R

http://r.seanrmccorkle.org

http://www.r-project.org

Data Types

Integer	`123 -11 0`	+ − × ÷ √
Floating point	`6.022e23` (6.022×10^{23})	+ − × ÷ √
Boolean	`True False`	and or not
Character strings	`"Hello"`	concatenation substring

Any reasonable computer system encodes all these into binary

High level programming languages save us the labor of writing binary instructions the computer understand

Some high level languages, like C and C++, are compiled. A translation from the language to machine binary

Another approach is interpreters: the language instructions are parsed and then directly executed by the interpreter

R is a high level programming language

It has a very large collection of functions for mathematics, statistics, and graphics

It is an interpreter - a program that reads what you type and then does stuff according to what you type

Loop: Read - evaluate - print

```
R version 2.4.1 (2006-12-18)
Copyright (C) 2006 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

>
```

In R, instructions are expressions

math-like combinations of data and operations

```
1
```

```
1 + 2
```

```
2.2 * (1.5 + 7.3)
```

* multiply

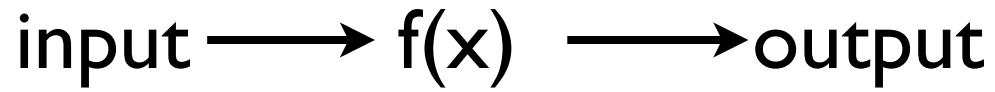
```
3.4^2
```

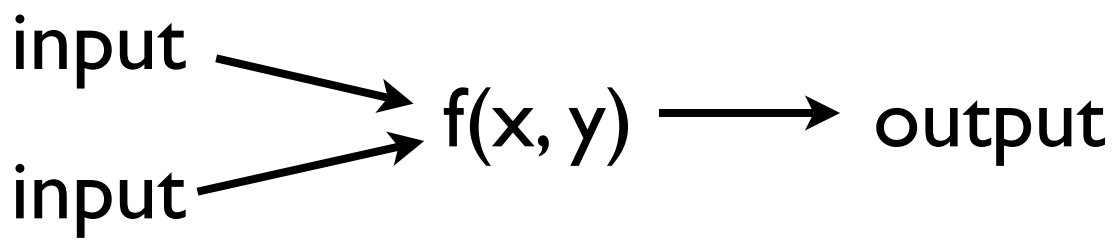
```
3.4^2
```

^ to the power of

parenthesis change the normal order of precedence

functions






```
sqrt( 5 )
```

```
exp( -1 )
```

```
sin(2.5)
```

```
log10( 25.2 )
```

Variables

```
x <- 1 + 2

y <- exp(2.3/5) * (5.3-1)^0.5

 ls()

 rm()
```

Vectors

```
v <- c( 3, 4, 17, 29 )
```

```
n:m
```

automatically generates a vector

```
c(n, n+1, n+2, ..., m )
```

```
1:12
```

```
sqrt( 1:12 )
```

```
v + u
```

what if the lengths don't match?

```
2 * v
```

Boolean

Data type allows only two values:

`TRUE` **`FALSE`**

`T` **`F`**

Operations

`&&`	and
`||`	or
`!`	not

```
(! T) && (F || T)
```

Relational operations

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

numeric rel-op numeric ⟶ boolean

```
5 > 2 ⟶ TRUE
```

```
(5 > 2) || (25 < 10)
```

What about vectors?

```
x <- c( 3, 5, 10, 7, 5)
y <- c( 1, 7, 9, 11, 5)
```

```
x  <  y        x  <  2
```

You can have boolean vectors

```
v <- c( T, F, T, F )
u <- c( F, F, T, T )
```

Special operators for boolean vectors `&` `|` `!`

```
u & v      u | v      ! u
```

these operators yield vectors of booleans

Subscripting vectors

```
v <- c( 5, 2, 1, 17, 7, 13, 21, 20)
```

```
        1  2  3   4  5   6   7   8
```

```
  v[3]    v[5]
```

R has a very powerful concept of subscripting,
which includes vectors as as subscript

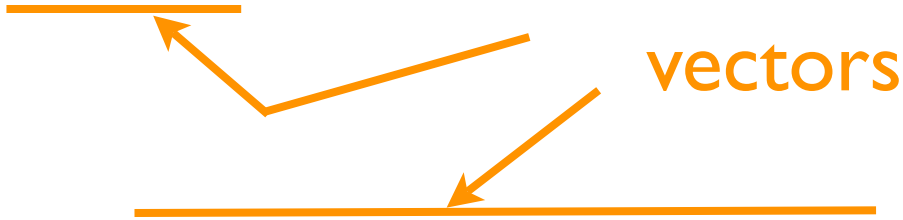
```
v[ c( 3, 5, 8) ]

v[ -c( 3, 5, 8) ]

v[ c( T, F, T, T, F, T, T, F) ]
```

```
v <- c( 5, 2, 1, 17, 7, 13, 21, 20)
```

```
v[ v > 5 ]
```




```
v[ v >= 5 & v < 20 ]
```

Useful for making *cuts* on data.
Remove unwanted data entirely from the vector.

