R Programming

Data Structures

Statistical Computing

- Statistical computing is a subdiscipline of statistics which brings computer technology to bear on the general problems encountered in statistical data analysis and data processing.
- It typically uses purpose-built computing systems and software components to handle problems in a way which is convenient for statisticians.
- It is differentiated from computational statistics which examines looks at the use of numerically intensive techniques to solve particular statistical problems.

This Course

- In this course we will look at general ideas in statistical computing, but will will focus on a single computing tool – the "R" language and environment.
- The course is very much about programming, but it will be different from those taught in computer science, because of its emphasis on statistical applications.
- Having a computer science background might be helpful, but you will probably need to unlearn a number things.

The R Environment

- R is the outgrowth of some research carried out at Auckland in the early 1990s.
- It was originally a test platform for trying out some ideas about how to implement statistical software.
- There is very little which was original about R – it borrowed heavily from ideas in Lisp and from existing software systems.
- What was novel was the use of a “free software” licence to distribute the resulting software.
- The use of the free licence has encouraged contributions from others and R is now the most fully featured statistical software system in the world.

The R Language

- R is a *collection-oriented* computer language.
- It works with collections of values rather than with individual values.
- The basic types of collection that R manipulates are *vectors* and *lists*.
- A *vector* is a collection of items (or elements) that are all of the same type (numbers, strings, true/false values).
- A *list* is a collection of items (or elements) of (possibly) different types.
- The elements of a list can be vectors, lists or other types of R object.

Vectors

- Vectors have a mode (or type) and a length.

```
> x = c(1, 2, 3, 4)

> length(x)
[1] 4

> mode(x)
[1] "numeric"
```

- The basic modes are: *logical*, *numeric*, *complex* and *character*.
- Vectors have a length which is greater than or equal to 0.

Vector Examples

```
> c(TRUE, FALSE, TRUE, FALSE)
[1]  TRUE FALSE  TRUE FALSE

> c(1, 2, 3, 5, 7)
[1] 1 2 3 5 7

> c(1+2i, -1i)
[1] 1+2i 0-1i

> c("hello", "goodbye")
[1] "hello"   "goodbye"
```

Lists

- Lists provide a way of storing objects of different types in a single container.
- Lists have a length and have mode `"list"`.

```
> x = list(17, "A nice number", TRUE)

> length(x)
[1] 3

> mode(x)
[1] "list"
```

- Lists have a length which is greater than or equal to zero.

List Examples

```
> list()
list()

> list(c(15, 16, 17), "A nice number", TRUE)
[[1]]
[1] 15 16 17

[[2]]
[1] "A nice number"

[[3]]
[1] TRUE
```

List Lengths

- The length of a list is the number of objects it was created from, not the total number of elements in those objects.

```
> x = list(c(1, 2, 3), c(4, 5))
> length(x)
[1] 2
```

- In this case, the length of `x` is 2, not 5.

Creating Vectors and Lists

- The function `vector` can be used to create both lists and vectors.
- The arguments to `vector` are the mode and length of the desired object.
- The vectors and lists created in this way are initialised with "zero" elements.

Examples: Creating Vectors and Lists

```
> vector("logical", 5)
[1] FALSE FALSE FALSE FALSE FALSE

> vector("numeric", 5)
[1] 0 0 0 0 0

> vector("character", 3)
[1] "" "" ""

> vector("list", 2)
[[1]]
NULL

[[2]]
NULL
```

Shortcuts

- The basic vector types have special functions which provide clearer ways of creating these kinds of vector.

```
> logical(4)
[1] FALSE FALSE FALSE FALSE

> numeric(5)
[1] 0 0 0 0 0

> character(3)
[1] "" "" ""
```

- There is no shortcut way of creating lists.

