An Introduction to R

Based on: An Introduction to R, W. N. Venables, D. M. Smith, and the R Development Core Team.

The document in pdf is on the course web page: http://ml.stat.purdue.edu, under the heading R Manual

These lecture slides are on the web page under the heading Lectures: Introduction to R Based on Manual

These lectures are under revision
Basics

Installation

R can be downloaded from one of the mirror sites

http://cran.r-project.org/mirrors.html

You should pick your nearest location

R Session

After R is started, there is a console waiting for input

At the prompt >, you can enter R code

> 1 + 2
[1] 3
Variable Assignment

We assign values to variables with the assignment operator ".=". Just typing the variable by itself at the prompt will print out the value.

We should note that another form of assignment operator "<-" is also in use.

```r
> x = 1
> x
[1] 1
> x <- 1
```
Functions

A function is invoked by its name, then a left parenthesis, then 0 or more arguments separated by ,’s, and the a right parenthesis.

The function c() concatenates. (Use () at end when referring to a function.)

In this case it forms a vector with length 3 and assigns it to the object v3.

```r
> v3 <- c(2,1,3)
```

```r
sort(v3)
```

```r
v3+1
```

Results of a function call can be an argument of a function

```r
sort(c(2,1,3))
```

```r
sortedv4 <- sort(v4 <- c(2,1,3,11))
```
Comments

All text after a "hashtag" within a single command line is a comment.

> 1 + 1 "hashtag" this is a comment
[1] 2
Install Packages

The “standard distribution” is maintained by the R core group
- What you get when you install R
- R internals
- Also, many capabilities for managing and analyzing data
- lattice, a package implementing the Trellis Display visualization framework
- `lm()` for linear regression
Install Packages

CRAN
- Repository of R software packages for managing and analyzing data
- Network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R
- Submitted by thousands around the world

Get an R CRAN package:
install.packages()
Help
R provides access to the documentation for R functions, data sets, and other objects in the standard R distribution and installed CRAN packages.

For objects in the standard distribution and loaded packages

> ?lm

> help(lm)

help() has more capabilities

For a function in an installed package not currently loaded

help(rlm, package="MASS")
Quit R
It will ask what to do with the work space. **Save workspace image? [y/n/c]**

- "y" will save new objects included in the R session, adding to saved objects from previous sessions.
- "n" does not save new objects in current session.
- "c" is to continue and not quit.

For example, data structures, functions, function outputs.
Basics

Objects and Methods

R is an “object-oriented system”. Almost everything in R is an object with a class.

```r
a <- 1
a
```
a is an object

```r
class(a)
print(a)
```

`print()` is a generic function for the method of printing (displaying) objects. How it works depends on the class.

If a print method is has not been written for a class, then `print.default()` is applied, pretty much a "dump".

Typing an object and hitting return applies `print()` to the object.
Objects and Methods

R internal storage mode of an object.

```r
typeof(1)
```
List of attached packages. Each is an environment.  

First in the list is R global environment, 1st in search list.

```
ls()
```

Lists objects in R global environment, 1st in search list.

```
objects(6)
```

Lists objects in 6th environment in search list.

```
ls() and objects(1) have the same result
```
Create vector of integers 1 to 10 and assign to \( a \).

\[
a \leftarrow 1:10
\]

Create vector of random normals with mean = 0 variance =1.

\[
b \leftarrow \text{rnorm}(10)
\]

Concatenate \( a \) and \( b \).

\[
c(a,b)
\]

Vector from 1 to 3 in steps of 0.5.

\[
\text{seq}(\text{from} = 1, \text{to} = 3, \text{by} = 0.5)
\]
Basics

Length of object.

\[ \text{length}(b) \]

Maximum of object.

\[ \text{max}(b) \]

Sort object.

\[ \text{sort}(b) \]

Reverse order of sorted values of object.

\[ \text{rev}(\text{sort}(b)) \]

Range of object.

\[ \text{range}(b) \]
Attach package lattice graphics to search list.

```r
library(lattice)
search()
```

