"
```

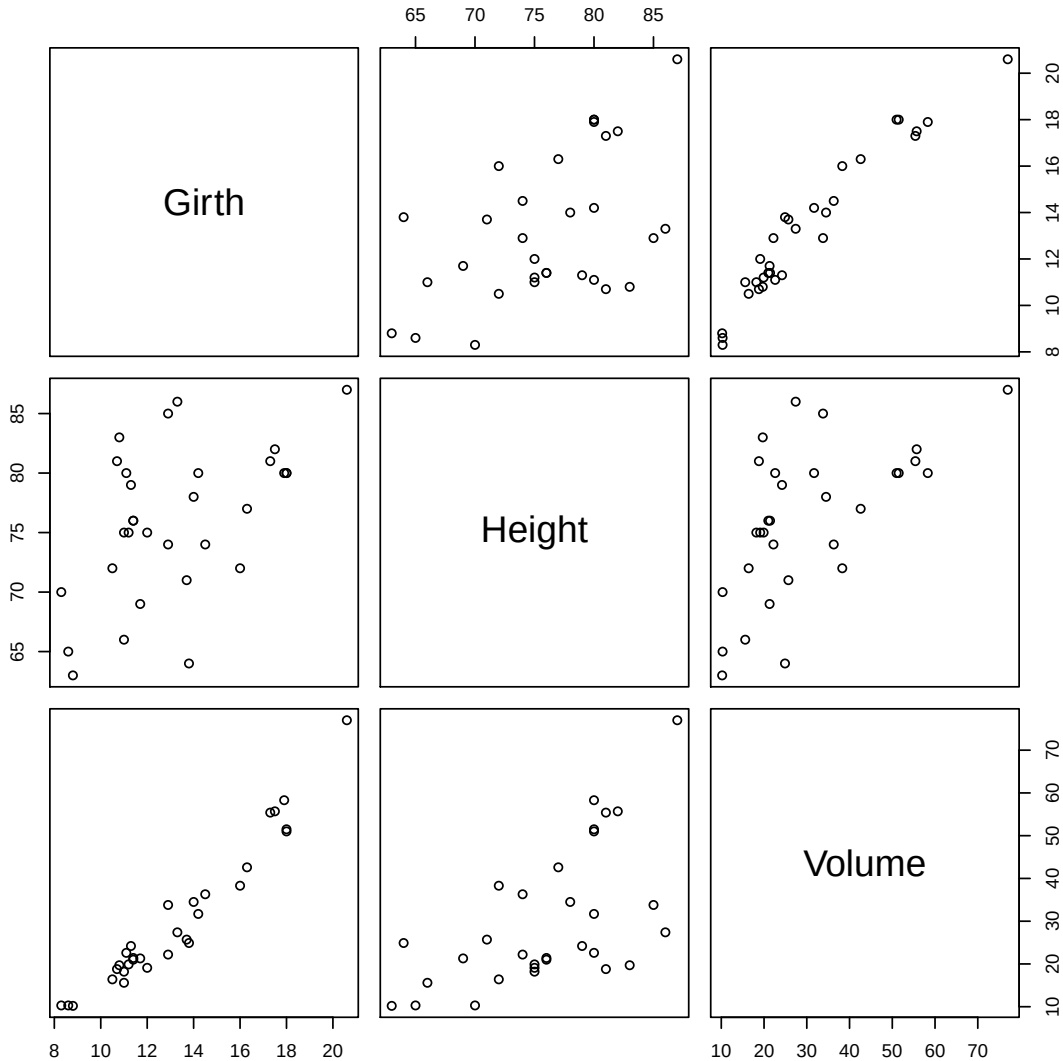
- Assignment can be used to make changes.

Consider data on Cherry trees, which is stored in a data frame, measuring trees girth, height, and volume.

```
> trees
   Girth Height Volume
1    8.3     70   10.3
2    8.6     65   10.3
3    8.8     63   10.2
4   10.5     72   16.4
...
30  18.0     80   51.0
31  20.6     87   77.0
```

A pairs plot offers a nice visual inspection of the relationships between variables in a data frame.

```
> pairs(trees)
```



Formulas

Modeling such data requires the use of formulas to express a relationship between variables.

For example, we may wish to fit the linear model

$$V_i = \beta_0 + H_i\beta_1 + G_i\beta_2 + G_i^2\beta_3 + \varepsilon_i.$$

The formula describing this is

```
> formula(Volume~Height + Girth + I(Girth^2))
Volume ~ Height + Girth + I(Girth^2)
```

- The `I()` "properly" express the squared term.

Formulas are required for most fitting commands, e.g., `lm`.

