2 DATA AND PLOTS

2.1 INTRODUCTION

This chapter introduces the different data types and data structures that are commonly used in R and how to visualise or 'plot' them. As you work through this book, you will gain experience of using all of these different data structures, sequentially building on ideas you have encountered previously (for example, developing your own functions). As you progress, the exercises will place more emphasis on solving problems, using and manipulating different data structures as you need them, rather than simply working through the example code. You should note the different functions called in the example code snippets that are used, such as `max`, `sqrt` and `length`. This chapter covers a lot of ground – it will:

- Review basic commands in R
- Introduce variables and assignment
- Introduce data types and classes (vectors, lists, matrices, S4, data frames)
- Describe how to test for and manipulate data types
- Introduce basic plot commands
- Describe how to read, write, load and save different data types

Chapter 1 introduced R, the reasons for using it in spatial analysis and mapping, and described how to install it. It also directed you to some of the many resources and introductory exercises describing basic operations in R. Specifically, it advised that you should work through the 'Owen Guide' (entitled *The R Guide*) up to the end of Section 5. This can be accessed via `http://cran.r-project.org/doc/contrib/Owen-TheRGuide.pdf`. This chapter assumes that you have worked your way through this introduction, which does not take long and is critical for the more specialised materials that will be introduced in the rest of this book.

2.2 THE BASIC INGREDIENTS OF R: VARIABLES AND ASSIGNMENT

The R interface can be used as a sort of calculator, returning the results of simple mathematical operations such as `(-5 + -4)`. However, it is normally convenient to *assign* values to *variables*. The variables that are created can then be manipulated or subject to further operations.

```
# examples of simple assignment
x <- 5
y <- 4
# the variables can be used in other operations
x+y

## [1] 9

# including defining new variables
z <- x + y
z

## [1] 9

# which can then be passed to other functions
sqrt(z)

## [1] 3
```

Note that in this text, R output is preceded by a double hash (##) so that it is clear that this is the output resulting from entering the R command, rather than something that should be typed in.

The snippet of code above is the first that you have come across in this book. There will be further snippets throughout each chapter. Two key points. First you are strongly advised to enter and run the code at the R prompt yourself. You may wish to write the code into a script or document as described in Chapter 1. The reasons for this are so that you get used to using the R console and to help your understanding of the code's functionality. In order to run the code in the R console, a quick way to enter it is to highlight the code (with the mouse or using the keyboard controls) and the press Ctrl-R, or Cmd-Enter on a Mac. Second, we would like to emphasise the importance of learning by doing

(Continued)

(Continued)

and getting your hands dirty. Some of the code might look a bit fearsome when first viewed, especially in later chapters, but the only really effective way to understand it is to give it a try. Remember that the code snippets are available on the book website `https://study.sagepub.com/brunsdoncomber` as scripts so that you can copy and paste these into the R console or your own script. A minor further point is that in the code comments are prefixed by # and are ignored by R when entered into the console.

The basic assignment type in R is to a *vector* of values. Vectors can have single values as in `x`, `y` and `z` above, or multiple values. Note the use of `c(4,5,...)` in the following to combine or *concatenate* multiple values:

```
# example of vector assignment
tree.heights <- c(4.3,7.1,6.3,5.2,3.2,2.1)
tree.heights

## [1] 4.3 7.1 6.3 5.2 3.2 2.1
```

Remember that UPPER and lower case matters to R. So `tree.heights`, `Tree.Heights` and `TREE.HEIGHTS` all refer to different variables. Make sure you type in upper and lower case exactly as it is written, otherwise you are likely to get an error.

In the example above, a vector of values have been assigned to the variable `tree.heights`. It is possible for a single assignment to refer to the entire vector, as in the code below that returns `tree.heights` squared. Note how the operation returns the square of each element in the vector.

```
tree.heights**2

## [1] 18.49 50.41 39.69 27.04 10.24 4.41
```

Other operations or functions can then be applied to these vectors variables:

```
sum(tree.heights)
## [1] 28.2
```



```
mean(tree.heights)
## [1] 4.7
```

And, if needed, the results can be assigned to yet further variables.

```
max.height <- max(tree.heights)
max.height

## [1] 7.1
```

One of the advantages of vectors and other structures with multiple data elements is that they can be subsetted. Individual elements or subsets of elements can be extracted and manipulated:

```
tree.heights

## [1] 4.3 7.1 6.3 5.2 3.2 2.1

tree.heights[1]    # first element

## [1] 4.3

tree.heights[1:3]  # a subset of elements 1 to 3

## [1] 4.3 7.1 6.3

sqrt(tree.heights[1:3]) # square roots of the subset

## [1] 2.074 2.665 2.510

tree.heights[c(5,3,2)]  # a subset of elements 5,3,2: note the ordering

## [1] 3.2 6.3 7.1
```

As well as numeric values as in the above examples, vectors can be assigned `character` or `logical` values as below. Different variables classes and types are described in more detail in the next section.

```
# examples of character variable assignment
name <- "Lex Comber"
name
```

```
## [1] "Lex Comber"

# these can be assigned to a vector of character variables

cities <- c("Leicester","Newcastle","London","Durham","Exeter")

cities

## [1] "Leicester" "Newcastle" "London" "Durham" "Exeter"

length(cities)

## [1] 5


# an example of a logical variable
northern <- c(FALSE, TRUE, FALSE, TRUE, FALSE)
northern

## [1] FALSE  TRUE FALSE  TRUE FALSE

# this can be used to subset other variables
cities[northern]

## [1] "Newcastle" "Durham"
```

As you explore the code in the text, the very strong advice of this book is that you write and develop the code using the in-built text editor in R. The scripts can be saved as .R files and code snippets can be run directly by highlighting them and then using Ctrl-R (Windows) or Cmd-Enter (Mac). Keeping your copies of your code in this way will help you keep a record of it, and will allow you to go back to earlier declarations of variables easily. A hypothetical example is shown below.

```
##### Example Script #####
## Load libraries
library(GISTools)
## load functions
source("My.functions.R")
## load some data
my.data <- read.csv(file = "my.data.csv")
## apply a function written in My.functions.R
cube.root.func(my.data)
## apply another R function
row.tot <- rowSums(my.data)
```


2.3 DATA TYPES AND DATA CLASSES

This section introduces data classes and data types to a sufficient depth for readers of this book. However, more formal descriptions of R data objects and class can be found in the R Manual on the CRAN website under the descriptions of:

- Basic classes: `http://stat.ethz.ch/R-manual/R-devel/library/methods/html/BasicClasses.html`
- Classes: `http://stat.ethz.ch/R-manual/R-devel/library/methods/html/Classes.html`

2.3.1 Data Types in R

Data in R can be considered as being organised into a hierarchy of data types which can then be used to hold data values in different structures. Each of the types is associated with a test and a conversion function. The basic or core data types and associated tests and conversions are shown in the table below.

type	test	conversion
character	is.character	as.character
complex	is.complex	as.complex
double	is.double	as.double
expression	is.expression	as.expression
integer	is.integer	as.integer
list	is.list	as.list
logical	is.logical	as.logical
numeric	is.numeric	as.numeric
single	is.single	as.single
raw	is.raw	as.raw

You should note from the table that each type has associated with it a test is.xyz, which will return `TRUE` or `FALSE` and a conversion as xyz. Most of the exercises,

methods, tools, functions and analyses in this book work with only a small subset of these data types:

- character
- numeric
- logical

These data types can be used to populate different data structures or classes, including vectors, matrices, data frames, lists and factors. The data types are described in more detail below. In each case the objects created by the different classes, conversion functions or tests are illustrated.

