Practical Programming

Documentation:

R homepage:

```
http://www.r-project.org/
```

R introduction:

```
http://cran.r-project.org/doc/manuals/r-release/R-intro.pdf
```

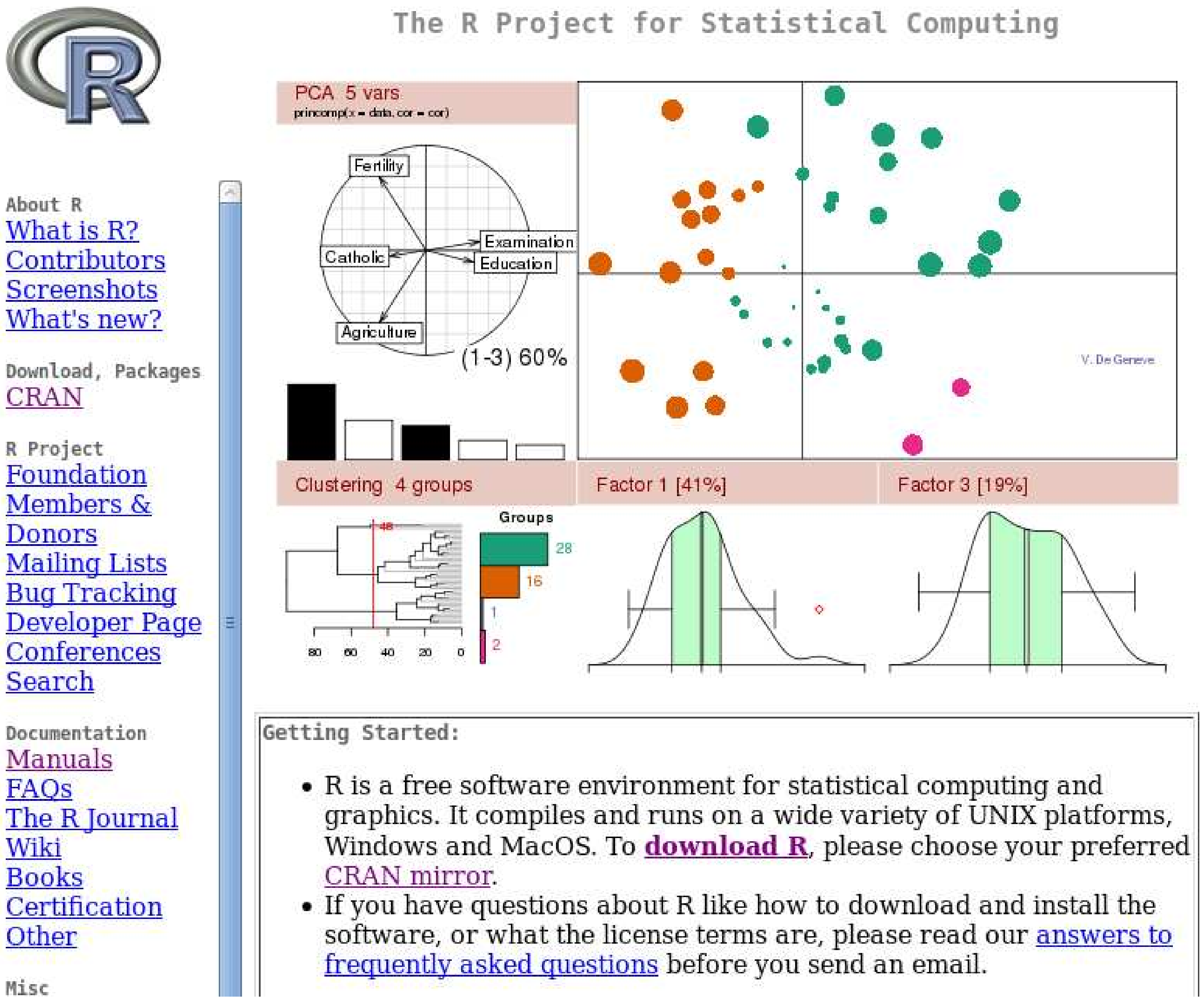
Quick-R:

```
http://www.statmethods.net/
```

Online course: Try-R

```
https://www.codeschool.com/courses/try-r
```

Download from http://cran.r-project.org/



The R Project for Statistical Computing

About R
What is R?
Contributors
Screenshots
What's new?

