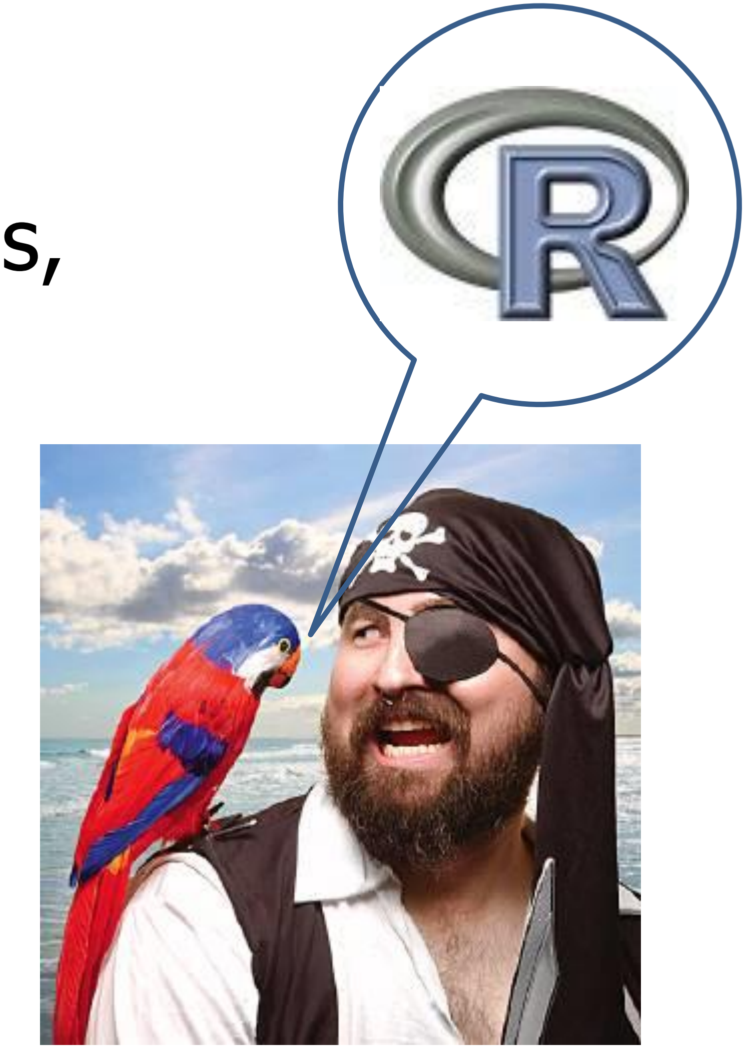
R basics

K. Limburg lecture notes,
3-28-2016

Updated slightly,
1-23-2023



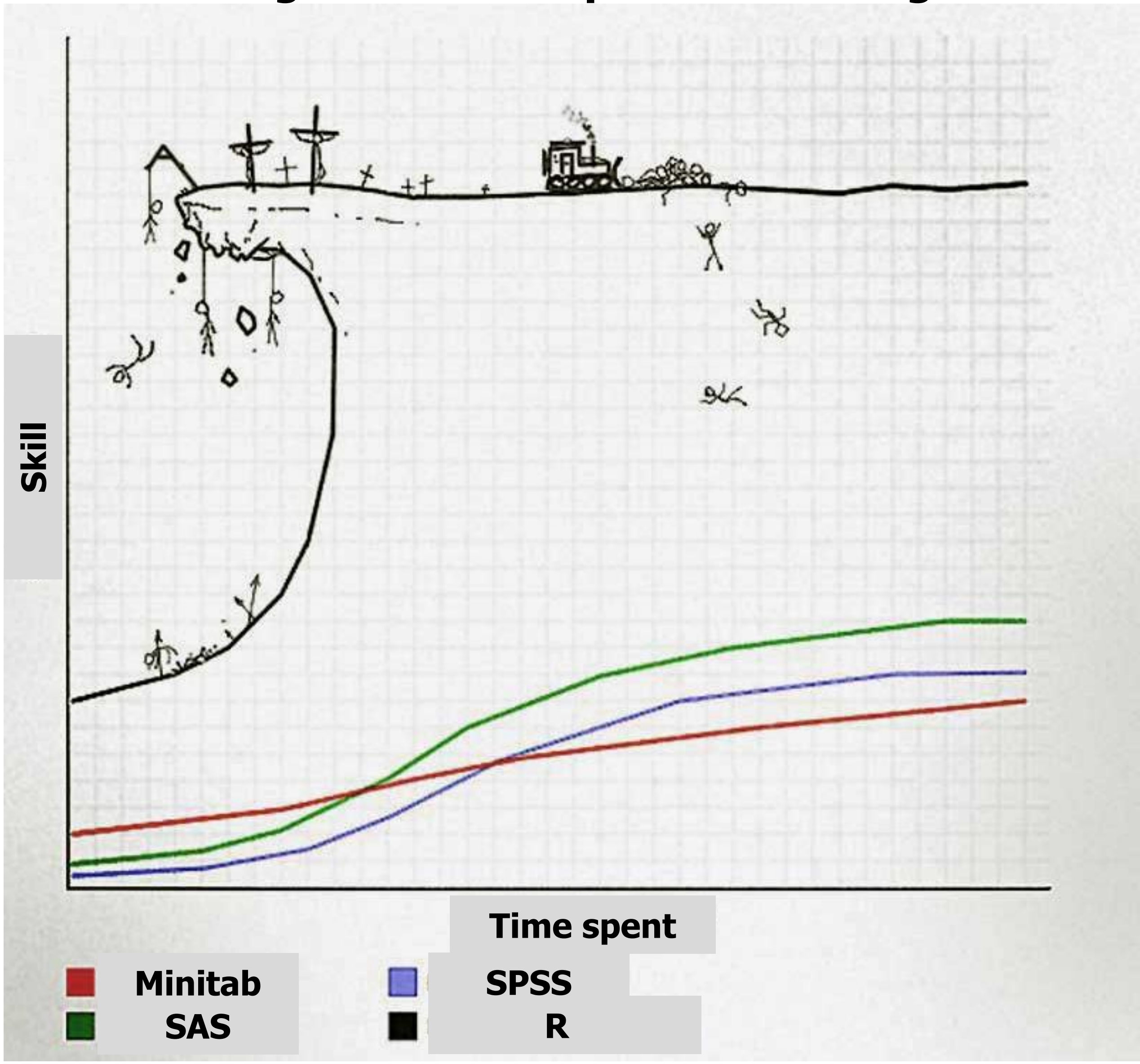
R is a programming language originally designed for doing statistical analysis

(R is the freeware version of S)

R has a huge user community

- Has become the de-facto programming language of choice amongst ecologists (and also many others)
- Tons of routines and packages written for R, and peer reviewed (doesn’t mean they’ll all work, but a lot do)
- Therefore, once you get familiar enough with R to grope around, you can do what everyone else does: *borrow code and cobble it to your own needs*

Learning Curves of Popular Stats Programs



(Homage to Tom Evans)

Why is there a steep learning curve to R?

- The commands are somewhat cryptic
  - (example: t(A) means “transpose the matrix called A”)
- The commands can have lots of arguments (i.e., parameters that you can tailor to your needs) – can be bewildering at first
- There are *sooo* many ways do to things!
- The help files are cryptic and clearly are developed by computer geeks

Nevertheless, you can get the hang of it after a while…

Here’s the thing though…

- “*Using R is a bit akin to smoking. The beginning is difficult, one may get headaches and even gag the first few times. But in the long run, it becomes pleasurable and even addictive. Yet, deep down, for those willing to be honest, there is something not fully healthy in it*.” --Francois Pinard
- Using R is hard at first, but once you start to get it down it can be incredibly rewarding

(more homage to Tom Evans)

Here's what I'll go over today:

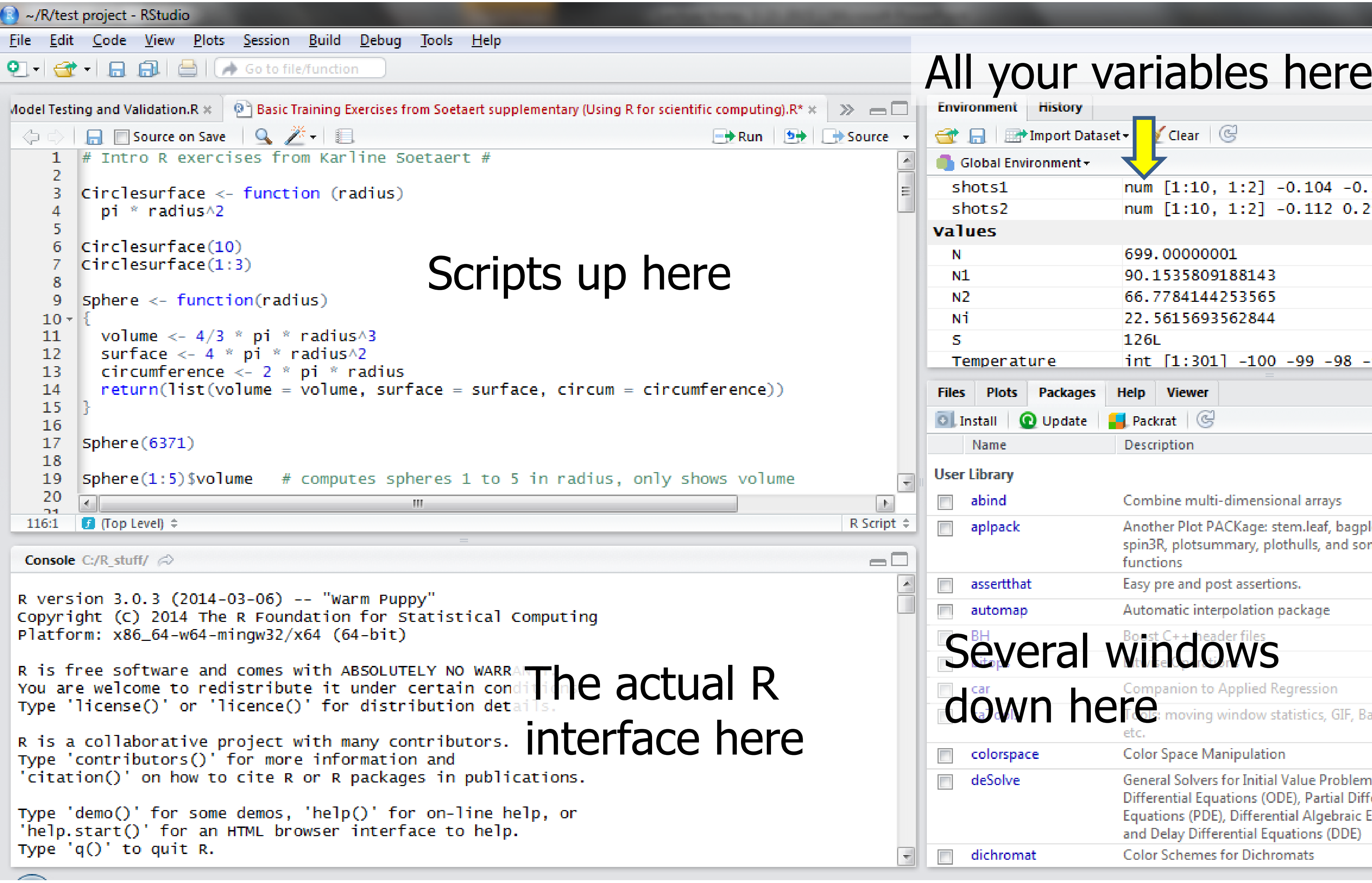
A bit about RStudio, the GUI editorial interface