Missing Values

- The indicator `NA` is used to indicate that a value within a vector is missing.

```
> c(1, 2, 3, NA, 4)
[1]  1  2  3 NA  4
```

- The equivalent indicator of missingness for list elements is denoted by `NULL`.

```
> list(10, NULL)
[[1]]
[1] 10

[[2]]
NULL
```

Testing for Missing Values

- The functions `is.na` and `is.null` can be used to test for the presence of missing values.

```
> x = c(1, NA, pi, NA, sqrt(2))

> is.na(x)
[1] FALSE  TRUE FALSE  TRUE FALSE

> is.null(x)
[1] FALSE

> is.null(NULL)
[1] TRUE

> is.null(list(x, NULL))
[1] FALSE
```

Named Vectors and Lists

- It is possible to associate a vector of names with the elements of a vector or list.
- This can be done when a vector is created with `c`, or when a list is created with `list`.

```
> c(a = 1, b = 2, c = 3)
a b c
1 2 3

> list(a = 1:4, b = "string")
$a
[1] 1 2 3 4

$b
[1] "string"
```

Manipulating Names

The names on vectors and lists can be inspected and set with the `names` function.

```
> x = c(a = 1, b = 2, 3)
> names(x)
[1] "a" "b" ""

> names(x) = c("A", "B", "C")
> x
A B C
1 2 3
```

Subsetting

- Sometimes it can be useful to extract a value or a subset of the values in a vector.
- The expression `x[i]` is used to denote a subset of the values in `x` as determined by the *subscript* `i`.
- There are four ways of specifying subsets:
  - by specifying indices,
  - by excluding indices,
  - by true/false selection,
  - by name.

Subsetting Vectors by Specifying Indices

- The subscript `i` consists of non-negative integers.
- Each value in `i` (except `0`) produces a value in the result.
- Subscripts in the range from `1` to `length(x)` produce the corresponding value from `x` in the result.
- Subscripts equal to `0` are ignored and subscripts greater than `length(x)` produce `NA`s.

```
> x = c(10, 20, 30, 40, 50)

> x[c(1, 1, 5, 0, 1, 6)]
[1] 10 10 50 10 NA
```

Subsetting Vectors by Exclusion

- The subscript consists of non-positive values.
- The values in `i` specify values to be excluded from the result.
- Subscripts in the range from `-1` to `-length(x)` cause the corresponding values in `x` to be excluded from the result.
- Subscripts equal to `0` or beyond `-length(x)` are ignored.

```
> x = c(10, 20, 30, 40, 50)

> x[c(-1, -2, 0, -6)]
[1] 30 40 50
```

Subsetting Vectors with Logical (True/False) Values

- The subscript consists of true/false values.
- `TRUE` subscript values cause the corresponding vector values to be included in the result; vector values corresponding to `FALSE` subscript values are omitted.

```
> x = c(10, 20, 30, 40)
> x[c(FALSE, TRUE, FALSE, TRUE)]
[1] 20 40
```

- If the subscript vector is shorter then the vector, its values are "recycled."

```
> x[c(FALSE, TRUE)]
[1] 20 40
```

Subsetting Vectors by Name

- A subscript can contain the names of elements in a vector to be extracted.
- The first element with each of the specified names is extracted.
- Subscripts not corresponding to an element name produce `NA` values in the result.

```
> x = c(a = 1, b = 2, c = 3)

> x[c("a", "b", "h")]
   a    b <NA>
   1    2   NA
```

(Note that the name of the last element is itself `NA`.)

Example: Extracting Subscript Patterns

- Assuming that x is defined by:

```
> x = c(10, 20, 30, 40, 50)
```

- Extract every third value from the vector.

```
> x[1:length(x) %% 3 == 0]
[1] 30

> x[c(FALSE, FALSE, TRUE)]
[1] 30
```

- Extract every third value starting with the second.

```
> x[c(FALSE, TRUE, FALSE)]
[1] 20 50
```

Some Subsetting Idioms

- All but the first value in `x`.

```
x[-1]
```

- All but the last value in `x`.

```
x[-length(x)]
```

- Successive differences.

```
x[-1] - x[-length(x)]
```

- The positive values in `x`.

```
x[x > 0]
```

Subsetting and NA Subscripts

- NA values are not permitted in exclusion (negative subscript) subsetting.
- In all other subsetting operations, any NA subscript will produce a corresponding NA value in the result.

```
> x = c(-1, 0, 1)
> x[c(1, NA)]
[1] -1 NA
```