Download, Packages
CRAN

R Project
Foundation
Members &
Donors
Mailing Lists
Bug Tracking
Developer Page
Conferences
Search

Documentation
Manuals
FAQs
The R Journal
Wiki
Books
Certification
Other

Misc

Getting Started:

- R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To **download R**, please choose your preferred CRAN mirror.
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

Computer Programming

Programming The process of writing a computer program.

Program A sequence of instructions used to direct a computer to perform some task. A program is 'software' (or 'code'). The computer on which it runs is 'hardware'.

Hardware Task-generic. (usually)

Software Task-specific. (usually)

Programming Languages

Computers obey instructions in ’machine-language’. Binary code (1s/0s).

Computers are usually programmed in ’high-level languages’ - human readable languages which may be translated into machine language.

Compiler Translates a whole program into machine language, so the instructions may be executed by the computer at some time in the future. Fast, less flexible.

Interpreter Interprets one instruction at a time in terms of machine-language instructions, obeying those instructions as they are interpreted. Slow, more flexible.

Programming Languages

C, C++, Python, Java, C#, Basic, Fortran, Pascal, Cobol, Ada, Perl, TCL...

R

Combines both a programming language and a data analysis package - can be used for either or both. If you can do it in a spreadsheet, you can probably do it better in R.

Increasing use in scientific community.

http://www.r-project.org/

Analogy: driving directions from M1

```
1. Join A64 eastbound
2. Proceed to junction with A19
3. If there are roadworks on the A19:
       Carry on to the junction with Hull Road
       Turn left on to Hull Road
       Turn left at next roundabout onto field lane
   else:
       Turn left onto A19
       Turn right onto Heslington lane
4. Continue to university roundabout
5. For each carpark in chemistry, biology, comp-sci:
       If there is space in carpark:
           Park in carpark. You’re done.
```

Mostly sequential. Some instructions conditional, some repeated. Symbols.

Writing a program

- Write a sequence of instructions using an editor.
- Save those instructions to a file.
- Run the program in the R interpreter.

Elements of a program

- Constants/Literals e.g. `4, 'Fred'`
- Variables e.g. `n, name`
- Expressions e.g. `a+b, a*b`
- Statements e.g. `message(a+b)`
  - Assignment e.g. `c = a+b`
  - Control flow e.g. conditions, loops
- Declarations
  - Functions

Constants/Literals

Literals are values included in a program instruction.

- Integer numbers... 1, 2, -1
- Floating point numbers... 1.0, -0.001, 3.1415926
- Character strings... ’hello world’, ’Fred’, ’42’
- Not available... NA

Note:

Strings use single quotes - ’ (Under @ on a UK keyboard)

or double quotes - " (shift-2)

But **not** back-quotes - ‘

A number is different from the text representation of a number.

Variables

Variable: A named storage location that can contain data that can be modified during program execution.

Variables store the information upon which a program is working. The information in a variable may be operated upon by the program to produce new information. 'Pigeon holes'.

Variables:

- Have a name... `a, b, name, bank.balance`
- Have a value... `3, 4.5, 'Fred'`
- The value can change as the program proceeds... `x = x + 1`

Note: Variable names can be any letter followed by any combination of letters, numbers and dots.

Expressions

Expressions produce new values from old ones.

Expressions combine variables and literals using operators. The result of the expression is determined by the values and operators used.

e.g.

```
1+2
a+b
((a*b)+2)/c
a
```

Note: `*` for multiply and `/` for divide.