Characters

Character variables contain text. By default the function `character` creates a vector of whatever length is specified. Each element in the vector is equal to `""` – an empty character element in the variable. The function `as.character` tries to convert its argument to character type, removing any attributes including, for example vector element names. The function `is.character` tests whether the arguments passed to it are of character type and returns `TRUE` or `FALSE` depending on whether its argument is of character type or not.

Consider the following examples of these functions and the results when they are applied to different inputs:

```
character(8)

## [1] "" "" "" "" "" "" "" ""

# conversion
as.character("8")

## [1] "8"

# tests
is.character(8)

## [1] FALSE

is.character("8")

## [1] TRUE
```


Numeric

Numeric data variables are used to hold numbers. The function `numeric` is used to create a vector of the specified length with each element equal to `0`. The function `as.numeric` tries to convert (coerce) its argument to numeric type. It is identical to `as.double` and to `as.real`. The function `is.numeric` tests whether the arguments passed to it are of numeric type and returns `TRUE` or `FALSE` depending on whether its argument is of numeric type or not. Notice how the last test returns `FALSE` because not all the elements are numeric.

```
numeric(8)

## [1] 0 0 0 0 0 0 0 0

# conversions
as.numeric(c("1980","-8","Geography"))

## [1] 1980 -8 NA

as.numeric(c(FALSE,TRUE))

## [1] 0 1

# tests
is.numeric(c(8, 8))

## [1] TRUE

is.numeric(c(8, 8, 8, "8"))

## [1] FALSE
```

Logical

The function `logical` creates a logical vector of the specified length and by default each element of the vector is set to equal `FALSE`. The function `as.logical` attempts to convert its argument to be of logical type. It removes any attributes including, for example, vector element names. A range of character strings `c("T", "TRUE", "True", "true")`, as well any number not equal to zero, are regarded as TRUE. Similarly, `c("F", "FALSE", "False", "false")` and zero are regarded as `FALSE`. All others as are regarded as `NA`. The function `is.logical` returns `TRUE` or `FALSE` depending on whether the argument passed to it is of logical type or not.

```
logical(7)

## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE

# conversion
as.logical(c(7, 5, 0, -4,5))

## [1]  TRUE TRUE FALSE  TRUE TRUE

# TRUE and FALSE can be converted to 1 and 0
as.logical(c(7,5,0,-4,5)) * 1

## [1] 1 1 0 1 1

as.logical(c(7,5,0,-4,5)) + 0

## [1] 1 1 0 1 1

# different ways to declare TRUE and FALSE
as.logical(c("True","T","FALSE","Raspberry","9","0", 0))

## [1]  TRUE TRUE FALSE    NA    NA    NA    NA
```

Logical vectors are very useful for indexing data, to select the data that satisfy some criteria. In spatial analysis this could be used to select database records that match some criteria. For example, consider the following:

```
data <- c(3, 6, 9, 99, 54, 32, -102)
# a logical test
index <- (data > 10)
index

## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE

# used to subset data
data[index]

## [1] 99 54 32

sum(data)

## [1] 101

sum(data[index])

## [1] 185
```


2.3.2 Data Classes in R

The different data types can be used to populate different data structures or classes. This section will describe and illustrate vectors, matrices, data frames, lists and factors – data classes that are commonly used in spatial data analysis.

Vectors

All of the commands in R in the previous subsection on data types produced vectors. Vectors are the most commonly used data structure and the standard one-dimensional R variable.You will have noticed that when you specified `character` or `logical`, etc., a vector of a given length was produced. An alternative approach is to use the function `vector`, which produces a vector of the length and type or mode specified. The function `as.vector` seeks to convert its argument into a vector of whatever mode is specified. The default is `logical`, and when you assign values to vectors R will seek to convert them to whichever vector mode is most convenient. The test `is.vector` returns `TRUE` if its argument is a vector of the specified class or mode with no attributes other than names, returning `FALSE` otherwise.

```
# defining vectors
vector(mode = "numeric", length = 8)

## [1] 0 0 0 0 0 0 0 0

vector(length = 8)

## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

# testing and conversion
tmp <- data.frame(a=10:15, b=15:20)
is.vector(tmp)

## [1] FALSE

as.vector(tmp)

##    a  b
## 1 10 15
## 2 11 16
## 3 12 17
## 4 13 18
## 5 14 19
## 6 15 20
```

Matrices

The function `matrix` creates a matrix from the data and parameters that are passed to it. This should normally include parameters for the number of columns and rows in the matrix. The function `as.matrix` attempts to turn its argument into a matrix, and the test `is.matrix` tests to see whether its argument is a matrix.

```
# defining matrices
matrix(ncol = 2, nrow = 0)

## [,1] [,2]

matrix(1:6)

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

matrix(1:6, ncol = 2)

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

# conversion and test
as.matrix(6:3)

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

is.matrix(as.matrix(6:3))

## [1] TRUE
```


Notice how it is possible to specify empty data structures - the first line of code above produced an empty matrix with two columns and no rows. This can be very useful in some circumstances.

Matrix rows and columns can be named.

```
flow <- matrix(c(2000, 1243, 543, 1243, 212, 545,
     654, 168, 109), c(3,3), byrow=TRUE)
# Rows and columns can have names, not just 1,2,3,...
colnames(flow) <- c("Leicester", "Liverpool","Elsewhere")
rownames(flow) <- c("Leicester", "Liverpool", "Elsewhere")
# examine the matrix
flow

##                Leicester      Liverpool      Elsewhere
## Leicester           2000           1243            543
## Liverpool           1243            212            545
## Elsewhere            654            168            109

# and functions exist to summarise
outflows <- rowSums(flow)
outflows

## Leicester  Liverpool  Elsewhere
##      3786       2000        931
```

However, if the data class is not a matrix then just use `names`, rather than `rownames` or `colnames`.

```
z <- c(6,7,8)
names(z) <- c("Newcastle","London","Manchester")
z

## Newcastle    London Manchester
##         6         7         8
```

R has many additional tools for manipulating matrices and performing matrix algebra functions that are not described here. However, as spatial scientists we are often interested in analysing data that has a matrix-like form, as in a data table. For example, in an analysis of spatial data in vector format, the rows in the attribute

table represent specific features (such as polygons) and the columns hold information about the attributes of those features. Alternatively, in a raster analysis environment, the rows and columns may represent specific latitudes and longitudes or northings and eastings or raster cells. Methods for analysing matrices in this way will be covered in more detail in later chapters as spatial data objects (Chapter 3) and spatial analyses (Chapter 5) are introduced.