- What result does the following expression produce and why?

```
> x[NA]
```

Subsetting Idioms Continued

- Remove the `NA` values from a vector.

```
x[!is.na(x)]
```

- The number of positive values in `x`.

```
length(x[!is.na(x) & x > 0])
```

(We have to remove `NA` values or they would count!)

```
> x = c(-1, NA, 1)
> length(x[x > 0])
[1] 2
> length(x[!is.na(x) & x > 0])
[1] 1
```

List Subsetting

- There are two types of subsetting for lists.
  - The expression `x[i]` produces a sub-list of `x` (i.e. the result is a list).
  - The expression `x[[i]]` extracts the `i`-th element from `x` (one element only).
- The same kinds of subsetting can be carried out as in the vector case.

List Subsetting Examples

```
> x = list(a = 1, b = 2)
> x[1]
$a
[1] 1

> x[-1]
$b
[1] 2

> x[[1]]
[1] 1

> x[["a"]]
[1] 1
```

Syntax for List Element Access

- Accessing list elements by name is a very common operation and there is a special syntax for it.
- The expression `x$a` can be used as a shorthand for `x[["a"]]`.
- This is an example of "syntactic sugar."

```
> x[["a"]]
[1] 1
> x[["b"]]
[1] 2
> x$a
[1] 1
> x$b
[1] 2
```

Altering Vector Subsets

- In addition to extracting subsets of vectors and lists the subset notation can be used to specify changes to subsets.
- To do this, the subset to be altered is placed on the right of the assignment symbol ("=" or "<-") and the values to replace those in the subset appear on the right.

```
> x = c(10, 20, 30, 40, 50)
> x[1] = 11
> x
[1] 11 20 30 40 50
> x[x > 30] = c(100, 200)
> x
[1]  11  20  30 100 200
```

Vector Alteration Semantics I

- If too few values are provided on the right-hand side, they are repeated as needed.

```
> x = c(10, 20, 30, 40, 50)
> x[c(1, 2, 3)] = 0
> x
[1]  0  0  0 40 50
```

This is referred to as *value recycling*.

Vector Alteration Semantics II

- If too many values are provided on the right-hand side, the extra values are ignored.

```
> x[1] = c(100, 200, 300)
Warning message:
In x[1] = c(100, 200, 300) :
  number of items to replace is not
  a multiple of replacement length
> x
[1] 100   0   0  40  50
```

List Alteration Semantics

- The use of subsetting to make changes to lists is similar, but not identical, to that for vectors.
- Expressions of the form `x[[i]] = v` can only change one item in a list.
- Expressions of the form

  ```
  x[i] = NULL
  x[[i]] = NULL
  ```

  both *remove* the specified items from lists.
- To replace elements by `NULL`s use expressions of the form

  ```
  x[i] = list(NULL)
  ```

Numeric Vectors

- Numeric vectors contain *floating-point* values.
- These are values of the form

$$\pm\frac{n}{d}$$

  where n and d are positive integers with d a power of 2.
- The values are *binary fractions* which are similar to the decimal fractions which humans are used to working with.
- The values that can be represented in this way are approximations to real numbers.
- There are limits on the size of both n and d which put limits on the set of values which can be represented.

Special Numeric Values

- In addition to binary fractions, there are a number of other special values which R's numeric vectors can contain.
- The value `NA`, represents missing values.
- The values `Inf` and `-Inf` indicate positive and negative infinity.

```
> c(-1/0, 1/0)
[1] -Inf  Inf
```

- The value `NaN` (for not a number) is used to indicate an undefined result for a mathematical operation.

```
> c(0/0, Inf/Inf)
[1] NaN NaN
```

Testing for Special Values

- The functions `is.na`, `is.nan`, `is.infinite` and `is.finite` can be used to test for special values.

```
> x = c(1, NA, NaN, -Inf, Inf)

> is.na(x)
[1] FALSE  TRUE  TRUE FALSE FALSE

> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE

> is.infinite(x)
[1] FALSE FALSE FALSE  TRUE  TRUE

> is.finite(x)
[1]  TRUE FALSE FALSE FALSE FALSE
```

Arithmetic

- R provides the following basic arithmetic operators.