Statements

A statement or instruction is a single step of a computer program.

- Built-in statement (comment, message)
- Assignments (including null assignments)
- Control-flow statements (loops, conditions)
- Declarations (functions, classes)

Each statement is placed on a new line. Spaces are generally ignored, but may be used for clarity.

Statements: Built-in

Comments: documentation for the programmer. The computer ignores comments.

Use #. The rest of the line is ignored.

```
# this is a comment - it does nothing.
```

Comments are vital for all but the most trivial programs.

Statements: Built-in

`message`

Display the result of an expression, e.g.

```
message(1+2)      # shows '3'
message(a)
message('a number: ', 3, ' ', a+b, ' ', c)
```

Or just give an expression:

```
1+2
a
```

demo

Statements: Assignment

An assignment stores the result of an expression in a variable.

e.g.

```
number.of.trains = 0
x = y
x = x + 1
area = pi * radius ^ 2
```

demo

Statements: Assignment

You can use either:

```
x = 1
```

or

```
x <- 1
```

The latter is probably clearer, but the former is easier to type and is standard across most computer languages.

Vectors and plotting

A vector can be used to hold multiple values of the same type (numeric, interger, character).

The different values are stored in numbered positions, and can be referred to by their ’index’ within the vector.

A vector can be created from a list of values, or with a sequence of numbers, or random numbers.

Vectors and plotting

Create a vector from a list of values:

```
x = c(1,2,3)
y = c('Tom','Dick','Harry')
```

Display the vector:

```
x
```

Or just one member:

```
x[1]
```

Vectors and plotting

Find the length of a vector:

```
length(x)
```

Vectors and plotting

Maths with vectors:

```
a = c(1,2,3)
b = c(4,5,6)
c = a+b
d = a^2
```

Vectors and plotting

Other ways to create a vector:

```
x = 1:5                # sequence 1,2,3,4,5
y = seq(1.0,2.0,0.5)   # sequence 1.0,1.5,2.0
z = rep(1,4)           # repeat 1,1,1,1
a = runif(6)           # random uniform dist: 6x 0..1
b = rnorm(6)           # random normal dist: 6 vals
c = sample(6,3)        # 3 numbers from 1..6
d = sample(6,9,replace=TRUE)
```

Vectors and plotting

Plot a histogram:

```
x = rnorm(10000)
hist(x)
```

Plot a graph:

```
y = rnorm(10)
plot(y)
```

Vectors and plotting

Line graph:

```
y1 = rnorm(10)
plot(y1,type='l')
```

Add another line:

```
y1 = runif(10)
plot(y1,type='l')
lines(y2)
```

Plot starts a new plot and sets the limits. Lines add lines.

Vectors and plotting

XY graph:

```
x = 1:10
y = xˆ2
plot(x,y,type='l')
```

Note: we can do maths on vectors.

Statements: Control-flow

Control-flow statements include statements which disrupt the sequential execution of instructions. They include conditions and loops.

They allow a 'group' of instructions to be executed conditionally or repeatedly. The group is indicated by enclosing it in braces (i.e. curly brackets: $\{\}$). *Exception: If the braces $\{\}$ are omitted, then the loop or condition contains a single statement.*

For clarity we indent the contents of the control flow statement.

Statements: Control-flow: Conditional

Conditionals (if-statements) allow groups of instructions to be executed dependent on the outcome of some test (the condition).

