# Pedigree: R is Public Domain Version of S

S 1999 Winner of the Association For Computing Machinery Software System Award

Other winners include: Unix (1983), TeX (1986), Smalltalk (1987), PostScript (1989), TCP/IP (1991), Remote Procedure Call (1994), World-Wide Web (1995), Tcl/tk (1997), Apache (1999), Java (2002), MAKE (2003), ...

“will forever alter the way people analyze, visualize, and manipulate data . . . S is an elegant, widely accepted, and enduring software system, with conceptual integrity, thanks to the insight, taste, and effort of John Chambers.”

# An Interactive Language for Data Analysis and Methodological Development

Much more efficient programming with the data than a lower level language

Can very easily tailor the analysis to the data, not just use package stuff

Fast prototyping of new methods

After development of a method in R, to get speed, rewrite in C and plug into R

# Support for R

Vast development community, more than any other software system for data analysis

More analytic statistical, machine learning, and visualization methods and any other system for data analysis

R core team is top notch

Number of R CRAN packages: 7069

# Popularity

Most used system for data analysis, overall, across all technical research communities in which data are analyzed

Very widely used in industry and government

It’s free

# Why Not SAS?

Most revenue of a company selling data analysis software

Many packages that are very well done, e.g., random effects models

Historically could handle larger datasets than the competition, which is why it became dominant.

But as a programming language for data analysis, very poor

## Sample Session

Quit R. It will ask what to do with the work space.

```
q()
```

Assign 1 to `a`

```
a <- 1
```

Print `a`. `a` is an object and `print()` is a generic function for the method print.

```
print(a)
```

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

```
a
```

R internal storage mode of an object.

```
typeof(1)
```

The class of an object.

```
class(b)
```

# Sample Session

Lists attached packages and objects in search order. Each is an environment.

```
search()
```

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

```
ls()
```

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

```
objects(1)
```

Lists objects in 6th environment in search list.

```
objects(6)
```

# Sample Session

Create vector of integers 1 to 10 and assign to `a`.

```
a <- 1:10
```

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

```
b <- rnorm(10)
```

Concatenate `a` and `b`.

```
c(a,b)
```

Vector from 1 to 3 in steps of 0.5.

```
seq(from = 1, to = 3, by = 0.5)
```

## Sample Session

Length of object.

```
length(b)
```

Maximum of object.

```
max(b)
```

Sort object.

```
sort(b)
```

Reverse order of sorted values of object.

```
rev(sort(b))
```

Range of object.

```
range(b)
```

# Sample Session

HTML interface to on-line help using a web browser available on your machine

```
help.start()
```

Help for a function (and other things) in R console window

```
?rnorm
```

# Sample Session

Attach package lattice graphics to search list.

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

Create two vectors of 100 random normals.

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

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

```
xyplot(y ~ x)
```

# Sample Session

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()
```

# Sample Session

Set graphics device to pdf and write to file.

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

Assign plot object to `yvsx`.

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

Write object to file.

```
yvsx
graphics.off()
```

# Sample Session

Replicate 1 to 10, 10 times.

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

Make `grps` a factor

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

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

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

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

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

# Sample Session

Set graphics device to pdf and write to file.

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

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

graphics.off()
```

# Sample Session

Make a data frame of three columns, x and y

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

Where is `x`?

```
find("x")
```

## Sample Session

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

```
attach(randata.df)

search()

find("x")

rm(x,y,grps)

find("x")
```

So you can do this.

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

To get rid of it.

```
detach()
```

This is a very powerful mechanism, but you must treat it with great respect because it can cause you great trouble.

## Sample Session

Another way to get at x and y

```
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()
```

## Sample Session

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()
```

# Sample Session

Fit a simple linear regression and look at the analysis

```
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)
```

## Sample Session

psl.csv on course web site.

```
?read.csv
```

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

```
rawpsl.df$perseat
```

```
rawpsl.df[,"perseat"]
```

Avoid this.

```
rawpsl.df[,10]
```

## Section 2.1 and 2.2

Assignment.

```
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.

```
log, exp, sin, cos, tan, sqrt, length, min, max, var
```

```
sum((x-mean(x))^2)/(length(x)-1)
```

Complex numbers.

```
sqrt(-17)
```

```
sqrt(-17+0i)
```

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
```

## Section 2.4

Logical operators.

`<, <=, >, >=, ==, !=`

Logical expressions result in logical objects. Scalar values are `TRUE (T)`, `FALSE (F)`

```
x == y
```

```
x <- 1:15
y <- 15:1
logical1 <- x < y
logical2 <- x == y
typeof(logical2)
class(logical2)
mode(logical2)
```

## Section 2.4

And.

```
(logical1) & (logical2)
```

Or.

```
(logical1) | (logical2)
```

Not.

```
!(logical1)
```

Coercion to numeric: F to 0 and T to 1

```
sum(logical2)
```

## Section 2.5

Missing elements of vector: NA

Vector can be numeric, character, or logical

```
z <- c(1:3,NA)
```

```
logicalz <- is.na(z)
```

```
logicalz
```

```
z == NA
```

NA in a vector will defeat a lot of functions.

```
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")
```

## Section 2.6

Character strings.

```
myname <- "Francesca"
```

```
myname <- ’Francesca’
```

But print method uses `""`

```
myname
```

Watch out. This is different.

```
myname <- " 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.

```
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`

# Section 2.7

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 <- 1:20
```

Values in the index vector must lie in the set

`1, 2, .   .   .  , length(x)`.