`-x`	negation (minus the values in `x`)
`+x`	unary plus (a no-op)
`x + y`	sum of `x` and `y`
`x - y`	difference of `x` and `y`
`x * y`	product of `x` and `y`
`x / y`	quotient of `x` and `y`
`x ^ y`	exponentiation (raising to a power)
`x %/% y`	integral division
`x %% y`	remainder after integral division

Integral Division and Remainder

- The `%/%` and `%%` operators are less familiar than the other operators.
- They are defined as follows:
  - The value of `x %/% y` is the value of `x / y`, *rounded down* to the next lowest integer.
  - The value of `x %% y` is defined as the value of `x - y * (x %/% y)`.

Integral Division and Remainder Examples

```
> 7 / 3
[1] 2.333333

> 7 %/% 3
[1] 2

> 7 %% 3
[1] 1

> -7 %/% 3
[1] -3

> -7 %% 3
[1] 2
```

Vector Arithmetic

- Because R is designed to work on vectors, we can expect its operations to work for vectors.
- In the case of equal-length vectors, it is natural to define addition as follows:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

  (i.e. elements in the same positions are added.

- All of R's binary operators work in this way.

```
> c(1, 2, 3) / c(.1, .2, .3)
[1] 10 10 10
```

Vector/Scalar Arithmetic

- R's operations also work on mixtures of vectors and scalars.
- The definition of how this works is quite natural.

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} c \end{bmatrix} = \begin{bmatrix} x_1 + c \\ x_2 + c \\ \vdots \\ x_n + c \end{bmatrix}$$

- Again, all of R's operators work in this way.

```
> c(1, 2, 3) + 10
[1] 11 12 13
```

Operating on Unequal-Length Vectors

- What is less obvious about arithmetic is what happens when vectors of different sizes are combined.

Operating on Unequal-Length Vectors

- What is less obvious about arithmetic is what happens when vectors of different sizes are combined.

```
> c(1, 2, 3, 4) + c(1, 2)
[1] 2 4 4 6
```

Operating on Unequal-Length Vectors

- What is less obvious about arithmetic is what happens when vectors of different sizes are combined.

```
> c(1, 2, 3, 4) + c(1, 2)
[1] 2 4 4 6
```

- This result is explained by the *recycling rule* which is used by R to define the meaning of this kind of calculation.

> *First enlarge the shorter vector by recycling its elements, then combine the vectors element by element.*

The Recycling Rule

- For the example:

```
> c(1, 2, 3, 4) + c(1, 2)
[1] 2 4 4 6
```

here is how the recycling rule works.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} \overset{recycle}{\Longrightarrow} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 1 \\ 2 \end{bmatrix} \overset{add}{\Longrightarrow} \begin{bmatrix} 2 \\ 4 \\ 4 \\ 6 \end{bmatrix}$$

Recycling In General

- The recycling rule does not just apply to arithmetic operators.
- Functions of multiple arguments are usually implemented so that they obey the recycling rule.

```
> round(1:4 * pi, 4)
[1]  3.1416  6.2832  9.4248 12.5664

> round(pi, 1:4)
[1] 3.1000 3.1400 3.1420 3.1416

> round(1/c(3, 7, 11), 1:3)
[1] 0.300 0.140 0.091
```

Mathematical Functions

- R has a rich assortment of mathematical functions.

`abs`, `sign`	absolute value and sign
`sqrt`	square roots
`exp`	exponential function
`log`, `log10`	natural and common logarithms
`sin`, `cos`, `tan`	trigonometric functions
`asin`, `acos`, `atan`	inverse trigonometric functions
`factorial`, `choose`	factorial and binomial coefficients
`round`, `signif`	rounding to nearest
`ceiling`, `floor`	rounding up and down

Statistical Tables

- R has a number of functions which play the part of statistical tables.
- The functions compute probability densities, distribution functions and inverse distribution functions.
- The corresponding function names begin with `d`, `p` and `q` followed by an abbreviated form of the distribution name.