Conditions:

Equality `x == 1,   y == n`

Inequality `x != 1,   x != 2*y+1`

Greater/less `x < 5, i > j`

Greater/less-or-equal `x <= 5, i >= j`

Note: equality is double ==

Statements: Control-flow: Conditional

```
if (CONDITION)
  INSTRUCTION

if (CONDITION) {
  INSTRUCTION-GROUP
}

if (CONDITION) {
  INSTRUCTION-GROUP
} else {
  INSTRUCTION-GROUP
}
```

Statements: Control-flow: Conditional

Most complex form...

```
if (CONDITION){
  INSTRUCTION-GROUP
} else if (CONDITION) {
  INSTRUCTION-GROUP
} else {
  INSTRUCTION-GROUP
}
```

Statements: Control-flow: Conditional

e.g.

```
if ( x > 0 )
  sqrtx = sqrt( x )
else
  sqrtx = 0.0
```

or

```
if ( x > 0 ) {
  sqrtx = sqrt( x )
} else {
  sqrtx = 0.0
}
```

Statements: Control-flow: Loop

`for` loop

Loops a group of instructions, setting the loop variable to each value in turn from a list.

```
for (VARIABLE in START:FINISH)
  INSTRUCTION

for (VARIABLE in START:FINISH) {
  INSTRUCTION-GROUP
}

for (number in 1:5) {
  square = number ˆ 2
  message( number, ’ ’, square )
}
```

Statements: Control-flow: Loop

Advanced usage:

```
for (VARIABLE in SEQUENCE) {
  INSTRUCTION-GROUP
}

for ( x in seq(0.00,1.00,0.01) )
  message(x)

for ( name in c('John', 'Micheal', 'Graham', 'Eric') )
  message('Hello ', name)
```

Statements: Control-flow: Loop

The `for` loop and sequences

begin:end generates integers from begin to end

seq(begin,end) generates numbers from begin to end

seq(begin,end,step) goes in steps of ’step’

```
for (number in 1:5) {
  square = number ^ 2
  message( number, ' ', square )
}
```

demo

More on Loops

Loops can be ’nested’, one inside another. For each iteration of the outer loop, all iterations of the inner loop are performed.

```
for (i in 1:4)
  for (j in 1:3)
    message( ’Outer loop ’,i,’ Inner loop ’,j )
```

```
Outer loop 1 Inner loop 1
Outer loop 1 Inner loop 2
Outer loop 1 Inner loop 3
Outer loop 2 Inner loop 1
Outer loop 2 Inner loop 2
Outer loop 2 Inner loop 3 ...
```

More on Loops

`while` loop.

Like a repeated-if statement. The loop is executed as long as the condition evaluates as true.

```
while (CONDITION) {
  INSTRUCTION-GROUP
}
```

Can be used like a for-loop (don't do this!)

```
x = 0
while ( x < 10 ) {
  message(x)
  x = x + 1
}
```

More on Loops

`break` statement.

Causes an early exit from a `for` or `while` loop. Control passes to the first statement after the loop.

e.g.

```
for ( n in 1:10 ) {
  message( ’start of loop ’,n )
  if ( n == 5 )
    break
  message( ’end of loop ’,n )
}
message( ’done loop’ )
```

More on Loops

`next` statement. (REDUNDANT: use if)

Causes the rest of this loop iteration to be skipped. Control passes to the beginning of the loop for the next value in the list.

e.g.

```
for ( n in 1:10 ) {
  message( ’start of loop ’,n )
  if ( n == 5 )
    next
  message( ’end of loop ’,n )
}
message( ’done loop’ )
```

More on Expressions

Operators:

`^`	exponentiation (power). `a^b` is a^b
`-`	Unary minus: negation
`%%`	remainder
`*` `/`	times, divide
`+` `-`	plus, minus
`==` `!=` `<=` `<` `>=` `>`	comparisons
`!` `&&` `||`	logical not, and, or operators

All are ’binary operators’, i.e. they take two operands, one before and one after, except unary minus.

Comparisions give ’TRUE’ or ’FALSE’. Logical operators work on ’TRUE’ and ’FALSE’.

More on Expressions

Order-of-precedence: The order in which different operators in an expression are evaluated.