Create two vectors of 100 random normals.

```r
x <- rnorm(100)
y <- rnorm(100)
```

Plot one vector against another with function `xyplot()` from lattice. Default is screen device.

```r
xyplot(y ~ x)
```
Basics

Set graphics device to pdf and write to file.

```
 trellis.device(pdf, file = "scatterplot.pdf")
```

Write display to `scatterplot.pdf`, finish plot, and take active device back to default.

```
 xyplot(y ~ x)
 graphics.off()
```
Set graphics device to pdf and write to file.

```r
trellis.device(pdf, file = "scatterplot.pdf")
```

Assign plot object to `yvsx`.

```r
yvsx <- xyplot(y ~ x)
class(yvsx)
```

Write object to file.

```r
yvsx
graphics.off()
```
Replicate 1 to 10, 10 times.

```r
grps <- rep(1:10,10)
```

Make `grps` a factor

```r
grps <- as.factor(grps)
levels(grps)
```

Plot `y` vs. `x` for each level of `grps`.

```r
xyplot(y ~ x | grps)
```

Plot `y` vs. `x` for each level of `grps` and make one plot per page

```r
xyplot(y ~ x | grps, layout = c(1,1,10))
```
Basics

Set graphics device to pdf and write to file.

```r
trellis.device(pdf, file = "scatterplot.pdf")
xyplot(y ~ x | grps, layout=c(1,1))
graphics.off()
```
Basics

Make a data frame of three columns, x and y

```r
randa <- data.frame(x=x, y = y, grps=grps)  
class(randa) 
names(randa) 
dim(randa)  
summary(randa)
```

Where is x?

```r
find("x")
```
Basics

Make `randata.df` an environment on the search list. Makes the columns in the data frame visible as variables.

```r
attach(randata.df)
search()
find("x")
rm(x,y,grps)
find("x")
```

So you can do this.

```r
xyplot(y ~ x|grps, layout=c(1,1))
```

To get rid of it.

```r
detach()
```

This is a very powerful mechanism, but you must treat it with great respect because it can cause you great trouble.
Another way to get at $x$ and $y$

```r
x <- seq(1,20,length=1000)
y <- x + 2*rnorm(1000)
grps <- as.factor(rep(1:2,500))
randata.df <- data.frame(x=x, y=y, grps=grps)
trellis.device(pdf, file = "scatterplot.pdf")
xyplot(y~x|grps, data=randata.df, layout=c(2,1))
graphics.off()
```
The panel function.

trellis.device(pdf, file = "scatterplot.pdf")

xyplot(y~x|grps, data = randata.df, layout=c(2,1),
panel=function(x,y){
  panel.xyplot(x,y)
  panel.loess(x,y, lwd=3)
}

graphics.off()
Fit a simple linear regression and look at the analysis

```r
linearfit.lm <- lm(y~x, data=randata.df)
linearfit.lm
summary(linearfit.lm)
linearfit.lm <- lm(y~x+grps, data=randata.df)
linearfit.lm
summary(linearfit.lm)
```
Basics

`psl.csv` on course web site.

```r
?read.csv

rawpsl.df <- read.csv("psl.csv")

rawpsl.df$perseat

rawpsl.df[,"perseat"]

rawpsl.df[1,]

rawpsl.df[1,"perseat"]

rawpsl.df[3:6,"perseat"]

Avoid this:

rawpsl.df[,10]
```
Section 2.1 and 2.2

Assignment.

```r
assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

Functions.

```r
log, exp, sin, cos, tan, sqrt, length, min, max, var
sum((x-mean(x))^2)/(length(x)-1)
```

Complex numbers.