More vector tricks

boolean * numeric

boolean + numeric

`TRUE` is converted to **`1`**,
`FALSE` is converted to **`0`**

```
v <- c( 5, 2, 1, 17, 7, 13, 21, 20)

m <- v > 5

v * m
```

m is a mask - converts unwanted data to zero

```
x <- c( 5, 7, 10, 17, 23, 33, 37, 42)
```


```
n <- length( x )
```

```
1:n
```

```
1:(n-1)
```

```
2:n
```

```
x[2:n] - x[1:(n-1)]
```

matrices

```
m <- matrix( 1:12, 3, 4 )
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

one element `m[3,2]`

a row `m[2,]`

a column `m[,3]`

some functions which take vector arguments but return a scalar

`sum()`

`summary()`

`mean()`

`sd()`

`max()`

`min()`

(what will **`sum( c(T,F,T,F,T) )`** do?)

Character strings

```
s <- “this is a string"
```

```
paste()
```

```
substr()
```

```
nchar()
```

help function

```
> help(replicate)
                        .
                        .
replicate( n, expr, simplify = TRUE )
                        .
                        .
```

The arguments n and expr are required,
but the argument simplify is not.
If not given, it defaults to T

Plotting

```
x <- 1:10

y <- sqrt(x)

plot( x, y )

plot( y )

plot( x, y, xlim=c(0,10) )

plot( x, y, type="l" )
```

Reading data

text file test.txt

```
x     a      b       quality
1.0   3.4    10.1    good
3.0   5      20.3    no_so_much
3.5   2.1    29.9    good
4.0   6.1    40.0    good
7.3   9.4    50.1    no_so_much
```

```
dat <- read.table("test.txt", header=T )
```

skips header, uses header as column names

```
dat <- read.table("test.txt", skip=1 )
```

skips header, names columns V1, V2, ...

```
dat <- read.table("test.txt")
```

treats header line as data, makes all columns character data

if file is in different directory use full path name

```
dat <- read.table("/dir/dir/test.txt", ...)
```

or change working directory

```
setwd("/dir/dir")
```

or use choose file dialog (returns full pathname)

```
file.choose()
dat <- read.table( file.choose(), ...)
```

```
> tab<-read.table("test.dat",header=T)
> tab
   x   a    b    quality
1 1.0 3.4 10.1        good
2 3.0 5.0 20.3 no_so_much
3 3.5 2.1 29.9        good
4 4.0 6.1 40.0        good
5 7.3 9.4 50.1 no_so_much
```

creates a data frame, like a matrix with extra features

```
tab[2,]     tab[4,]     tab[c(1,3,4),]
```

rows

```
tab[,1]     tab[,3]     tab[,c(1,3,4)]

tab$x     tab["x"]     tab[c("x","a","b")]
```

columns

histogram

```
hist( y )

hist( y, breaks=100 )

hist( y, breaks=0:100 )

h <-hist( y, breaks=0:100, plot=F )
```

fits

```
> tab
    x   a    b    quality
1 1.0 3.4 10.1        good
2 3.0 5.0 20.3 no_so_much
3 3.5 2.1 29.9        good
4 4.0 6.1 40.0        good
5 7.3 9.4 50.1 no_so_much
```

```
> fit<- lm( a ~ x, tab)
> fit
> plot(tab$x,tab$a)
> abline(coef=fit[[1]])
```

programming

```
f <- function(x) sqrt(x) + 1

d <- function(x,y)
    {
     s <- sum( (y-x)^2 )
     sqrt(s)
    }

f(5)    d(7,3)
```

```
{
    expr
    expr
    expr
}
```

Sequential evaluation

```
if ( boolean-expr )
    expr
```

Conditional evaluation

```
if ( boolean-expr )
    expr
else
    expr
```

```
while( boolean-expr )
    expr
```

repeated evaluation

```
for ( i in vec )
    expr
```

How do you load R functions already written?

```
source( "file" )
```

```
sink( "file" )
```
redirect output to a file

```
sink()
```

q()