Basics about

- setting up directories
- getting data files into and out of R
- data types
- hierarchical rules of computation
- data frames (one of the most common data structures used in R)
- functions
- "programming structures" and practices
- plotting

Homework: Work through "Using R for scientific computing" by Karline Soetaert (sent to you on 3/27)

The RStudio environment


~/R/test project - RStudio
File Edit Code View Plots Session Build Debug Tools Help
Go to file/function
Model Testing and Validation.R
Basic Training Exercises from Soetaert supplementary (Using R for scientific computing).R*
Source on Save
Run
Source
116:1
(Top Level)
R Script
Console C:/R_stuff/
Environment History
Import Dataset
Clear
Global Environment
Files Plots Packages Help Viewer
Install Update Packrat

All your variables here

Scripts up here

The actual R interface here

Several windows down here

```
# Intro R exercises from Karline Soetaert #

Circlesurface <- function (radius)
  pi * radius^2

Circlesurface(10)
Circlesurface(1:3)

Sphere <- function(radius)
{
  volume <- 4/3 * pi * radius^3
  surface <- 4 * pi * radius^2
  circumference <- 2 * pi * radius
  return(list(volume = volume, surface = surface, circum = circumference))
}

Sphere(6371)

Sphere(1:5)$volume   # computes spheres 1 to 5 in radius, only shows volume
```

```
R version 3.0.3 (2014-03-06) -- "Warm Puppy"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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.

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.
```

```
  shots1        num [1:10, 1:2] -0.104 -0.
  shots2        num [1:10, 1:2] -0.112 0.2
Values
  N             699.00000001
  N1            90.1535809188143
  N2            66.7784144253565
  Ni            22.5615693562844
  S             126L
  Temperature   int [1:301] -100 -99 -98 -
```

	Name	Description
User Library		
□	abind	Combine multi-dimensional arrays
□	aplpack	Another Plot PACKage: stem.leaf, bagpl spin3R, plotsummary, plothulls, and sor functions
□	assertthat	Easy pre and post assertions.
□	automap	Automatic interpolation package
□	BH	Boost C++ header files
□	bitops	Bitwise Operations
□	car	Companion to Applied Regression
□	caTools	Tools: moving window statistics, GIF, Ba etc.
□	colorspace	Color Space Manipulation
□	deSolve	General Solvers for Initial Value Problem Differential Equations (ODE), Partial Diffe Equations (PDE), Differential Algebraic E and Delay Differential Equations (DDE)
□	dichromat	Color Schemes for Dichromats

Navigating in your directories

R will launch in a particular directory; but it may not be where you want to work.

(I like to keep my R stuff in a directory located in the root directory.)

So, one thing that might be handy is a list of commands for how to move around.

Moving around in your directories:

getwd() … returns the current directory path (empty parens means current directory)

list.files() … lists out what’s in the current directory

setwd(“C:/dir/nextdir”) … sets the directory to where you want. Beware – you need quotation marks, the slash has to be forward (/), and R is case sensitive!

dir.create(“new_dir”) … creates a new subdirectory within the current directory

dir.create(“../new_dir”) … creates a new subdirectory one level up

You may want to create data folders, script folders, etc.

Getting data files into and out of R (the basics)

Input

read.txt(“filedirectory/filename.txt”, header = TRUE) ← reads in a text file with headers

read.csv(etc) ← reads in a .csv file *make sure you have the .csv extension!*

read.table(file = “filedirectory/filename.txt”, header = TRUE) ← also commonly used

For Excel, there’s a package called **readxl** that you load, and then call like this: MyData <- read_excel(“filedirectory/filename.xlsx”)

Output

write.table(VariableNames *or* DataFrameName, file = “NameOfFile.txt” *or .csv*, sep = “ ”, append = FALSE *to open a new file*, na = “NA” *for missing values*)

write.csv(…) or write.txt(…) will also work

Data types

Numbers – integer, real, imaginary (and complex)

String – alphanumerics – can’t calculate with these

Vectors – “ordered collection” of numbers

Matrix – 2-dimensional collection (array) of numbers. Keep track of each value by its [row,column] “address” (elements)

Array – an n-dimensional collection of numbers. Keep track of them like a matrix. (Ex.: a 3-D array would have [x,y,z] elements)

Data frame – a collection of data that can consist of numbers and strings

List – multi-functional collections of data – can combine all of the above.