You will have noticed in the code snippets that a number of new functions are introduced. For example, early in this chapter, the function `sum` was used. R includes a number of functions that can be used to generate descriptive statistics such as `sum`, and `max`. You should explore these as they occur in the text to develop your knowledge of and familiarity with R. Further useful examples are in the code below and throughout this book. You could even store them in your own R script. R includes extensive help files which can be used to explore how different functions can be used, frequently with example snippets of code. An illustration of how to find out more about the `sum` function and some further summary functions is given below:

```
?sum
help(sum)
# Create a variable to pass to other summary functions
x <- matrix(c(3,6,8,8,6,1,-1,6,7),c(3,3),byrow=TRUE)
# Sum over rows
rowSums(x)
# Sum over columns
colSums(x)
# Calculate column means
colMeans(x)
# Apply function over rows (1) or columns (2) of x
apply(x,1,max)
# Logical operations to select matrix elements
x[,c(TRUE,FALSE,TRUE)]
# Add up all of the elements in x
sum(x)
# Pick out the leading diagonal
diag(x)
# Matrix inverse
solve(x)
# Tool to handle rounding
zapsmall(x %*% solve(x))
```


Factors

The function `factor` creates a vector with specific categories, defined in the `levels` parameter. The ordering of factor variables can be specified and an `ordered` function also exists. The functions `as.factor` and `as.ordered` are the coercion functions. The test `is.factor` returns `TRUE` or `FALSE` depending on whether their arguments is of factor type or not and `is.ordered` returns `TRUE` when its argument is an ordered factor and `FALSE` otherwise.

First, let us examine factors:

```
# a vector assignment
house.type <- c("Bungalow", "Flat", "Flat",
    "Detached", "Flat", "Terrace", "Terrace")
# a factor assignment
house.type <- factor(c("Bungalow", "Flat",
    "Flat", "Detached", "Flat", "Terrace", "Terrace"),
    levels=c("Bungalow","Flat","Detached","Semi","Terrace"))
house.type

## [1] Bungalow Flat  Flat  Detached Flat  Terrace  Terrace
## Levels: Bungalow  Flat  Detached  Semi  Terrace

# table can be used to summarise
table(house.type)

## house.type
## Bungalow Flat Detached Semi Terrace
##        1    3        1    0       2

# 'levels' control what can be assigned
house.type <- factor(c("People Carrier", "Flat",
    "Flat", "Hatchback", "Flat", "Terrace", "Terrace"),
    levels=c("Bungalow","Flat","Detached","Semi","Terrace"))
house.type

## [1] <NA>    Flat    Flat    <NA>    Flat    Terrace    Terrace
## Levels: Bungalow Flat Detached Semi Terrace
```

Factors are useful for categorical or classified data – that is, data values that must fall into one of a number of predefined classes. It is obvious how this might be relevant to geographical analysis, where many features represented in spatial data are labelled using one of a set of discrete classes. Ordering allows inferences about preference or hierarchy to be made (lower–higher, better–worse, etc.) and

this can be used in data selection or indexing (as above) or in the interpretation of derived analyses.

Ordering

There is no concept of ordering in factors. However, this can be imposed by using the `ordered` function.

```
income <-factor(c("High", "High", "Low", "Low",
    "Low", "Medium", "Low", "Medium"),
    levels=c("Low", "Medium", "High"))
income > "Low"

## [1] NA NA NA NA NA NA NA NA

# 'levels' in 'ordered' defines a relative order
income <-ordered (c("High", "High", "Low", "Low",
    "Low", "Medium", "Low", "Medium"),
    levels=c("Low", "Medium", "High"))
income > "Low"

## [1] TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
```

Thus we can see that ordering is implicit in the way that the levels are specified and allows other, ordering related functions to be applied to the data.

The functions `sort` and `table` are new functions. In the above code relating to factors, the function `table` was used to generate a tabulation of the data in `house.type`. It provides a count of the occurrence of each level in `house.type`. The command `sort` orders a vector or factor. You should use the help in R to explore how these functions work and try them with your own variables. For example:

```
sort(income)
```

Lists

The `character`, `numeric` and `logical` data types and the associated data classes described above all contain elements that must all be of the same basic type. Lists do not have this requirement. Lists have slots for different elements and can be considered as an ordered collection of elements. A list allows you to gather a variety of different data types together in a single data structure and the *n*th element of a list is denoted by double square brackets.


```
tmp.list <- list("Lex Comber",c(2005, 2009), "Lecturer",
    matrix(c(6,3,1,2), c(2,2)))
tmp.list

## [[1]]
## [1] "Lex Comber"
##
## [[2]]
## [1] 2005 2009
##
## [[3]]
## [1] "Lecturer"
##
## [[4]]

##      [,1] [,2]
## [1,]    6    1
## [2,]    3    2

# elements of the list can be selected
tmp.list[[4]]

##      [,1] [,2]
## [1,]    6    1
## [2,]    3    2
```

From the above it is evident that the function `list` returns a list structure composed of its arguments. Each value can be tagged depending on how the argument was specified. The conversion function `as.list` attempts to coerce its argument to a list. It turns a factor into a list of one-element factors and drops attributes that are not specified. The test `is.list` returns `TRUE` if and only if its argument is a list. These are best explored through some examples; note that `list` items can be given names.

```
employee <- list(name="Lex Comber", start.year = 2005,
    position="Professor")
employee

## $name
## [1] "Lex Comber"
##
## $start.year
## [1] 2005
```

```
##
## $position
## [1] "Professor"
```

Lists can be joined together with `append`, and `lapply` applies a function to each element of a list.

```
append(tmp.list, list(c(7,6,9,1)))

## [[1]]
## [1] "Lex Comber"
##
## [[2]]
## [1] 2005 2009
##

## [[3]]
## [1] "Lecturer"
##
## [[4]]
##      [,1] [,2]
## [1,]    6    1
## [2,]    3    2
##
## [[5]]
## [1] 7 6 9 1

# lapply with different functions
lapply(tmp.list[[2]], is.numeric)

## [[1]]
## [1] TRUE
##
## [[2]]
## [1] TRUE

lapply(tmp.list, length)

## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
```



```
##
## [[3]]
## [1] 1
##
## [[4]]
## [1] 4
```

Note that the `length` of a matrix, even when held in a list, is the total number of elements.

Defining your own classes

In R it is possible to define your own data type and to associate it with specific behaviours, such as its own way of printing and drawing. For example, you will notice in later chapters that the `plot` function is used to draw maps for spatial data objects as well as conventional graphs. Suppose we create a list containing some employee information.

```
employee <- list(name="Lex Comber", start.year = 2005,
    position="Professor")
```

This can be assigned to a new class, called `staff` in this case (it could be any name but meaningful ones help).

```
class(employee) <- "staff"
```

Then we can define how R treats that class in the form `<existing function>.<class>`; to change, for example, how it is printed. Note how the existing function for printing is modified by the new class definition:

```
print.staff <- function(x) {
    cat("Name: ",x$name, "\n")
    cat("Start Year: ",x$start.year, "\n")
    cat("Job Title: ",x$position, "\n")}
# an example of the print class
print(employee)

## Name:  Lex Comber
## Start Year:  2005
## Job Title:  Professor
```

You can see that R knows to use a different `print` function if the argument is a variable of class `staff`. You could modify how your R environment treats existing

classes in the same way, but do this with caution. You can also 'undo' the class assigned by using `unclass` and the `print.staff` function can be removed permanently by using `rm(print.staff)`:

```
print(unclass(employee))

## $name
## [1] "Lex Comber"
##
## $start.year
## [1] 2005
##
## $position
## [1] "Professor"
```