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

```
c("x","y")[rep(c(1,2,2,1), 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.

```
x <- 1:20

x[x == 7]

x <- c(1:10,NA)

x

x[!is.na(x)]

(x+1)[(!is.na(x)) & x>8]
```

## 4. A vector of character strings.

Vector object has a names attribute to identify its components.

```
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.

```
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 a object

Atomic objects are all of same `mode`: numeric1, complex, logical, character and raw

Vectors must have the same mode

`NA` is an exception

```
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).

```
z <- 0:9

digits <- as.character(z)

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 recursiverather than atomic structures since their components can themselves be lists in their own right.

```
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 <- numeric()
length(e)
mode(e)
```

This is “interesting”

```
e[3] <- 17
e
```

You can truncate the length of an object.

```
alpha <- 1:10
alpha <- alpha[2 * 1:5]
alpha
length(alpha) <- 3
```

```
alpha

length(alpha) <- 10

alpha
```

## Section 3.3

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

```
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

```
attr(rawpsl.df,"names")
```

Define or change an attribute of an object

```
a <- 1:10
```

```
attr(a,"ownerofa") <- "myname"
```

Note the print method

```
a
```

Compare with

```
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)
```

```
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

```
?array

x <- 1:60

z <- x

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

dim(z)
```

## Section 5.1

```
class(z)

typeof(z)

mode(z)

str(z)

summary(z)

?mode

?typeof
```

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 = NU

dimnames(z)

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

```
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()`

```
x <- 1:60
z <- array(x, dim=c(4,5,3))
```

The repeat-to-fill convention

```
allzero <- array(0, dim=c(4,5,3))
rep01 <- array(0:1, dim=c(4,5,3))
```

## Section 5.4

Element by element operations for the same `dim()`. Result has the same `dim()`.

```
z + rep01

z * rep01

z == rep01*z

2*z + rep01 + 1000
```

Can also have mixed arrays

```
z + 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

## Section 5.3

Generate a $4 \times 5$ `array`

```
w <- array(1:20, dim=c(4,5))
```

```
w
```

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 <- array(c(1:3,3:1), dim=c(3,2))
```

```
i
```

```
w[i]
```

```
w[i] <- 0
```

```
w
```

## Section 5.4

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 <- 2*A*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

Outer product of two vectors

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

```
?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,"+")
```

## Section 5.5

The multiplication function can be replaced by an arbitrary function of two variables

Suppose we want to evaluate the function

$$f(x, y) = \cos(y)/(1 + x^2)$$

over a regular grid of values with $x$ and $y$ coordinates

```
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)
```

## Section 5.6

```
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))
```

## Section 5.7

Matrix multiplication %*%

```
A <- matrix(1:60, ncol = 4)

B <- matrix(1:60, nrow = 4)

B%*%A

A%*%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
```

# Section 5.7

The function

```
crossprod()
```

forms crossproducts.

It is the same as

```
t(X) %*% y
```

but the operation is more efficient.

If the second argument to `crossprod()` is omitted it is taken to be the same as the first.

Diagonal of A

```
A <- matrix(1:25, ncol = 5)
diag(A)
```

A diagonal matrix

```
x <- 1:10
diag(x)
```

What?

```
A <- matrix(1:60, ncol = 4)
diag(A)
```

```
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)
```

## Section 5.7

Least squares fit

```
X <- matrix(rnorm(1000), ncol = 4)
y <- rnorm(250)
yfit <- lm.fit(X, y)
yfit2 <- lm(y~X)
```

```
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)
```

## Section 5.10

```
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)
```

# Section 6.1 and 6.2

Add to list

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

Remove element of list

```
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] <- 1:5

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

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

c(Lst,Lst2)

c(Lst2,Lst)
```

The apply functions

```
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?

```
rawpsl.df[,"perseat"]

rawpsl.df[,10]

rawpsl.df$perseat

attach(rawpsl.df); perseat

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

# Loops

```
x <- 1:5
for (n in x) print(nˆ2)

for (n in x){

n2 <- nˆ2

print(n2)
}

cumsum(x)
```

# Loops

```
cusu <- x[1]

for (i in 2:length(x)){

cusu <- c(cusu, cusu[i-1]+x[i])
}
```

# Loops

Eliminate first value of cumulative sum

```
cusu <- NULL

for (i in 2:length(x)){

cusu <- c(cusu, cusu[i-1]+x[i])
}
```

# Conditional

```
if-else

if(r ==4){

x <- 1

} else
{

x <- 3

y <- 4
}
```

# Conditional

```
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

```
oddcount2 <- function(x) sum(x %% 2 == 1)
```

## Parameters and Variables

```
rm(x,y)
z <- pi
f <- function(x) {
y <- 2*x
print(x)
print(y)
print(z)
}
```

`x` is a formal parameter

`y` is a local variable

`z` is a free variable

## Superassignment

```
f <- function(x){
y <<- 2*x
print(x)
print(y)
print(z)
}
```

or use

```
y <- assign(2*x)
```

Such a side effect is not a good practice.

# Evaluation

```
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)
```

# Functions are Objects

```
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 and Typeof

```
class(g)

typeof(g)

g

sum()

mean()
```

## Environments

Collection of objects present at the time function was written.

```
environment(mean)

environment(g)
```

`g()` is at the top level, interpreter command prompt

R output:

```
R_GlobalEnv
```

In R code

```
.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 inheritence 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



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