```r
sqrt(-17)
sqrt(-17+0i)
```
Section 2.3

Sequences and precedence.

1:30

2*1:30

n <- 10

1:n-1

1:(n-1)

seq(from=1, to=30)

seq(-5, 5, by=.2) -> s3

w <- seq(length=51, from=-5, by=.2)

rep(w, times=5)

rep(w, each=5)

?seq

?rep
Logical operators.

\(<, \leq, >, \geq, ==, !=\)

Logical expressions result in logical objects. Scalar values are \text{T}RU\E (T), \text{F}ALSE (F).

\(x == y\)
x <- 1:15
y <- 15:1
logical1 <- x < y
logical2 <- x == y
typeof(logical2)
class(logical2)
mode(logical2)
And.

(logical1) \& (logical2)

Or.

(logical1) \mid (logical2)

Not.

!(logical1)

Coercion to numeric: F to 0 and T to 1

\text{sum}(\text{logical2})
Section 2.5

Missing elements of vector: NA

Vector can be numeric, character, or logical

\[ z \leftarrow c(1:3, NA) \]

\[ \text{logical}z \leftarrow \text{is.na}(z) \]

\[ \text{logical}z \]

\[ z == \text{NA} \]

NA in a vector will defeat a lot of functions.

\[ \text{sum}(z) \]
Section 2.5

Not a Number, or NaN, values are also NA

1/0
0/0
Inf - Inf
is.na(0/0)
is.nan(0/0)
is.nan(NA)
is.na(NaN)

Watch out.
is.nan("a")
Character strings.

\[
myn\name \leftarrow "Francesca"
\]

\[
myn\name \leftarrow \texttt{'Francesca'}
\]

But print method uses " "

\[
myn\name
\]

Watch out. This is different.

\[
myn\name \leftarrow " \texttt{ Francesca}"
\]
Section 2.6

Does not work.

' ' '

" " "

But.

' ' '

" " "

" " "
Section 2.6

C-style escape sequences

'\n' newline
'\r' carriage return
'\t' tab
'\b' backspace
'\\' backslash '\'

"\"

"""

"make \n this \n vertical"

See ?Quotes
Section 2.6

Concatenate arbitrary number of arguments one by one into character strings.

```r
paste()
paste("Mean of 1 to 10 =", mean(1:10))
a <- c("1", NA)
paste(a,a)
paste(c("x","y"), 1:2, sep ="")
paste(c("x","y"), 1:10, sep =")
paste(c("x","y"), 1, sep =")
string <- "?"
paste(c("x","y"), 1, sep = string)
```

See ?paste
Subsets of the elements of a vector may be selected by appending to the name of the vector an index vector in square brackets.

More generally any expression that evaluates to a vector may have subsets of its elements similarly selected by appending an index vector in square brackets immediately after the expression.
1. A vector of positive integers.

\[ x \leftarrow 1:20 \]

Values in the index vector must lie in the set

\[ 1, 2, \ldots, \text{length}(x). \]

\[ x[c(6, 2, 20)] \]

\[ c("x","y")[\text{rep}(c(1,2,2,1), \ \text{times}=4)] \]

2. A vector of negative integral quantities.

\[ x[-c(6, 2, 20)] \]
3. A logical vector.

Must be of the same length as the vector from which elements are to be selected.

```r
x <- 1:20
x[x == 7]
x <- c(1:10, NA)
x
x[!is.na(x)]
(x+1)[(!is.na(x)) & x>8]
```
Section 2.7

4. A vector of character strings.

Vector object has a names attribute to identify its components.

```r
fruit <- c(5, 10, 1, 20)
names(fruit) <- c("orange", "banana", "apple", "peach")
fruit
names(fruit)
fruit[c("apple","orange")]
```
Section 2.7

An indexed expression can also appear on the receiving end of an assignment, in which case the assignment operation is performed only on those elements of the vector.

```r
x <- c(1:10,NA)
x[is.na(x)] <- 0

y <- -10:10
y[y < 0] <- -y[y < 0]

same as

y <- abs(y)
```
Section 3.1

Just about everything in R is an object.

Atomic objects are all of the same mode: numeric, complex, logical, character and raw.

Vectors must have the same mode.

NA is an exception.

```r
mode(NA)
mode(NaN)
mode(NA)
```
Section 3.1

R provides changes of mode almost anywhere it could be considered sensible to do so (and a few where it might not be).