`dnorm`, `pnorm`, `qnorm`	Normal distibution
`dchisq`, `pchisq`, `qchisq`	Chi-squared distribution
`dt`, `pt`, `qt`	t distribution
`dF`, `pF`, `qF`	F distribution
`dbinom`, `pbinom`, `qbinom`	Binomial distribution
`dpois`, `ppois`, `qpois`	Poisson disibution

Distribution Examples

- What is the probability that at standard normal random variable is less than 2?

```
> pnorm(2)
[1] 0.9772499
```

- If a coin is tossed 100 times, what is the chance of seeing fewer than 45 heads?

```
> pbinom(44, 100, .5)
[1] 0.1356265
```

- If cars pass a point on a road at at rate of 1 per minute, what is the chance of seeing exactly 5 cars in a five minute interval?

```
> dpois(5, 5)
[1] 0.1754674
```

Summary Functions – `min`, `max` and `range`

- The functions `min` and `max` return the minimum and maximum values contained in any of their arguments, and the function `range` returns a vector of length 2 containing the minimum and maximum of the values in the arguments.

```
> max(1:100)
[1] 100

> max(1:100, Inf)
[1] Inf

> range(1:100)
[1]   1 100
```

Summary Functions – `sum` and `prod`

- The functions `sum` and `prod` compute the sum and product of all the elements in their arguments.

```
> sum(1:100)
[1] 5050

> prod(1:10)
[1] 3628800
```

Summary Functions and NA

- In any of these summary functions the presence of NA and NaN values in any of the arguments will produce a result which is NA and NaN.

```
> min(NA, 100)
[1] NA
```

- NA and NaN values can be disregarded by specifying an additional argument of na.rm=TRUE.

```
> min(10, 20, NA, na.rm = TRUE)
[1] 10
```

Cumulative Summaries

- There are also cumulative variants of the summary functions.

```
> cumsum(1:10)
 [1]  1  3  6 10 15 21 28 36 45 55
> cumprod(1:10)
 [1]       1       2       6      24     120
 [6]     720    5040   40320  362880 3628800
> cummax(1:10)
 [1]  1  2  3  4  5  6  7  8  9 10
> cummin(1:10)
 [1] 1 1 1 1 1 1 1 1 1 1
```

- These cumulative summary functions do not have a `na.rm` argument.

Parallel Summary Functions

- Finally, there are parallel versions of the minimum and maximum functions. These take a number of vector arguments and apply the recycling rule to them, and then compute the summaries across the corresponding values of the arguments.

```
> pmin(c(1, 10), c(10, 2))
[1] 1 2

> pmax(0, c(-1, 0, 1))
[1] 0 0 1
```

- The parallel versions of the minimum and maximum functions do accept an `na.rm` argument.

Generating Sequences

- The `:` operator can be used to generate sequences.

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
```

- The operator can accept start and end values which are non-integer.

```
> 1.5:7.5
[1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5

> 1.5:7
[1] 1.5 2.5 3.5 4.5 5.5 6.5
```

More General Sequences

- The stepsize in sequences created by the `:` operator is always 1.
- The `seq` function makes it possible to produce more general sequences.
- Sequences are generated according to the values of the `from`, `to`, `by` and `length` arguments.
- Any three of these arguments can be used to create a sequence.
- In addition, the `along` argument can be used to produce the full set of indices for a vector.

Sequence Examples

```
> seq(0, 1, by = .1)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
[11] 1.0

> seq(0, 1, length = 11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
[11] 1.0

> seq(0, by = .1, length = 11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
[11] 1.0

> seq(to = 1, by = .1, length = 11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
[11] 1.0
```

Sequence Examples

```
> x = 1:10
> seq(along = x)
 [1]  1  2  3  4  5  6  7  8  9 10

> x = numeric(0)
> seq(along = x)
integer(0)