Classes in lists

Variables can be assigned to new or user-defined class objects. The example below defines a function to create a new `staff` object.

```
new.staff <- function(name,year,post) {
  result <- list(name=name, start.year=year, position=post)
  class(result) <- "staff"
  return(result)}
```

A list can then be defined, which is populated using that function as in the code below (note that functions will be dealt with more formally in later chapters).

```
leics.uni <- vector(mode='list',3)
# assign values to elements in the list
leics.uni[[1]] <- new.staff("Fisher, Pete", 1991,
    "Professor")
leics.uni[[2]] <- new.staff("Comber, Lex", 2005,
    "Lecturer")
leics.uni[[3]] <- new.staff("Burgess, Robert", 1998, "VC")
```

And the list can be examined:

```
leics.uni
## [[1]]
## Name:  Fisher,  Pete
## Start Year:  1991
```



```
## Job Title:  Professor
##
## [[2]]
## Name:  Comber,  Lex
## Start Year:  2005
## Job Title:  Lecturer
##
## [[3]]
## Name:  Burgess,  Robert
## Start Year:  1998
## Job Title:  VC
```

2.3.3 Self-Test Questions

In the next pages there are a number of self-test questions. In contrast to the previous sections where the code is provided in the text for you to work through (i.e. you to enter and run it yourself), the self-test questions are tasks for you to complete, mostly requiring you to write R code. Answers are provided at the end of this chapter. The self-test questions relate to the main data types that have been introduced: factors, matrices, lists (named and unnamed) and classes.

Factors

Recall from the descriptions above that factors are used to represent categorical data – where a small number of categories are used to represent some characteristic in a variable. For example, the colour of a particular model of car sold by a showroom in a week can be represented using factors:

```
colours <- factor(c("red","blue","red","white",
    "silver","red","white","silver",
    "red","red","white","silver","silver"),
    levels=c("red","blue","white","silver","black"))
```

Since the only colours this car comes in are red, blue, white, silver and black, these are the only levels in the factor.

Self-Test Question 1. Suppose you were to enter:

```
colours[4] <- "orange"
colours
```

What would you expect to happen? Why?

Next, use the `table` function to see how many of each colour were sold. First reassign the colours, as you may have altered this variable in the previous self-test question. You can find previously used commands using the up and down arrows on your keyboard.

```
colours <- factor(c("red","blue","red","white",
    "silver", "red", "white", "silver",
    "red","red","white","silver","silver"),
    levels=c("red","blue","white","silver","black"))
table(colours)

## colours
##     red blue white silver black
##       5    1     3      4     0
```

Note that the result of the `table` function is just a standard vector, but that each of its elements is named – the names in this case are the levels in the factor. Now suppose you had simply recorded the colours as a character variable, in `colours2` as below, and then computed the table:

```
colours2 <-c("red","blue","red","white",
    "silver","red","white","silver",
    "red","red","white","silver")
# Now, make the table
table(colours2)

## colours2
##    blue red  silver  white
##       1   5       3      3
```

Self-Test Question 2. What two differences do you notice between the results of the two `table` expressions?

Now suppose we also record the type of car – it comes in saloon, convertible and hatchback types. This can be specified by another factor variable called `car.type`:

```
car.type <- factor(c("saloon","saloon","hatchback",
    "saloon","convertible","hatchback","convertible",
    "saloon", "hatchback","saloon", "saloon",
    "saloon", "hatchback"),
    levels=c("saloon","hatchback","convertible"))
```

The table function can also work with two arguments:


```
table(car.type, colours)

##                colours
## car.type       red blue white silver black
##   saloon         2    1     2      2     0
##   hatchback      3    0     0      1     0
##   convertible    0    0     1      1     0
```

This gives a two-way table of counts – that is, counts of red hatchbacks, silver saloons and so on. Note that the output this time is a `matrix`. For now enter:

```
crosstab <- table(car.type,colours)
```

to save the table into a variable called `crosstab` to be used later on.

Self-Test Question 3. What is the difference between `table(car.type,colours)` and `table(colours,car.type)`?

Finally in this section, ordered factors will be considered. Suppose a third variable about the cars is the engine size, and that the three sizes are 1.1 litres, 1.3 litres and 1.6 litres. Again, this is stored in a variable, but this time the sizes are ordered. Enter:

```
engine <- ordered(c("1.1litre","1.3litre","1.1litre",
    "1.3litre","1.6litre","1.3litre","1.6litre",
    "1.1litre","1.3litre","1.1litre", "1.1litre",
    "1.3litre","1.3litre"),
    levels=c("1.1litre","1.3litre","1.6litre"))
```

Recall that with `ordered` variables, it is possible to use comparison operators >, <, >= and <=. For example:

```
engine > "1.1litre"
## [1] FALSE TRUE FALSE TRUE TRUE TRUE TRUE FALSE TRUE FALSE FALSE
## [12] TRUE TRUE
```

Self-Test Question 4. Using the `engine`, `car.type` and `colours` variables, write expressions to give the following:

- The colours of all cars with engines with capacity greater than 1.1 litres.
- The counts of types (hatchback, etc.) of all cars with capacity below 1.6 litres.
- The counts of colours of all hatchbacks with capacity greater than or equal to 1.3 litres.

Matrices

Recall that in the previous section you created a variable called `crosstab` – and that this was a matrix. In the section on matrices, a number of functions were shown that could be applied to matrices:

```
dim(crosstab) # Matrix dimensions

## [1] 3 5

rowSums(crosstab) # Row sums

##      saloon      hatchback      convertible
##          7              4                2

colnames(crosstab) # Column names

## [1] "red"    "blue"    "white"    "silver"    "black"
```

Another important tool for matrices is the `apply` function. This applies a function to either the rows or columns of a matrix giving a single-dimensional list as a result. A simple example finds the largest value in each row:

```
apply(crosstab,1,max)

##    saloon    hatchback    convertible
##        2            3              1
```

In this case, the function `max` is applied to each row of `crosstab`. The `1` as the second argument specifies that the function will be applied row by row. If it were `2` then the function would be column by column:

```
apply(crosstab,2,max)

##     red     blue     white     silver     black
##       3        1         2          2         0
```

A useful function is `which.max`. Given a list of numbers, it returns the index of the largest one. For example:

```
example <- c(1.4,2.6,1.1,1.5,1.2)
which.max(example)

## [1] 2
```

so that in this case the second element is the largest.

Self-Test Question 5. What happens if there is more than one number taking the largest value in a list? Either use the help facility or experimentation to find out.

Self-Test Question 6. `which.max` can be used in conjunction with `apply`. Write an expression to find the index of the largest value in each row of `crosstab`.