```r
z <- 0:9
digits <- as.character(z)
digits
digits
d <- as.integer(digits)
d
```
Section 3.1

R also operates on objects called lists, which are of mode list.

Ordered sequences of objects which individually can be of any mode.

Lists are known as recursive rather than atomic structures since their components can themselves be lists in their own right.

```r
fruit.list <- as.list(fruit)

names(fruit.list)
lapply(fruit.list, mode)
fruit.list[1]
mode(fruit.list[1])
fruit.list[[1]]
mode(fruit.list[[1]])
fruit.list[[1]] <- ls
lapply(fruit.list, mode)
```
attributes(fruit)

attributes(rawpsl.df)
Section 3.2

An empty object may still have a mode

\[
e
\]

\[
e \leftarrow \text{numeric}()
\]

\[
\text{length}(e)
\]

\[
\text{mode}(e)
\]

This is “interesting”

\[
e[3] \leftarrow 17
\]

\[
e
\]

You can truncate the length of an object.

\[
\text{alpha} \leftarrow 1:10
\]

\[
\text{alpha} \leftarrow \text{alpha}[2 \times 1:5]
\]

\[
\text{length(alpha)} \leftarrow 3
\]

\[
\text{alpha}
\]

\[
\text{length(alpha)} \leftarrow 10
\]

\[
\text{alpha}
\]
Section 3.3

Return a list of all the non-intrinsic attributes currently defined of an object

```r
mode(attributes(rawpsl.df))
names(attributes(rawpsl.df))
attributes(rawpsl.df)
is.list(rawpsl.df)
is.data.frame(rawpsl.df)
typeof(rawpsl.df)
```
Section 3.3

Select an attribute of an object

\texttt{attr(rawpsl.df,"names")}

Define or change an attribute of an object

\texttt{a <- 1:10}

\texttt{attr(a,"ownerofa") <- "myname"}

Note the print method

\texttt{a}

Compare with

\texttt{rawpsl.df[1:2,]}

Section 3.4

All objects in R have a class, reported by `class()`.

For simple vectors this is just the same as `mode()`, for example:

```
numeric logical character list
```

Other classes

```
matrix array factor data.frame
```

Generic functions like `print()` provide methods for classes

Remove the class

```
unclass(rawpsl.df[1:2,])
```

```
is.data.frame(unclass(rawpsl.df[1:2,]))
is.list(unclass(rawpsl.df[1:2,]))
is.list(unclass(unclass(rawpsl.df[1:2,])))
```
Section 4.1

factor: a vector object used to specify a discrete classification of the elements of a vector into levels.

Factors can be ordered or unordered

?factor

?ordered

lapply(rawpsl.df, class)

attach(rawpsl.df)

is.ordered(area)

is.factor(area)
Section 4.1

attach(rawpsl.df)
sapply(rawpsl.df, is.factor)
ar <- factor(as.character(area))
levels(ar)[13] <- "090e2"
ar <- factor(as.character(ar), levels = area.names)
levels(ar)
levels(ar)[1] <- "90e2"
area.names <- levels(ar)[c(13, 1:12)]
levels(ar)
Section 4.2

attach(rawpsl.df)

?tapply

tapply(perseat, area, mean)
tapply(perseat, area, range)
twofacs <- list(area, factor(month))
tapply(perseat, twofacs, mean)
Section 5.1

An array can be considered as a multiply subscripted collection of data entries, for example numeric.

A $4 \times 5 \times 3$ array, 3-dimensional

```r
?array

x <- 1:60

z <- x

dim(z) <- c(4, 5, 3)

dim(z)
```
All elements of an array are of the same mode and type
Section 5.2

Reference individual elements of the array using indices in each dimension

\[ z[4, 2, 1] \]

Vector indexing

\[ z[c(2:4), 4:5, c(1, 3)] \]

Empty indexing

\[ z[, , 1] \]

Same as

\[ z[1:4, 1:5, 1] \]

Omit \[ z[-1, ,] \]
Extra

dimnames(z) <- list(dim1 = NULL, dim2 = NULL, dim3 = NULL)

dimnames(z)

dimnames(z) <- list(
dim1 = as.character(1:4),
dim2 = as.character(1:5),
dim3 = as.character(1:3)
)
Section 5.2

z2 <- z

z2[1,1,1] <- pi

print(pi, digits = 20)

?print.default

class(z2)

typeof(z2)

mode(z2)

str(z2)

summary(z2)

summary(as.character(z2))
Section 5.4

Do the same with `array()`