> 1:length(x)
[1] 1 0
```

Repeating Values

- The `rep` function makes it possible to repeat the values which occur in a vector.
- The first argument to `rep` gives the values to be repeated and the second specifies how many times to repeat them.
- The second argument can be a single value which specifes how many times to repeat the sequence of values in the first argument.
- It can also be a vector (with the same length as the first argument) which specifies how many times to repeat each value in the first argument.

Repetition Examples

```
> rep(1:4, 3)
 [1] 1 2 3 4 1 2 3 4 1 2 3 4

> rep(1:4, c(2, 3, 2, 3))
 [1] 1 1 2 2 2 3 3 4 4 4

> rep(1:4, rep(3, 4))
 [1] 1 1 1 2 2 2 3 3 3 4 4 4

> rep(1:4, each = 3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

Logical Vectors

- Logical vectors contain the values `TRUE` and `FALSE`.

```
> b = c(TRUE, TRUE, FALSE, FALSE)
```

- Extracting and modifying elements of logical vectors takes place in exactly the same way as extracting and modifying subsets of numeric vectors.

```
> b[1:3]
[1]  TRUE  TRUE FALSE

> b[1] = FALSE
> b
[1] FALSE  TRUE FALSE FALSE
```

Generating Logical Values

- Logical values are often produced as the result of assertions made about other types of value.

```
> 3 < 4
[1] TRUE

> 3 > 4
[1] FALSE
```

- Comparison operators can also be applied to vectors of values.

```
> c(1, 2, 3, 4) < 3
[1]  TRUE  TRUE FALSE FALSE
```

(The recycling rule applies here.)

Comparison Operators

- The full set of comparison operators appears below. The operators all return logical values.

`<`	less than
`<=`	less or equal
`==`	equal
`!=`	not equal
`>`	great than
`>=`	greater or equal

Logical Operators

- Logical values can be combined with the boolean operators; `&` which denotes logical "and," `|` which denotes logical "or" and `!` which denotes unary "not."

```
> x = c(1, 2, 3, 4)

> x < 2 | x > 3
[1]  TRUE FALSE FALSE  TRUE
```

- The functions `any` and `all` can be used to see if any or all the elements of a logical vector are true.

```
> all(x > 0)
[1] TRUE

> any(x > 2)
[1] TRUE
```

Logic and NA Values

- Logical vectors can contain NA values. This produces a three-valued logic.

x & y		y		
		TRUE	FALSE	NA
x	TRUE	TRUE	FALSE	NA
	FALSE	FALSE	FALSE	FALSE
	NA	NA	FALSE	NA

x | y		y		
		TRUE	FALSE	NA
x	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	NA
	NA	TRUE	NA	NA

Character Vectors

- Character vectors have elements which are character strings.
- Strings are sequences of characters enclosed in double quotes, `"like this"`, or single quotes, `'like this'`.
- Character vectors can be manipulated just like any other kind of vector.

```
> s = c("first", "second", "third")

> s[1:2]
[1] "first"  "second"

> s[1] = "initial"
> s
[1] "initial" "second"  "third"
```

String Manipulation

- The function `nchar` returns the lengths the character strings in a character vector.

```
> s
[1] "initial" "second"  "third"

> nchar(s)
[1] 7 6 5
```

- Substrings can be extracted with `substring`.

```
> substring(s, 1, 2:3)
[1] "in"  "sec" "th"
```

(Note the use of recycling here.)

Pasting Strings Together

- Strings can be "glued together" by using paste.

```
> paste("First", "Second", "Third")
[1] "First Second Third"
```

- By default, strings are joined with a space between them, but it is possible to use any sequence of characters as a separator.

```
> paste("First", "Second", "Third", sep = ":")
[1] "First:Second:Third"
```

- Using an empty separator string will concatenate the strings.

```
> paste("First", "Second", "Third", sep = "")
[1] "FirstSecondThird"
```

Pasting Vectors

- Pasting will work with vectors as well as simple strings. The result is defined by the recycling rule.

```
> paste(s, "element", sep = "-")
[1] "initial-element" "second-element"
[3] "third-element"
```

- An additional argument, `collapse`, will cause the elements of the result of elementwise pasting to be joined together with the given specified separator.

```
> paste(s, collapse =" -> ")
[1] "initial -> second -> third"
```