```
> lm(Volume~Height+Girth+I(Girth^2),
+    data=trees)

Call:
lm(formula = Volume~Height+Girth+I(Girth^2),
   data=trees)

Coefficients:
(Intercept)       Height        Girth
    -9.9204       0.3764      -2.8851
 I(Girth^2)
     0.2686
```

They offer handy abbreviations for compactly expressing longer formulas.

```
> coef(lm(Volume ~ . + I(Girth^2), data=trees))
(Intercept)       Girth      Height
 -9.9204060  -2.8850787   0.3763873
 I(Girth^2)
  0.2686224
```

- (More shorthand examples in the code.)
- also see `? formula`

Time Series

Many important problems look at how a variable changes over time, and R includes a class to represent this data: time series objects.

- Regression functions for time series (like `ar` or `arima`) use time series objects.
- Many plotting functions have special methods for time series.

The `turkey` data, below, contains quarterly total sales (in thousands) of one-day-old turkey chicks from hatcheries in Eire over a period of years starting in 1974.

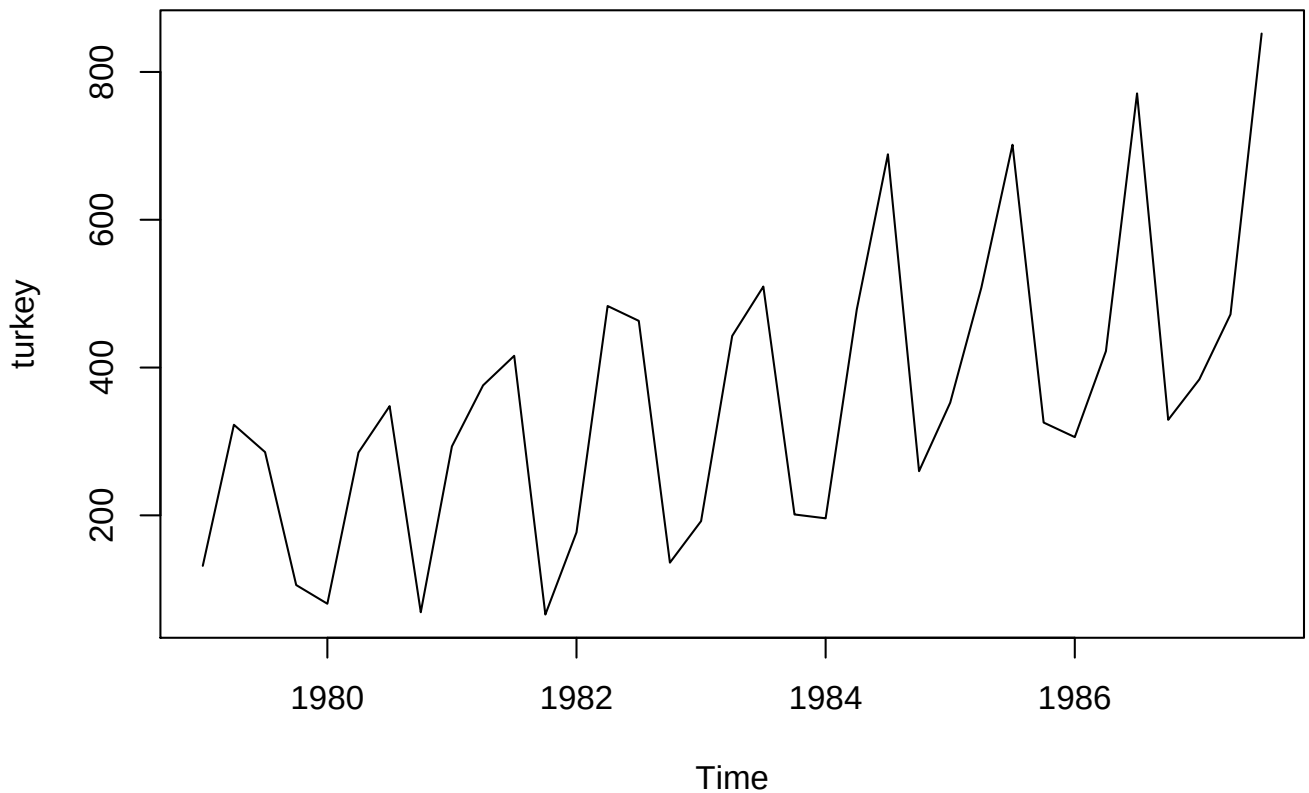
```
> turkey
 [1] 131.7 322.6 285.6 105.7  80.4 285.1
 [7] 347.8  68.9 293.3 375.9 415.9  65.8
[13] 177.0 483.3 463.2 136.0 192.2 442.8
[19] 509.6 201.2 196.0 478.6 688.6 259.8
[25] 352.5 508.1 701.5 325.6 305.9 422.2
[31] 771.0 329.3 384.0 472.0 852.0
```

Converting to a time series object allows extra attributes in the data representation.