`^`	exponentiation (power). `a^b` is a^b
`-`	Unary minus: negation
`%%`	remainder
`* /`	times, divide
`+ -`	plus, minus
`== != <= < >= >`	comparisons
`! && ||`	logical not, and, or operators

Order of precedence is from top to bottom. For equal level, left to right in expression. Use (curved) brackets to override. e.g.

`a+b*c` is equivalent to `a + (b*c)`

`a^b+c` is equivalent to `(a^b) + c`

More on Expressions

Logical expressions:

Comparisons give a ’TRUE’ or ’FALSE’ result, e.g. ’==’ is TRUE if the expressions it compares are equal.

&&	gives TRUE if both of the expressions are also true.
||	gives TRUE if either of the expressions are true.
!	(unary) gives TRUE if its right-hand expression is false.

More on Expressions

Logical expressions: e.g.

```
if ( year %% 4 == 0 && year %% 100 != 0 )
  days.in.year = 366
else
  days.in.year = 365

if ( probability < 0.0 || probability > 1.0 ):
  message( ’Impossible probability’ )
```

More on Expressions

Integers and floating point....

Floating-point Any variable holding a number is by default a floating-point number.

Integers Any variable holding a number generated using : is an integer.

If an expression combines an integer and floating-point number, the integer is converted to floating point first.

`7 / 2`   gives `3.5`

`7 / 2.0` gives `3.5`

More on Expressions

Use `as.integer` and `as.numeric` to convert:

```
j = as.integer(i)
x = as.numeric(x)
```

These also convert strings to numbers:

`as.numeric('7.0')` gives `7.0`

`as.character` converts any variable to a string:

`as.character(7.0)` gives `'7.0'`

A number and its string representation are different things.

```
s = '7.0'
x = as.numeric(s)
```

Functions

The problem:

We may want to do a common task at several different places in a program. Writing the same code several times is wasteful and leads to errors.

Large and complex blocks of code are hard to understand and debug. Breaking a program down into smaller, simpler chunks gives greater clarity and allows each chunk to be debugged separately.

Functions

The solution:

Functions are named blocks of code which can be called by name. A set of variables (arguments) are passed to the function to provide the information it requires to perform its task. Another variable (or data structure) may be returned, containing the result of the task.

The variables passed to a function are called its *arguments*.

The variable returned from a function is called its *return value*.

Anything else done by the function is called a *side effect*.

Functions

Example (trivial): Hypotenuse of a right triangle.

```
c = sqrt( a^2 + b^2 )
message( c )
```

We can wrap the code in a function and give it a name:

```
hypotenuse = function() {
  c = sqrt( a^2 + b^2 )
  message( c )
}
```

Functions

We can then use the function whenever we need it. The ’()’ shows it is a function.

```
a = 3.0
b = 4.0
hypotenuse()
a = 5.0
b = 12.0
hypotenuse()
```

```
5.0
13.0
```

But: what happens if we want to do something with the result? (e.g. convert from mm to m or feed it into another calculation)

Functions

A function is more useful if it gives back an answer for further use, rather than printing it. We can then store it in a variable, use it in a calculation, or display it.

```
hypotenuse = function() {
  c = sqrt( a^2 + b^2 )
  return( c )
}

a = 3.0
b = 4.0
z = hypotenuse()
z = 1000.0 * z
```

Functions

Another problem: our function always uses the values of the variables `a` and `b`.

What happens if we want to calculate the hypotenuse of a triangle whose sides are `x` and `y`?

```
a = x
b = y
z = hypotenuse()
```

Inconvenient. (Even more so if a and b were already in use for something else).