The function `levels` returns the levels of a variable of type `factor` in character form. For example:

```
levels(engine)

## [1] "1.1litre" "1.3litre" "1.6litre"
```

The order they are returned in is that specified in the original `factor` assignment and the same order as row or column names produced by the `table` function. This means that it can be used in conjunction with `which.max` when applied to matrices to obtain the row or column names instead of an index number:

```
levels(colours)[which.max(crosstab[,1])]

## [1] "blue"
```

Alternatively, the same effect can be achieved by the following:

```
colnames(crosstab)[which.max(crosstab[,1])]

## [1] "blue"
```

You should unpick these last two lines of code to make sure you understand what each element is doing.

```
colnames(crosstab)

## [1]  "red"   "blue"   "white"   "silver"   "black"

crosstab[,1]

##       saloon  hatchback  convertible
##           2          3            0

which.max(crosstab[,1])

## hatchback
##         2
```

More generally, a function could be written to apply this operation to any variable with names:

```
# Defines the function
which.max.name <- function(x) {
    return(names(x)[which.max(x)])}
# Next, give the variable 'example' names for the values
names(example) <- c("Leicester","Nottingham",
     "Loughborough","Birmingham","Coventry")
example

## Leicester  Nottingham  Loughborough  Birmingham  Coventry
##        1.4         2.6           1.1         1.5       1.2

which.max.name(example)

## [1] "Nottingham"
```

Self-Test Question 7. Finally, `which.max.name` could be applied (using `apply`) to a matrix, to find the row name with the largest value, for each of the columns, or vice versa. For the car sales matrix, this would give you the best-selling colour for each car type (or vice versa). Write the `apply` expression for each of these.

Note that in the last code snippet, a `function` was defined called `which.max.name`. You have been using functions, but these have all been existing ones as defined in R until now. Functions will be thoroughly dealt with in Chapter 4, but you should note two things about them at this point. First is the form:

```
function name <- function(function inputs) {
  variable <- function actions
  return(variable)
}
```

Second are the syntactic elements of the curly brackets `{ }` that bound the code and the `return` function that defines the value to be returned by the function.

Lists

From the text in this chapter, recall that lists can be named and unnamed. Here we will only consider the named kind. Lists may be created by the `list` function. To create a list variable enter:

```
var <- list(name1=value1, name2=value2,...)
```


Note that the above is just a template used as an example – entering it into R will give an error as there are no variables called `value1`, `value2`, etc., and the dots `...` in this context are not valid R syntax.

Self-Test Question 8. Suppose you wanted to store both the row- and column-wise `apply` results (from Question 7) into a list called `most.popular` with two named elements called `colour` (containing the most popular colour for each car type) and `type` (containing the most popular car type for each colour). Write an R expression that shows the best-selling colour and car types into a list.

Classes

The objective of this task is to create a class based on the list created in the last section. The class will consist of a list of most popular colours and car types, together with a third element containing the total number of cars sold (called `total`). Call this class `sales.data`. A function to create a variable of this class, given `colours` and `car.type`, is below:

```
new.sales.data <- function(colours, car.type) {
      xtab <- table(car.type,colours)
      result <- list(colour=apply(xtab,1,which.max.name),
          type=apply(xtab,2,which.max.name),
          total=sum(xtab))
      class(result) <- "sales.data"
      return(result)}
```

This can be used to create a `sales.data` object which has the `colours` and `car.type` variables assigned to it via the function:

```
this.week <- new.sales.data(colours,car.type)
this.week

## $colour
##      saloon  hatchback  convertible
##       "red"      "red"      "white"
##
## $type
##         red       blue        white     silver      black
## "hatchback"   "saloon"     "saloon"   "saloon"   "saloon"
##
## $total
## [1] 13
##
## attr(,"class")
## [1] "sales.data"
```

In the above code, a new variable called `this.week` of class `sales.data` is created. Following the ideas set out in the preceding section, it is now possible to create a `print` function for variables of class `sales.data`. This can be done by writing a function called `print.sales.data` that takes an input or argument of the `sales.data` class.

Self-Test Question 9. Write a print function for variables of class `sales.data`. This is a difficult problem and should be tackled by those with previous programming experience. Others can try it now, but should return to it after the functions have been formally introduced in the next chapter.

2.4 PLOTS

There are a number of plot routines and packages in R. In this section some basic plot types will be introduced, followed by some more advanced plotting commands and functions. The aim of this section is to give you an understanding of how the basic plot types can be used as building blocks in more advanced plotting routines that are called in later chapters to display the results of spatial analysis.

2.4.1 Basic Plot Tools

The most basic plot is the scatter plot. Figure 2.1 was created from the function `rnorm` which generates a set of random numbers.

```
x1 <- rnorm(100)
y1 <- rnorm(100)
plot(x1,y1)
```

The generic `plot` function creates a graph of two variables that are plotted on the *x*-axis and the *y*-axis. The default settings for the `plot` function produce a

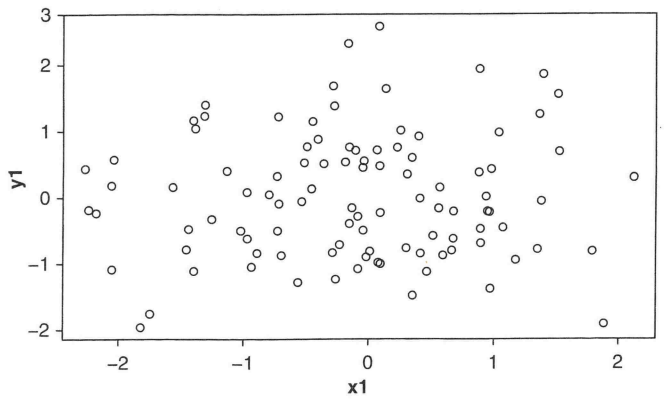

Figure 2.1 A basic scatter plot

scatter plot, and you should note that by default the axes are labelled with expressions passed to the `plot` function. Many parameters can be set for `plot` either by defining the plot environment (described later) or when the plot is called. For example, the option `col` specifies the plot colour and `pch` the plot character:

```
plot(x1,y1,pch=16, col='red')
```

Other options include different types of plot: `type = 'l'` produces a line plot of the two variables, and again the `col` option can be used to specify the line colour and the option `lwd` specifies the plot line width. You should run the code below to produce different line plots:

```
x2 <- seq(0,2*pi,len=100)
y2 <- sin(x2)
plot(x2,y2,type='l')
plot(x2,y2,type='l', lwd=3, col='darkgreen')
```

You should examine the help for the `plot` command (reminder: type `?plot` at the R prompt) and explore different plot types that are available. Having called a new plot as in the above examples, other data can be plotted using other commands: `points`, `lines`, `polygons`, etc. You will see that `plot` by default assumes the plot type is `point` unless otherwise specified. For example, in Figure 2.2 the line data described by `x2` and `y2` are plotted, after which the points described by `x2` and `y2r` are added to the plot.

```
plot(x2,y2,type='l', col='darkgreen', lwd=3,
ylim=c(-1.2,1.2))
y2r <- y2 + rnorm(100,0,0.1)
points(x2,y2r, pch=16, col='darkred')
```

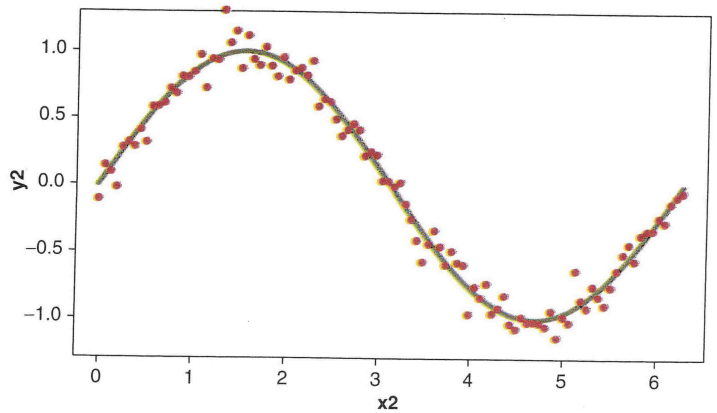

Figure 2.2 A line plot with points added

In the above code, the `rnorm` function creates a vector of small values which are added to `y2` to create `y2r`. The function `points` adds points to an existing plot. Many other options for plots can be applied here. For example, note the `ylim` option. This sets the limits on the *y*-axis, and `xlim` does the same for the *x*-axis. You should apply the commands below to the plot data.

```
y4 <- cos(x2)
plot(x2, y2, type='l', lwd=3, col='darkgreen')
lines(x2, y4, lwd=3, lty=2, col='darkblue')
```

Notice that, similar to `points`, the function `lines` adds lines to an existing plot. Note also the `lty` option: this specifies the type of line (dotted, simple, etc.) and is described in the plot parameters below.