```r
x <- 1:60

z <- array(x, dim=c(4,5,3))
```

The repeat-to-fill convention

```r
allzero <- array(0, dim=c(4,5,3))
rep01 <- array(0:1, dim=c(4,5,3))
```
Element by element operations for the same \texttt{dim()}. Result has the same \texttt{dim()}.

\begin{verbatim}
z + rep01 
z * rep01 
z == rep01*z 
2*z + rep01 + 1000
\end{verbatim}
Can also have mixed arrays

\[ z + \text{rep01} \]

Rules

Any short vector operands are extended by recycling their values until they match the size of any other operands.

As long as short vectors and arrays only are encountered, the arrays must all have the same dim attribute or an error results.

Any vector operand longer than a matrix or array operand generates an error.

If array structures are present and no error or coercion to vector has been precipitated, the result is an array structure with the common dim attribute of its array operands.
Generate a $4 \times 5$ array

\[ w \leftarrow \text{array}(1:20, \text{dim}=c(4,5)) \]

Extract elements $x[1,3], x[2,2]$ and $x[3,1]$ as a vector structure, and replace these entries in the array $x$ by zeroes

Use a $3 \times 2$ subscript array

\[ i \leftarrow \text{array}(c(1:3,3:1), \text{dim}=c(3,2)) \]

\[ w[i] \]

\[ w[i] \leftarrow 0 \]

\[ w \]
Arrays may be used in arithmetic expressions and the result is an array formed by element-by-element operations on the data vector.

The `dim` attributes of operands generally need to be the same.

This becomes the dimension vector of the result.

A, B, and C have the same `dim`.

\[ D \leftarrow 2 \times A \times B + C + 1 \]

has this `dim` and operations are element by element.

Precedence is an issue.
Section 5.4

Precedence

The expression is scanned from left to right

Any short vector operands are extended by recycling their values until they match the size of any other operands

As long as short vectors and arrays only are encountered, the arrays must all have the same dim attribute or an error results

Any vector operand longer than a matrix or array operand generates an error

If array structures are present and no error or coercion to vector has been precipitated, the result is an array structure with the common dim attribute of its array operands
Section 5.5

Outer product of two vectors

\[ u = (u_1, \ldots, u_m) \ (1 \times m) \text{ vector} \]
\[ v = (v_1, \ldots, v_n) \ (1 \times n) \text{ vector} \]
\[ u'v = [o_{ij}] (m \times n \text{ matrix}) \]
\[ = [u_i v_j] \]
?outer

u <- 1:10

v <- rep(1:2, times=6)

dim(u)

dim(v)

dim(outer(u,v))

dim(outer(v,u))

outer(v,u)

outer(u,v,"+")
The multiplication function can be replaced by an arbitrary function of two variables.

Suppose we want to evaluate the function

\[ f(x, y) = \frac{\cos(y)}{1 + x^2} \]

over a regular grid of values with \( x \) and \( y \) coordinates.