```
> turkey <- ts(turkey, start=1974, freq=4)
> turkey
      Qtr1  Qtr2  Qtr3  Qtr4
1979 131.7 322.6 285.6 105.7
1980  80.4 285.1 347.8  68.9
1981 293.3 375.9 415.9  65.8
1982 177.0 483.3 463.2 136.0
1983 192.2 442.8 509.6 201.2
1984 196.0 478.6 688.6 259.8
1985 352.5 508.1 701.5 325.6
1986 305.9 422.2 771.0 329.3
1987 384.0 472.0 852.0
```

Time series `plot()` methods show the extra information, and automatically connect the "dots".

```
> plot(turkey)
```



Environments

Like everything in R, environments (which define a scope for symbols) are objects.

- Internally, R stores symbol mappings in hash tables.

You can list the objects/symbols in the current environment.

```
> objects()
[1] "turkey" "v"      "x"      "y"
[5] "z"
```

You can remove/delete objects from the current environment.

```
> rm(x)
> objects()
[1] "turkey" "v"      "y"      "z"
```

When a user starts a new session in R, the system creates a new global environment for objects.

- The global environment is not actually the *root* of the tree of environments.
- Rather, it is the last environment in the chain/search path of environments initialized when R started up.
- Every environment has a parent environment except one, the *empty environment*.
- All environments chain back to the empty environment.

`(see accompanying .R file)`

Environments and Functions

When a function is called in R,

- a new environment is created within the body of the function
- and the arguments of the function are assigned symbols in the local environment.

E.g.,

```
> env.demo <- function(a, b, c, d) {
+   print(objects())
+ }
> env.demo(1, "truck", 1:5, pi)
[1] "a" "b" "c" "d"
```

The parent environment of a function is the environment from which the function was *created*.

- I.e., not necessarily the one from which it was called.
- If the function was created in another environment, such as a package, then the parent environment will not be the same as the calling environment.

However, it is possible to access the environment from which a function was called.

R maintains a stack of calling environments.

- Each time a new function is called, a new environment is *pushed* onto the stack.
- When the function is finished evaluating, the environment is *popped* off.

There are many functions for inspecting/manipulating the stack.

- The most useful is `parent.frame()` which allows you to access the calling environment.
- The `eval()` function can evaluate an expression in any environment.

The best example of when this would be useful comes from how formulas are used within modeling functions like `lm` and `glm` when the programmer omits the `data` argument.

```
> x <- 1:10
> y <- 5 + x + rnorm(10)
> fit <- lm(y~x)
```

- The formula `y~x` *represents* a relationship between variables.
- It does not copy/paste those variables to `lm`.
- Rather, `lm` reads the formula and pulls `x` and `y` from the calling environment via `parent.frame()`

Sometimes it is convenient to treat a data frame or a list as an environment.

- This lets you refer to each item in the data frame or list by name as if you were using symbols.

```
> df <- data.frame(a=1, b=2, c=3)
> d <- with(df, a+b+c)
> print(d)
[1] 6
> df2 <- within(df, d <- a+b+c)
> df2
  a b c d
1 1 2 3 6
```

Attach

R provides a shorthand for adding objects to the current environment

- e.g., the columns of a data frame.

```
> df
  a b c
1 1 2 3
> attach(df)
> print(c(a,b,c))
[1] 1 2 3
```

- `attach()` works with data frames, lists, and data saved with `save()`.
- You can undo an attachment.

  ```
  > detach(df)
  ```
- `attach()` will warn if an attaching column name clashes with a symbol in the current environment (and it will not attach that column).

  In particular, be careful with `attach()` when working with multiple data sets with variables of the same or similar names.