You should examine the different types (and other plot parameters) in `par`. Enter `?par` for the help page to see the full list of different plot parameters. One of these, `mfrow`, is used below to set a combined plot of one row and two columns. This needs to be reset or the rest of your plots will continue to be printed in this way. To do this enter:

```
par(mfrow = c(1,2))
# reset
par(mfrow = c(1,1))
```

The function `polygon` adds a polygon to the plot. The option `col` sets the polygon fill colour. By default a black border is drawn; however, including the parameter `border = NA` would result in no border being drawn. In Figure 2.3 two different plots of the same data illustrate the application of these parameters.

```
x2 <- seq(0,2*pi,len=100)
y2 <- sin(x2)
y4 <- cos(x2)
# specify the plot order; see ?par for more information
par(mfrow = c(1,2))
# plot #1
plot(y2,y4)
polygon(y2,y4,col='lightgreen')
# plot #2: this time with 'asp' to set the aspect ratio
of the axes
plot(y2,y4, asp=1, type='n')
polygon(y2,y4,col='lightgreen')
```


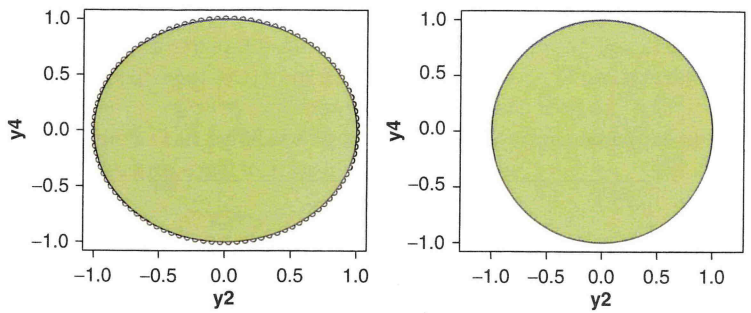

Figure 2.3 Points with polygons added

The parameter `asp` fixes the aspect ratio, in this case to 1 so that the `x` and `y` scales are the same, and `type = 'n'` draws the plot axes to correct scale (i.e. of the `y2` and `y4` data) but adds no lines or points.

So far the plot commands have been used to plot pairs of *x* and *y* coordinates in different ways: points, lines and polygons (this may suggest different vector types in a GIS for some readers). We can extend these to start to consider geographic coordinates more explicitly with some geographic data. You will need to install the `GISTools` package, which may involve setting a mirror site as described in Chapter 1. The first time you use any package in R it needs to be downloaded before it is installed.

```
install.packages("GISTools", depend = T)
```

Then you can call the package in the R console:

```
library(GISTools)
```

You will see some messages when you load the package, letting you know that the packages that `GISTools` makes use of have also been loaded automatically. The code below loads a number of datasets with the `data(georgia)` command. It then selects the first element from the `georgia.polys` dataset and assigns it to a variable called `appling`. This contains the coordinates of the outline of Appling County in Georgia. It then plots this to generate Figure 2.4. You only need to *install* a package onto your computer the first time you use it. Once it is installed it can simply be called. That is, there is no need to download it again, you can simply enter `library(package)`.

```
# library(GISTools)
data(georgia)
# select the first element
appling <- georgia.polys[[1]]
```

```
# set the plot extent
plot(appling, asp=1, type='n', xlab="Easting",
ylab="Northing")
# plot the selected features with hatching
polygon(appling, density=14, angle=135)
```

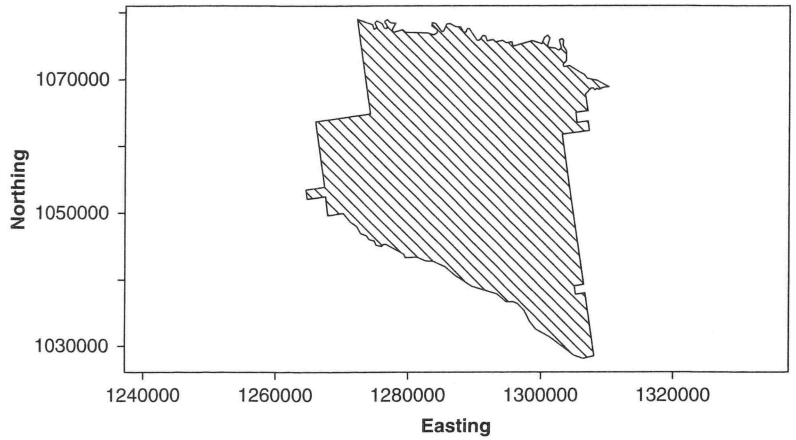

Figure 2.4 Appling County plotted from coordinate pairs

There are a number of things to note in this bit of code.

1. `data(georgia)` loads three datasets: `georgia`, `georgia2` and `georgia.polys`.
2. The first element of `georgia.polys` contains the coordinates for the outline of Appling County.
3. Polygons do not have to be regular: they can, as in this example, be geographical zones. The code assigns the coordinates to a variable called `appling` and the variable is a two-column matrix.
4. Thus, with an `x` and `y` pairing, the following plot commands all work with data in this format: `plot`, `lines`, `polygon`, `points`.
5. As before, the `plot` command in the code below has the `type = 'n'` parameter and `asp` fixes the aspect ratio. The result is that that the *x* and *y* scales are the same but the command adds no lines or points.