```r
funongrid <- function(x, y) cos(y)/(1 + x^2)
x <- 0:5
y <- 2*pi*seq(0,1,by=.25)
z <- outer(x,y,funongrid)
```
x <- 1:60

z <- array(x, dim=c(15,4))

?matrix

z <- matrix(x, ncol = 4)

Transpose of matrix

t(z)

aperm(z, c(2,1))
Section 5.6

?aperm

Generalized transpose

x <- 1:60

z <- array(x, dim=c(4,5,3))

dimnames(z) <- list(
dim1 = as.character(1:4),
dim2 = as.character(1:5),
dim3 = as.character(1:3)
)

aperm(z, c(2,1,3))
Matrix multiplication \( \% \times \% \)

\[
A \leftarrow \text{matrix}(1:60, \text{ncol} = 4)
\]

\[
B \leftarrow \text{matrix}(1:60, \text{nrow} = 4)
\]

\[
B \% \times \% A
\]

\[
A \% \times \% B
\]
A <- matrix(1:25, ncol = 5)
x <- rep(c(0,1),length=5)
dim(x)
x %*% A
A %*% x
matrix(x)
x <- matrix(rep(0,1,length=5),nrow=1)
x %*% A
A %*% x
x %*% A %*% x
x %*% A %*% x
x %*% A %*% x
x %*% A %*% x
x %*% A %*% x
The function

\texttt{crossprod()} forms crossproducts.

It is the same as

\texttt{t(X) \%\*\% y}

but the operation is more efficient.

If the second argument to \texttt{crossprod()} is omitted it is taken to be the same as the first.
Diagonal of \( A \)

\[
A <- \text{matrix}(1:25, \ ncol = 5) \\
\text{diag}(A)
\]

A diagonal matrix

\[
x <- 1:10 \\
x <- \text{matrix}(1:60, \ ncol = 4) \\
\text{diag}(x)
\]

What?

\[
\text{What?}
\]
A <- matrix(rnorm(16), ncol = 4)
x <- rnorm(4)
b <- A %*% x

Solve for x

solve(A,b)

A <- matrix(c(1,0,1,0), nrow = 2)
b <- rnorm(2)
solve(A,b)
Least squares fit

\[
X \leftarrow \text{matrix}(\text{rnorm}(1000), \text{ncol} = 4)
\]

\[
y \leftarrow \text{rnorm}(250)
\]

\[
yfit \leftarrow \text{lm.fit}(X, y)
\]

\[
yfit2 \leftarrow \text{lm}(y \sim X)
\]
Section 5.8

X1 <- matrix(rnorm(100), ncol = 4)
X2 <- matrix(rnorm(80), ncol = 4)
rbind(X1, X2)
rbind(1:10, 11:20)

X1 <- matrix(rnorm(100), nrow = 4)
X2 <- matrix(rnorm(80), nrow = 4)
cbind(X1, X2)
cbind(1:10, 11:20)
Section 5.9

Concatenate vectors

c(1:10, 24:32)

Turn arrays into vectors

X1 <- matrix(rnorm(100), ncol = 4)
attach(rawpsl.df)

ls()

table(area)

table(area, price)

table(price, area)
Section 6.1 and 6.2

Lst <- list(name="Fred", wife="Mary", no.children=3, child.ages=c(4,7,9))

Lst[[1]]

Lst[1]

Lst["name"]

Lst[["name"]]

Lst$name

Lst$na

Lst[c("name","child.ages")]

attributes(Lst)

names(Lst)
Add to list

```r
Lst[[5]] <- 1:5
names(Lst)[5] <- "vector"
```

Remove element of list

```r
Lst$vector <- NULL
Lst <- Lst[1:4]
```
Section 6.1 and 6.2

What will happen here?

Lst[c("name","wrong.name")]
Lst[c("name","child.ages")]
Lst[[c("name","child.ages")]]
Lst[5] <- list(1:5)
Lst[[5]] <- list(1:5)
Lst[[5]] <- list(vector=1:5)
is.vector(Lst)
Lst$name
Section 6.3

Concatenate lists

```r
Lst2 <-
list(fanofteams = c("Bears","Bulls","Yankees"))
c(Lst,Lst2)
c(Lst2,Lst)
```

The apply functions

```r
lapply()
sapply()
```
> names(rawpsl.df)
[1] "day"       "month"     "year"      "date"      "num"
[8] "row"       "total"     "perseat"   "price"

How many ways can we get the variable *perseat* in one line?
Section 6.3

rawpsl.df[, "perseat"]

rawpsl.df[, 10]

rawpsl.df$perseat

attach(rawpsl.df); perseat

rawpsl.df[, -c(1:9, 11)]
Loops