Functions

So we use arguments:

```
hypotenuse = function( a, b ) {
  c = sqrt( a^2 + b^2 )
  return( c )
}
```

We can put the numbers we want to use as a and b in the parentheses when we call the function:

```
z  =  hypotenuse(x,y)
w  =  hypotenuse(u,v)
message( hypotenuse(3.0,4.0) )
```

Functions

Defining a function:

```
FUNCTIONNAME = function( ARGUMENTS ) {
  INSTRUCTION-GROUP
}
```

Where `INSTRUCTION-GROUP` usually ends with

```
  return( VALUE )
```

Calling a function:

```
VALUE = FUNCTIONNAME( ARGUMENTS )
```

Functions

```
# define a useful subroutine
factorial = function( n ) {
  nfact = 1
  for ( m in 2:n )
    nfact = nfact * m
  return( nfact )
}

# now the main program
for (i in 1:10)
  message( i, ’ ’, factorial(i) )
```

Functions

Scope:

The scope of a variable is the region of a program source within which it is defined. It cannot be accessed outside this region.

In R, the scope of a variable is the function in which it is defined. When the function finishes, the value is lost.

Arguments are also lost. Information is only saved through the return value.

Exception: Global variables. Do not use.

Both arguments and return value are optional.

Functions

Therefore you should always assume that the variables in a function *are completely different to* the variables in the calling program. The only communication is through arguments and return values, and these may have different names.

```
factorial = function( n ) {
  nfact = 1
  for ( m in 2:n )
    nfact = nfact * m
  return( nfact )
}
```

Note parameter n, returns nfact

```
for (i in 1:10)
  message( i, factorial(i) )
```

Note parameter i, returns to print

Functions

Functions can have any number of simple variables or lists as arguments:

```
# find the middle of three numbers
middle = function( x, y, z ) {
  if ( x >= y && y >= z )
    mid = y
  if ( y >= z && z >= x )
    mid = z
  if ( z >= x & x >= y )
    mid = x
  return( mid )
}

# main program
c = middle( 31.5, 30.2, 33.6 )
```

Functions

Functions can have any number of simple variables or lists as arguments:

```
# return the longest word in a list
longword = function( words ) {
  longest = ""
  for ( i in 1:length(words) )
    if ( nchar(words[i]) > nchar(longest) )
      longest = words[i]
  return( longest )
}

# main program
mylist = c( "John", "Fitzgerald", "Kennedy" )
x = longword( mylist )
```

Functions

What if there is an error in the arguments? One solution is to return the missing value ’NA’.

```
def squareroot( value ) {
  if ( value >= 0.0 )
    return(sqrt(value))
  else
    return( NA )
}
```

Can then check for NULL using is.na in the calling program.

```
y = sqrt(x)
if (is.na(y)) message( ’Error’ )
```

Functions

Functions with optional arguments:

When using standard libraries we will use some functions with optional arguments. Optional arguments may be specified by giving the name and value of the argument, otherwise they will take default values. e.g.

```
mean( x )
mean( x, na.rm=TRUE )
```

Functions

Remember:

A well defined function operates on data which is provided as *arguments* (i.e. in the parentheses), and produces a result in the form of a return value (not printed to the screen).

A well defined function is defined at the start of the program, before any of the main program code which calls it.

Built in Functions

Basic mathematics:

```
sin( 3.14159 / 4.0 )
```

```
0.707106312094
```

```
4.0 * atan( 1.0 )
```

```
3.14159265359
```

Note: trig functions are *always* in radians, not degrees.

Built in Functions

Basic mathematics:

Trig fns: `sin, cos, tan`

Inverse trig fns: `asin, acos, atan`

Exponential: `exp, log, log10`

Other: `sqrt`

Variables: `pi`

Built in Functions

Vectors:

Sum `sum(x)`

Mean `mean(x)`

Median `median(x)`

Std dev `sd(x)`

Min/max `min(x), max(x)`

More Data Structures

Array: A collection of data items distinguished by their indices (or ”subscripts”).

e.g. We could have an array of month names, called `monthNames`. This holds 12 values, which are accessed using an index.

e.g.

```
monthNames[1]
```

```
January
```

```
monthNames[3]
```

```
March
```

A vector is a type of array.