2.4.2 Plot Colours

Plot colours can be specified as red, green and blue (RGB) values: that is, three values in the ranges 0 to 1. Having run the code above, you should have a variable called `appling` in your workspace. Now try entering the code below:


```
plot(appling, asp=1, type='n', xlab="Easting",
ylab="Northing")
polygon(appling, col=rgb(0,0.5,0.7))
```

A fourth parameter can be added to `rgb` to indicate transparency as in the code below, where the range is 0 to 1 for invisible to opaque.

```
polygon(appling, col=rgb(0,0.5,0.7,0.4))
```

Text can also be added to the plot and its placement in the plot window specified. The `cex` parameter (for *c*haracter *ex*pansion) determines the size of text. Note that parameters like `col` also work with `text` and that HTML colours also work, such as `"B3B333"`. The code below generates two plots. The first plots a set of random points and then plots `appling` with a transparency shading over the top. The second plots `appling`, but with some descriptive text. The result of applying these plot commands should look like Figures 2.5 and 2.6.

```
# set the plot extent
plot(appling, asp=1, type='n', xlab="Easting", ylab="Northing")
# plot the points
points(x = runif(500,126,132)*10000,
    y = runif(500,103,108)*10000, pch=16, col='red')
# plot the polygon with a transparency factor
polygon(appling, col=rgb(0,0.5,0.7,0.4))
```

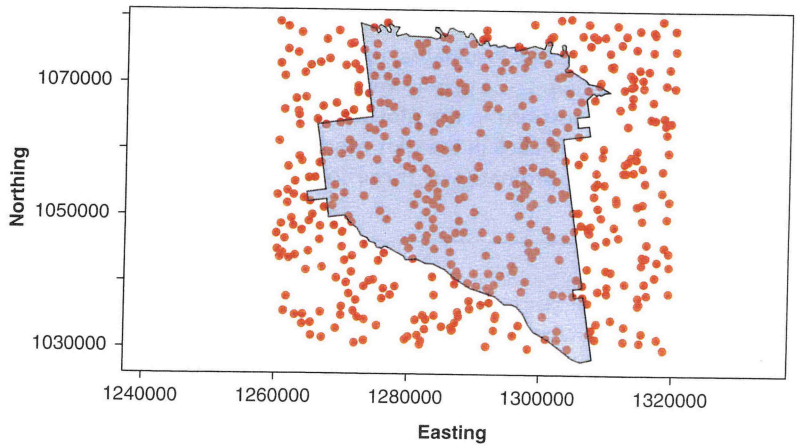

Figure 2.5 Appling County with transparency

```
plot(appling, asp=1, type='n', xlab="Easting",
    ylab="Northing")
polygon(appling, col="#B3B333")
# add text, specifying its placement, colour and size
text(1287000,1053000, "Appling County",cex=1.5)
text(1287000,1049000, "Georgia",col='darkred')
```

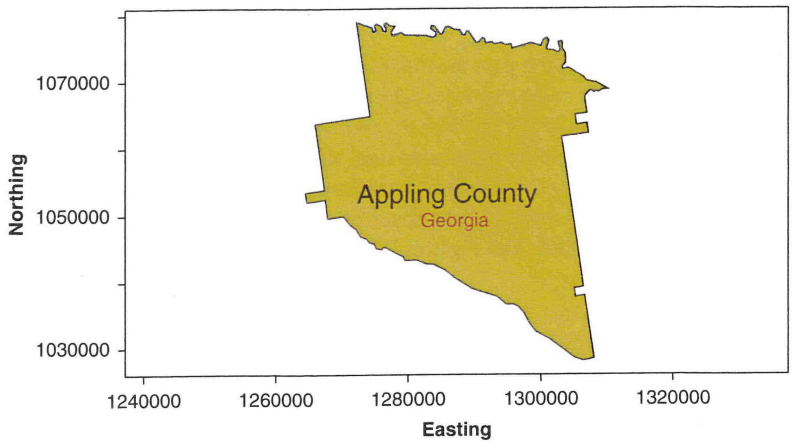

Figure 2.6 Appling County with text

In the above code, the coordinates for the text placement needed to specified. The function `locator` is very useful in this context: it can be used to determine locations in the plot window. Enter `locator()` at the R prompt, and then left-click in the plot window at various locations. When you right-click, the coordinates of these locations are returned to the R console window.

Other plot tools include `rect`, which draws rectangles. This is useful for placing map legends as your analyses develop. The three code blocks below produce the plot in Figure 2.7.

```
plot(c(-1.5,1.5),c(-1.5,1.5),asp=1, type='n')
# plot the green/blue rectangle
rect(-0.5,-0.5,0.5,0.5, border=NA, col=rgb(0,0.5,0.5,0.7))
```


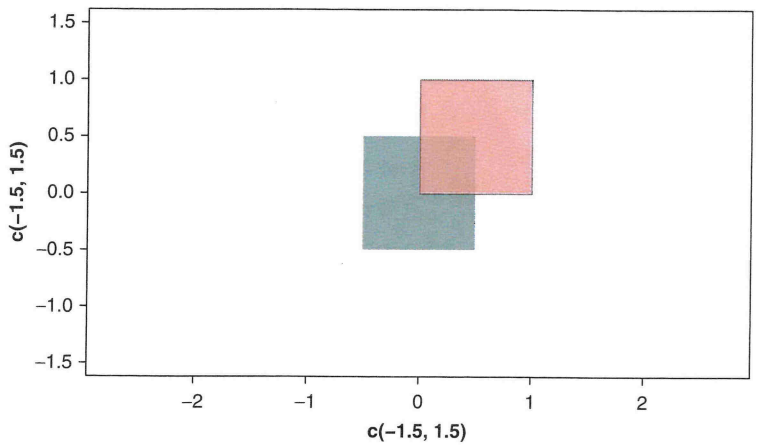

Figure 2.7 Plotting rectangles

```
# then the second one
rect(0,0,1,1, col=rgb(1,0.5,0.5,0.7))
```

The command `image` plots tabular and raster data as shown in Figure 2.8. It has default colour schemes, but other colour palettes exist. This book strongly recommends the use of the `RColorBrewer` package, which is described in more detail in Chapter 3, but an example of its application is given below:

```
# load some grid data
data(meuse.grid)
# define a SpatialPixelsDataFrame from the data
mat = SpatialPixelsDataFrame(points = meuse.grid[c("x", "y")],
      data = meuse.grid)
# set some plot parameters (1 row, 2 columns)
par(mfrow = c(1,2))
# set the plot margins
par(mar = c(0,0,0,0))
# plot the points using the default shading
image(mat, "dist")
# load the package
library(RColorBrewer)
# select and examine a colour palette with 7 classes
greenpal <- brewer.pal(7,'Greens')
# and now use this to plot the data
image(mat, "dist", col=greenpal)
```

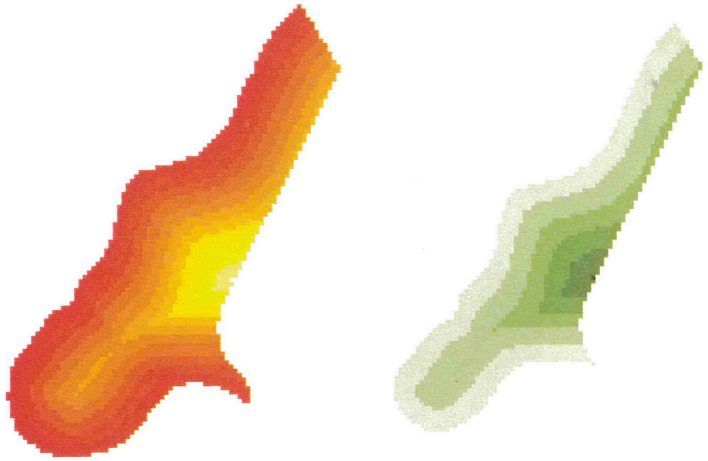

Figure 2.8 Plotting raster data

```
# reset par
par(mfrow = c(1,1))
```

You should note that the `par(mfrow = c(1,2))` results in one row and two columns and that it is reset in the last line of code.