```r
x <- 1:5
for (n in x) print(n^2)

for (n in x) {
  n2 <- n^2
  print(n2)
}

cumsum(x)
```
cusu <- x[1]
for (i in 2:length(x)){
cusu <- c(cusu, cusu[i-1]+x[i])
}
Eliminate first value of cumulative sum

cusu <- NULL
for (i in 2:length(x)){
cusu <- c(cusu, cusu[i-1]+x[i])
}

if-else

if(r ==4) {

x <- 1

} else {

x <- 3

y <- 4

}
Conditional

```r
ifelse()
x <- 1:10
y <- ifelse(x %% 2 == 0, 5, 12)
y
```
Loops and Conditionals

while

i <- 1

while (i <= 10) i <- i + 4

i
Loops and Conditionals

repeat
i <- 1
repeat{
  (i <- i + 4)
  if (i > 10) break
}
i
Operators

```
x + y
x - y
x * y
x / y
x ^ y
x %% y
x %/% y
x == y
x <= y
x >= y
x & y
x | y
!x
x && y
x || y
A %*% B
```
Creating a Function

oddcount <- function(x) {
  k <- 0
  for(n in x) {
    if (n %% 2 == 1) k <- k + 1
  }
  return(k)
}

Can replace return(k) with k

Where is k?
Avoid Loops if Possible

oddcnt2 <- function(x) sum(x %% 2 == 1)
Parameters and Variables

\[ rm(x, y) \]

\[ z \leftarrow \pi \]

\[ f \leftarrow \text{function}(x) \{ \]
\[ y \leftarrow 2 \times x \]
\[ \text{print}(x) \]
\[ \text{print}(y) \]
\[ \text{print}(z) \]
\[ \} \]

\( x \) is a formal parameter

\( y \) is a local variable

\( z \) is a free variable
f <- function(x) {
  y <<- 2*x
  print(x)
  print(y)
  print(y)
  print(z)
}

or use

y <- assign(2*x)

Such a side effect is not a good practice.
f <- function(x = 1, y = 1, z = 1) x + cos(y) + exp(z)

Can call by position or name; lazy evaluation

These are equivalent

f(1, pi, 2)
f(z = 2, y = pi)
f(, pi, 2)
Start-up Function

Function (.First) is executed at R start-up

.First <- function() library(lattice)
g <- function(x) {
  return(x+1)
}

class(g)

body(g)

formals(g)

attributes(g)
Looking at Functions

g
page(g)
vi(g)
edit(g)
page(panel.xyplot)
Functions Can be Arguments to Other Functions

?tapply
page(tapply)
vi(tapply)
if(?)
class(g)

typeof(g)

g

sum()

mean()
Environments

Collection of objects present at the time function was written.

\texttt{environment}(\texttt{mean})

\texttt{environment}(\texttt{g})

\texttt{g()} is at the top level, interpreter command prompt

\textbf{R output:}

\texttt{R\_GlobalEnv}

\textbf{In R code}

\texttt{.GlobalEnv}
Local and Global

w <- 6
f <- function(y) {
d <- 8
h <- function() {
  return(d*(w+y))
}
return(h())
}

ls()
ls.str()
h()
Local and Global

Global to $f()$

$w$

Local to $f()$

$h()$ $d$

Global to $h()$

$d, y, w$

We have a hierarchy and inheritance here

$h()$ searches “up” the call chain to get variables that it does not see locally

If we call $f()$ multiple times, $h()$ goes in and out of existence
Environments

`ls()` within a function returns the names of the current local variables

`ls(envir=parent.frame(n=1))` specifies how many frames to go up the call chain
Environments

w <- 6
f <- function(y){
d <- 8

h <- function(){
  return(d*(w+y))
}
return(h())
}
Environments and the Call Chain

```r
w <- 6
f <- function(y){
d <- 8
h <- function()
print(environment())
return(d*(w+y))
}
return(h())
```