More Data Structures

2D arrays are called matrices:

```
x = matrix(0,2,3)
x
```

```
       [,1] [,2] [,3]
[1,]      0    0    0
[2,]      0    0    0
```

More Data Structures

```
x = matrix( runif(6), 2, 3 )  # random
x = matrix( 1:6, 2, 3 )       # sequence
x
```

```
      [,1] [,2] [,3]
[1,]     1    3    5
[2,]     2    4    6
```

```
x[1,3]
```

```
5
```

More Data Structures

N-dimensional arrays:

```
x = array(0,dim=c(3,4,5))
x[3,4,5]
```

(You won’t need these.)

More Data Structures

Lists:

Just like vectors, but can hold data of mixed types (including other arrays). (You won’t need these.)

More Data Structures: Data Frames

These behave like a table in a spreadsheet, containing a set of named columns, with each column holding a vector of values. (A data frame works like a list of vectors).

```
head(Formaldehyde)
```

```
carb optden
1 0.1 0.086
2 0.3 0.269
3 0.5 0.446
4 0.6 0.538
5 0.7 0.626
6 0.9 0.782
```

More Data Structures: Data Frames

We can access columns by name:

```
Formaldehyde$carb
```

```
[1] 0.1 0.3 0.5 0.6 0.7 0.9
```

```
Formaldehyde$optden
```

```
[1] 0.086 0.269 0.446 0.538 0.626 0.782
```

Or by number:

```
Formaldehyde[,2]
```

```
[1] 0.086 0.269 0.446 0.538 0.626 0.782
```

More Data Structures: Data Frames

We can access elements:

```
Formaldehyde[1,2]
```

```
0.086
```

```
Formaldehyde$optden[1]
```

```
0.086
```

More Data Structures: Data Frames

There can be multiple columns:

```
Indometh
```

```
Subject time conc
1 1 0.25 1.50
2 1 0.50 0.94
3 1 0.75 0.78
4 1 1.00 0.48
5 1 1.25 0.37
...
66 6 8.00 0.09
```

More Data Structures: Data Frames

We can select subsets of the data:

```
Indometh[Indometh$Subject==6,]
```

```
Subject time conc
56 6 0.25 2.31
57 6 0.50 1.44
58 6 0.75 1.03
...
66 6 8.00 0.09
```

Character string Manipulation

Length:

```
s = ’The quick brown fox’
nchar(s)
```

```
19
```

Substring:

```
substr( s, start=5, stop=9 )
```

```
quick
```

Character string Manipulation

Join:

```
a = "The"
b = "quick"
c = paste( a, b )
c
```

```
The quick
```

Or: `c = paste(a,b,sep='')`

Input and output

`message` will output to the console (terminal window). (So will `print`)

We also need to read data from files, and from the user.

Input and output

readline() reads from the keyboard:

```
name = readline()
x = as.numeric( readline( "Enter a number: " ) )
```

But usually we’ve got lots of data and don’t want to type it all in. So we read directly from a file.

Input and output

Reading from a file `data.txt`:

```
1 0.25 1.50
1 0.50 0.94
1 0.75 0.78
1 1.00 0.48
1 1.25 0.37
```

```
data = read.table( "data.txt" )
```

Input and output

```
> head(data)
V1 V2 V3
1 1 0.25 1.50
2 1 0.50 0.94
3 1 0.75 0.78
4 1 1.00 0.48
5 1 1.25 0.37
```

```
concs = data$V3
```

Input and output

```
Subject time conc
1 0.25 1.50
1 0.50 0.94
1 0.75 0.78
1 1.00 0.48
1 1.25 0.37
```

```
data = read.table( "datahead.txt", header=TRUE )
```

Input and output

```
> head(data)
Subject time conc
1 1 0.25 1.50
2 1 0.50 0.94
3 1 0.75 0.78
4 1 1.00 0.48
5 1 1.25 0.37
```

```
concs = data$conc
```

Input and output

If there are no headers, we can set them ourselves:

```
data = read.table( "data.txt" )
names(data) = c('Subject','time','conc')
```

or:

```
data = read.table( "data.txt", col.names=c('Subject','
```

Input and output

Other options:

For data separated by commas:

```
data = read.table( "data.txt", sep="," )
```

To skip 4 lines of comments at the top of a file:

```
data = read.table( "data.txt", skip=4 )
```

For more:

```
?read.table
```

Debugging

When you write code, you get bugs. Therefore you need to debug the code.