How do you get data into one of these data types?

- Interactively: use R as a calculator

```
> 2+2      > 8.5 * sin(2)/pi   > 2^12
[1] 4     [1] 2.4602           [1] 4096
```

- Create variables

```
A <- 2+2; A
(the "<-" means "is assigned to", used a LOT)
(use semicolons to separate statements on the same
line (better yet, don't put them on the same
line))

Variable names are case-sensitive!

A ≠ a     Dog ≠ dog

a <- 2^4; b <- exp(a); c <- log(b * a)

(in R, natural log is "log" and any other base is
added to the term, e.g., log base 2 is log2(blah)
```

A digression on the hierarchy of computation…

Hierarchical order of calculations (what gets done first):

1. Trig functions (sin(), cos(), etc.), other transcendental functions (log(), exp(), sqrt(), etc.)
2. Multiplication, division
3. Addition, subtraction

When in doubt, enclose the bits you want to calculate together in parentheses!

```
> X=3; Z=(X+2*sqrt(X))/(X-5*sqrt(X)); Z
[1] -1.142
```

Try: `Z = X+2*sqrt(X) / X-5*sqrt(X)`

Ways to populate variables with more than one number in them.

1. The “concatenate” or “combine” function

   c(blah, blah, blah)

   This allows one to input numbers or strings, or even combinations, or other variables, into a variable

```
>  A <- c(1,2,3); A
[1] 1 2 3
> B <- 2; A <- c(1,B,3); A
[1] 1 2 3
> A <- c("cat","dog", "mouse"); A
[1] "cat" "dog" "mouse"
```

What happens with this?

```
> A <- c("cat", 10, "mouse"); A
```

R recognizes the data type and assigns it thusly.

Populating variables, cont'd.

2. The ":" lets you create a sequence of values by increments of 1

```
A <- c(1:10); A
[1] 1 2 3 4 5 6 7 8 9 10
```

3. More generally, use the "seq" to generate a sequence of values by any set increment:

```
A <- seq(from=0, to=15, by=3)
[1] 0 3 6 9 12 15

A <- seq(1,15,3)
```

(← SAME THING)

```
A <- seq(from=0, to=2*pi, by=pi/2)
            ** BUT NOT **
A <- seq(1:100, 10)
```

Will get an error code

Populating variables, cont'd.

4. The "rep" command repeats numbers

```
A <- rep(1, times = 5)
[1] 1 1 1 1 1
A <- rep(c(1,3), times=5)
[1] 1 3 1 3 1 3 1 3 1 3
A <- rep(c(1:3), times=5)
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

See p. 9 of Soetaert's tutorial for more examples

Populating variables, cont'd.

5. You can also combine vectors together to form matrices.

The function **cbind()** will join them as columns
 ----- " ----- **rbind()** -------- " -------- rows

```
> A <- c(1:5)
> B <- c(6:10)
> C <- cbind(A,B)
> C
     A  B
[1,] 1  6
[2,] 2  7
[3,] 3  8
[4,] 4  9
[5,] 5 10
```

```
> A <- c(1:5)
> B <- c(6:10)
> C <- rbind(A,B)
> C
  [,1] [,2] [,3] [,4] [,5]
A    1    2    3    4    5
B    6    7    8    9   10
```

Note the convention here of referring to elements in the matrix!

When to use ( ) and when to use [ ]:

( ) ← use for calling functions of any sort
(most commands are functions…)

[ ] ← use for indexing a data storing structure
(e.g., an array, matrix, vector, data frame…)

Some other miscellany about vectors, matrices, and arrays:

- length(A) ← returns the # of elements in the vector
- dim(A) ← returns the dimension of matrix or array A
- ncol(A) ← returns the # of columns
- nrow(A) ← returns the # of rows

The data frame

This is a data structure that combines different data types. Here is an example:

```
> pet <- c("cat", "dog", "dog", "hamster")

> petName <- c("Felix", "Fido", "Missy", "Spotty")

> petWeight <- c(5, 45, 30, 0.3)

> MyPets <- data.frame(PetType = pet, Name = petName,
>                      Weight = petWeight)
> MyPets
```

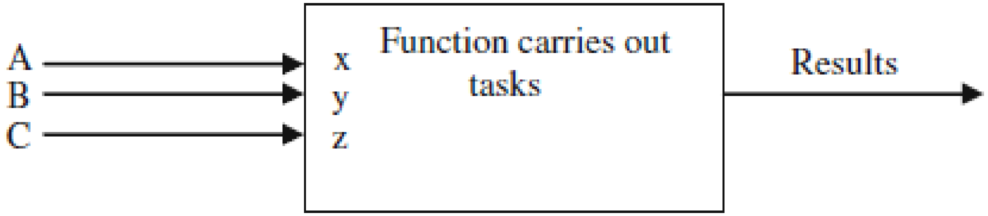
```
 PetType   Name   Weight
1     cat  Felix    5.0
2     dog   Fido   45.0
3     dog  Missy   30.0
4 hamster Spotty    0.3
```

Note the default format (vectors are glued together as columns)

You can set up data frames in Excel, then import as .txt or .csv files

Functions, especially user-defined ones

Virtually all the commands in R are a class of programming called a ***function***.



A

B

C

x

y

z

Function carries out tasks

Results

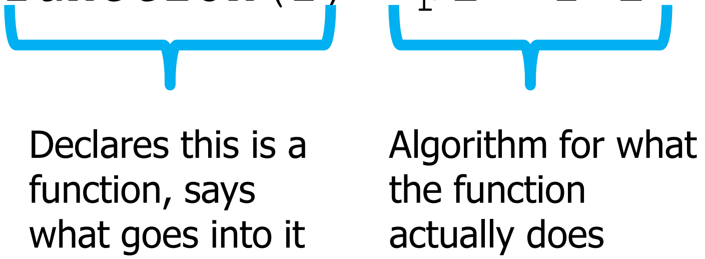
The basic structure of a function is as follows:

```
myFunctionName <- function(variable(s) that go into it)
        {
          statements that do the function
          return(what comes out of the function)
        }
```

Some functions are just one-liners:

```
> AreaOfCircle <- function(r)  pi * r^2

> AreaOfCircle(10)
[1] 314.1593
```



Declares this is a function, says what goes into it

Algorithm for what the function actually does

A more complex function will have several lines of code. The “guts” of the function needs to be inside of curly braces { }

From Soetaert’s guide:

```
Sphere <- function(radius)
{
  volume <- 4/3 * pi * radius^3
  surface <- 4 * pi * radius^2
  return(list(volume=volume, surface=surface))
}
```

☞ In Soetaert et al.’s package **deSolve** (simulation package), your model itself is written as a function

A few more bits:

1. For programming, we often have test statements

That is, **if** (such-&-such condition exists) then (do something) *and oftentimes, there's an* **else** *too*

Another example from Soetaert:

```
Dummy <- function (x)
{
 if ( x<0 ) string <- “x < 0” else
 if ( x<2 ) string <- “0 >= x < 2” else
            string <- “x >= 2”
 print(string)
}
```

Try it, then test it, then modify it…

A few more bits, cont’d.:

2. Loops – one of the things computers do well, though it’s not so efficient in R (so they say)

Loops do things repeatedly until they are done, or until you break out of the loop

In R, the construct is as follows:

```
for (counter in start:end)
  {
    do stuff
  }
```

```
Ex: for (i in 1:10) print(i)
```

A few more bits, cont'd.:

3. R packages: this is the golden treasure-trove!

Have to install them from the WWW, then each time you want to use them in a new session, you have to call them up with the command **library(*packageName*)**