> The command `contour(mat, "dist")` will generate a contour plot of the matrix above. You should examine the help for `contour`. A nice example of its use can be found in code in the help page for the `volcano` dataset that comes with R. Enter the following in the R console:
>
> ```
> ?volcano
> ```

2.5 READING, WRITING, LOADING AND SAVING DATA

There are a number of ways of getting data in and out of R, and three are briefly considered here: reading and writing text files, R data files and spatial data.

2.5.1 Text Files

Consider the `appling` data variable above. This is a matrix variable, containing two columns and 125 rows. You can examine the data using `dim` and `head`:


```
# display the first six rows
head(appling)
# display the variable dimensions
dim(appling)
```

You will note that the data fields (columns) are not named. However, these can be assigned.

```
colnames(appling) <- c("X", "Y")
```

The data can be written into a comma-separated variable file using the command `write.csv` and then read back into a different variable, as follows:

```
write.csv(appling, file = "test.csv")
```

This writes a .csv file into the current working directory. If you open it using a text editor or spreadsheet software, you will see that it has three columns, X and Y as expected plus the index for each record. This is because the default for `write.csv` includes `row.names = TRUE`. Again examine the help file for this function.

```
write.csv(appling, file = "test.csv", row.names = F)
```

R also allows you to read .csv files using the `read.csv` function. Read the file you have created into a variable:

```
tmp.appling <- read.csv(file = "test.csv")
```

Notice that in this case what is read from the .csv file is assigned to the variable `tmp.appling`. Try reading this file without assignment. The default for `read.csv` is that the file has a header (i.e. the first row contains the names of the columns) and that the separator between values in any record is a comma. However, these can be changed depending on the nature of the file you are seeking to load into R. A number of different types of files can be read into R. You should examine the help files for reading data in different formats. Enter `??read` to see some of these listed. You will note that `read.table` and `write.table` require more parameters to be specified than `read.csv` and `write.csv`.

2.5.2 R Data Files

It is possible to save variables that are in your workspace to a designated file. This can be loaded at the start of your next session. For example, if you have been running the code as introduced in this chapter you should have a number of variables, from `x` at the start to `engine` and `colours` and the `appling` data above.

You can save this *workspace* using the drop-down menus in the R interface or using the `save` function. The R interface menu route saves everything that is present in your workspace – as listed by `ls()` – whilst the `save` command allows you to specify what variables you wish to save.

```
# this will save everything in the workspace
save(list = ls(), file = "MyData.RData")
# this will save just appling
save(list = "appling", file = "MyData.RData")
# this will save appling and georgia.polys
save(list = c("appling", "georgia.polys"), file =
    "MyData.RData")
```

You should note that the .RData file binary format is very efficient at storing data: the `appling .csv` file used 4 kb of memory, whilst the .RData file used only 2 kb. Similarly, .RData files can be loaded into R using the menu in the R interface or loaded at the command line at the R console:

```
load("MyData.RData")
```

This will load the variables in the .RData file into the R console.

2.5.3 Spatial Data Files

Very often we have data that is in a particular format, such as *shapefile* format. R has the ability to load many different spatial data formats.

Consider the `georgia` dataset that was loaded earlier. This can be written out as a shapefile in the following way:

```
data(georgia)
writePolyShape(georgia, "georgia.shp",)
```

You will see that a shapefile has been written into your current working directory, with its associated supporting files (`.dbf`, etc.) that can be recognised by other applications (QGIS, etc.). Similarly, this can be read into R and assigned to a variable, provided that a package calling the `writePolyShape` and `readShapePoly` functions in `maptools` such as `GISTools` has been loaded into R:

```
new.georgia <- readShapePoly("georgia.shp")
```

You should examine the `readShapeLines`, `readShapePoints`, `readShapePoly` functions and their associated `write` functions. You should also note that R is able to read and write other proprietary spatial data formats, which

you should be able to find through a search of the R help system or via an internet search engine.

ANSWERS TO SELF-TEST QUESTIONS

Q1. `orange` is not one of the factor's levels, so the result is `NA`.

```
colours[4] <- "orange"
colours

## [1] red blue red <NA> silver red white silver red red
## [11] white silver silver
## Levels: red blue white silver black
```

Q2. There is no count for `'black'` in the character version – `table` does not know that this value exists, since there is no `levels` information. Also the order of colours is alphabetical in the character version. In the factor version, it is based on that specified in the `factor` function.

Q3. The first variable is tabulated along the rows, the second along the columns.

Q4. Colours of all cars with engines with capacity greater than 1.1 litres:

```
# Undo the colour[4] <- "orange" line used above
colours <- factor(c("red","blue","red","white",
    "silver","red","white","silver",
    "red","red","white","silver"),
levels=c("red","blue","white","silver","black"))
colours[engine > "1.1litre"]

## [1] blue white <NA> red white red silver <NA>
## Levels: red blue white silver black
```

Counts of types of all cars with capacity below 1.6 litres:

```
table(car.type[engine < "1.6litre"])

##
##  saloon  hatchback  convertible
##      7          4            0
```

Counts of colours of all hatchbacks with capacity greater than or equal to 1.3 litres:

```
table(colours[(engine >= "1.3litre") & (car.type ==
"hatchback")])

##
##  red  blue  white  silver  black
##    2     0      0       0      0
```

Q5. The index returned corresponds to the *first* number taking the largest value.
Q6. An expression to find the index of the largest value in each row of `crosstab` using `which.max` and `apply`.

```
apply(crosstab,1,which.max)

##  saloon  hatchback  convertible
##      1          1            3
```

Q7. `apply` functions to return the best-selling colour and car type.

```
apply(crosstab,1,which.max.name)

##  saloon  hatchback  convertible
##  "red"      "red"      "white"

apply(crosstab,2,which.max.name)

##         red      blue     white    silver     black
## "hatchback" "saloon" "saloon" "saloon" "saloon"
```

Q8. An R expression that shows the best-selling colour and car types into a list.

```
most.popular <- list(colour=apply(crosstab,1,which.max.name),
    type=apply(crosstab,2,which.max.name))
most.popular

##  $colour
##      saloon  hatchback  convertible
##      "red"      "red"      "white"
##
##  $type
##         red      blue     white    silver     black
##  "hatchback"  "saloon" "saloon" "saloon" "saloon"
```


Q9. A print function for variables of the class `data.frame`.

```
print.sales.data <- function(x) {
    cat("Weekly Sales Data:\n")
    cat("Most popular colour:\n")
    for (i in 1:length(x$colour)) {
      cat(sprintf("%12s:%12s\n",names(x$colour)[i],
      x$colour[i]))}
    cat("Most popular type:\n")
    for (i in 1:length(x$type)) {
      cat(sprintf("%12s:%12s\n",names(x$type)[i],
      x$type[i]))}
    cat("Total Sold = ",x$total)
}

this.week

## Weekly Sales Data:
## Most popular colour:
##       saloon:        red
##    hatchback:        red
##  convertible:      white
## Most popular type:
##          red:  hatchback
##         blue:     saloon
##        white:     saloon
##       silver:     saloon
##        black:     saloon
## Total Sold = 13
```

Although the above is one *possible* solution to the question, it is not unique. You may decide to create a very different-looking `print.sales.data` function. Note also that although until now we have concentrated only on `print` functions for different classes, it is possible to create class-specific versions of *any* function.