Approaches:

- Stare at it and hope. (Least effort, least results).
- Dry run. (Most effort required).
- Diagnostic code. (e.g. print).
- Use a debugger. (Most skill required).

Debugging

- Dry run:

  Run your program on paper, writing out the values of every variable as each statement is executed. This is an extremely good exercise, but less useful in real life.

- Diagnostic code:

  Put in a print statement after each line, printing anything which was changed by the previous line. Read the output and make sure everything is behaving as it should.

Debugging

- Stare at it and hope.

  If you don’t see the problem in 60 seconds, then STOP! Use one of the previous methods.

- Use a debugger.

  Beyond the scope of this course.

Algorithms

algorithm a step-by-step procedure for solving a problem or accomplishing some end especially by a computer. (Webster)

To write a program, we first devise an algorithm to describe the problem, and then implement the algorithm in a particular language.

Algorithms are sometimes described by flowcharts.

Algorithms

The individual instructions which make a computer program are very simple. To program any complex task we must break it down into these very simple steps.

e.g. Swap two numbers:

```
a = 3
b = 7
message( a, ’ ’, b )
# SWAP THE NUMBERS HERE
message( a, ’ ’, b )
```

```
3 7
7 3
```

Algorithms

If we say

```
a = b
b = a
```

then a is set to the value of b. But we've lost the old value of a, and so we can't set b to it.

We must add another step, using a temporary variable:

```
x = b
b = a
a = x
```

Algorithms

Finding the largest number in a list:

```
mylist = c( 5, 11, 64, 27, 31, 1 )
```

We want to find the largest value, in this case 64.

How do we do this? There are a number of ways.

Algorithms

A first attempt: The largest number in the list can be detected by testing that none of the other numbers are bigger than it. Therefore we can try each number in turn, and see if there is any other bumber bigger.

We need two nested loops, the outer to try each number in turn, and the inner to test it against every other number.

```
list = c( 5, 11, 64, 27, 31, 1 )
for (i in 1:length(list)) {
  islargest = "yes"
  for (j in 1:length(list))
    if ( list[j] > list[i] )
      islargest = "no"
  message( list[i], ’ ’, islargest )
}
```

Algorithms

This method contains two nested loops over all the data. Therefore the number of comparisons, and the time taken to run is proportional to the square of the number of elements in the list. For a long list, this can be very slow.

We say the efficiency of the method is order N squared ($O(n^2)$).

A much more efficient method is possible, with an efficiency $O(n)$.

To do this we will store the largest number found so far, and search through the list updating this value whenever we find a larger one.

Algorithms

```
list = c( 5, 11, 64, 27, 31, 1 )
largestSoFar = 0
for (i in 1:length(list)) {
  if ( list[i] > largestSoFar )
    largestSoFar = list[i]
}
largestSoFar
```

```
64
```

What happens if all the numbers in the list are negative?

Algorithms

Sorting: Sorting lists is one of the most common algorithms.

A simple way to sort a list would be to find the smallest element in the list, and swap it to the front. Then find the smallest in the rest of the list, and swap it to the place after the first one. Carry on until all the numbers have been sorted.

This approach will work, but is messy to code. A simpler approach is built out of the simple swap operations we used before:

Loop over all the elements of the list, bar the last one. If this element is bigger than the next one, then swap them. This means that the list will be slightly more sorted than it was before.

Repeat this process over and over again until all the elements are sorted. We must count the swaps made each time, to see if the sort is finished.