```
library()  ← shows all currently available libraries
library(deSolve)  ← loads up the deSolve package
library(help = deSolve) ← loads up the help file
help(package = deSolve) ← displays the help file
```

A few more bits, cont’d.:

5. Scripts and comments

You can use your editor to write whole sets of commands (a “script,” known in the good old days as a “program”)

When you do this, it’s **essential** that you add comments to annotate the code.

A comment is preceded by a hash-mark (#)

Plotting: see Soetaert manual, pp 26-33

Plotting is pretty complex, and there are several packages that go beyond R.

Nevertheless, there's some built-in plotting functions that'll get you pretty far.

Here is a simple start, using a built-in data set called **Orange**.

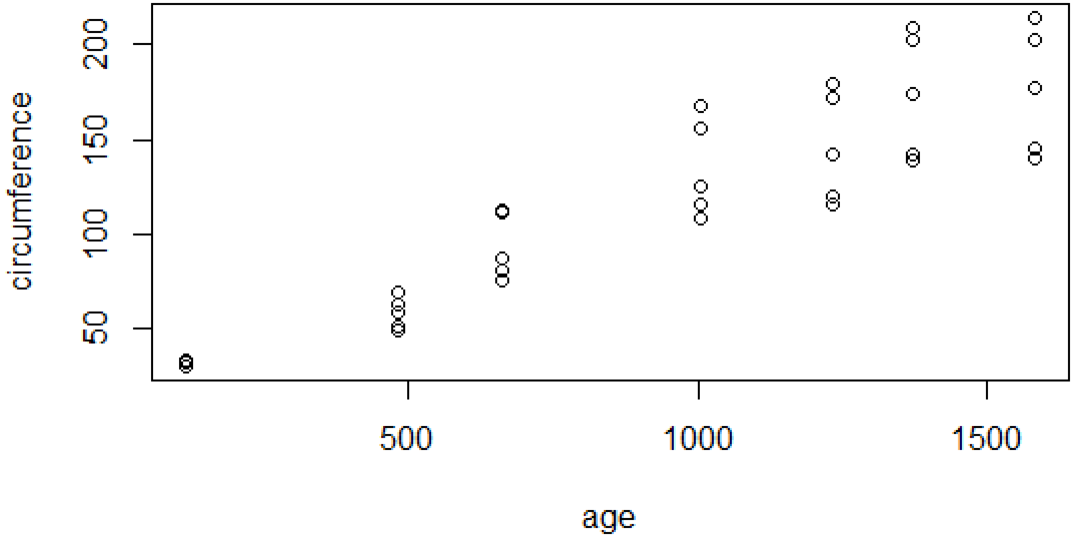
```
> head(Orange)
  Tree  age  circumference
1    1  118             30
2    1  484             58
3    1  664             87
4    1 1004            115
5    1 1231            120
6    1 1372            142
```

← shows first 6 rows of the data frame

Let's plot **circumference** as a function of **age** (a simple X-Y plot)

```
plot(circumference ~ age, data = Orange)
```

The “~” means that **circumference** is the dependent var and is plotted against **age**, the independent var. We also tell R which data set to use.



We can add a few more fancy things, like better X and Y labels, a different symbol, and color the symbol
(note – this is one place where you can go crazy…)

```
plot(x = Orange$age, y = Orange$circumference,
    xlab= "Age in days", ylab= "Trunk circumference, mm",
    pch = 17, col = "lightblue")
```

Orange$age ← how we address the age variable in the Orange data frame

xlab, ylab ← x and y labels

"pch = *<number>*" ← tells R to use a certain symbol (type ?pch to see what’s available)

col = color (type ?col)

Remember, any time you want help*,

type ?*<command>*

Best thing to do is “a,b,c”:

a) Find the example in the help file
b) Work the example
c) Try to modify the example

* such as it is…