Complex-Valued Vectors

Vectors can contain values which a complex numbers. These are written in the standard notation.

```
> z = 10+20i
> z
[1] 10+20i
```

Arithmetic carried out on complex values obeys the rules of complex arithmetic.

```
> z = -1+0i
> sqrt(z)
[1] 0+1i
```

Type Coercion

- Types are automatically converted or *coerced* in R if the need arises.

```
> c(TRUE, 17)
[1]  1 17

> c(TRUE, 17, "twelve")
[1] "TRUE"   "17"     "twelve"
```

- There is a natural ordering of the vector modes in R which provides a natural way of defining coercion automatically.

$$\text{Logical} \rightarrow \text{Numeric} \rightarrow \text{Complex} \rightarrow \text{Character}$$

Automatic Type Coercion

$$\text{Logical} \rightarrow \text{Numeric} \rightarrow \text{Complex} \rightarrow \text{Character}$$

- Logical values can be interpreted as numeric or complex if we take the value of `FALSE` to be 0 and the value of `TRUE` to be 1.
- Numeric values can be interpreted as complex by taking the imaginary parts to be zero.
- All modes can be interpreted as character values by just taking their printed representation.

Type Coercion Idioms

- Coercion is at the heart of many idioms in R.
- A common example is counting the number of vector elements for which a condition is true. For example, the expression `sum(x > 5)` counts the number of elements in `x` which are greater than 5.

```
> sum(x > 5)
[1] 5
```

- The proportion is also easily computed.

```
> sum(x > 5) / length(x)
[1] 0.5
```

Explicit Coercion

- R will automatically coerce vectors using the natural ordering of vector modes. Other coercions must be carried out using explicit coercion with the functions `as.logical`, `as.numeric`, `as.complex` and `as.character`.

```
> "1" + "2"
Error in "1" + "2" : non-numeric argument to
                     binary operator

> as.numeric("1") + as.numeric("2")
[1] 3
```

R Programming

Functions

Functions

- All computations in R are carried out by calling functions.
- A call to an R function takes zero or more arguments and returns a single value.
- Defining functions provides users a way of adding new functionality to R.
- Functions defined by users have the same status as the functions built into R.

A Simple Function

- Here is a simple function that squares its argument.

```
> square = function(x) x * x
```

- This function can be used in exactly the same way as any other R function.

```
> square(10)
[1] 100
```

- Because the * operator acts element-wise on vectors, the new square function will act that way too.

```
> square(1:10)
 [1]   1   4   9  16  25  36  49  64  81 100
```

Functions Defined in Terms of Other Functions

- The `square` function is just like any other R function and can be used in other function definitions.

```
> sumsq = function(x) sum(square(x))
> sumsq(1:10)
[1] 385
```

- As with any other kind of R object, functions can be redefined.

```
> sumsq = function(x) sum(square(x - mean(x)))
> sumsq(1:10)
[1] 82.5
```

Anonymous Functions

- It is usual to assign functions a name when they are created, but it is possible to define and use anonymous functions.

```
> (function(x) x * x)(10)
[1] 100
```

- The parentheses around the function definition are necessary. Without them, the rightmost part of the expression (namely, `x(10)`) would be taken to be a call to the function `x` within the body of the function being defined.

- The most common use-case for anonymous functions is to pass them as arguments to functions.

Example: The `which` Function

- There is a very useful R function, called `which`, that takes a logical vector and returns the indices of the values in the vector that are true.
- Here is `which` being used on a set of random uniform values generated with the `runif` function.

```
> u = runif(5)
> u
[1] 0.2875775 0.7883051 0.4089769 0.8830174
[5] 0.9404673
> which(u > .5)
[1] 2 4 5
> which(.25 < u & u < .75)
[1] 1 3
```

How the `which` Function Works

- Given a set of logical values stored in a vector `x`, for example,

  ```
  > x = u > .5
  ```

  the indices of elements of the vector can be obtained using the expression

  ```
  > 1:length(x)
  [1] 1 2 3 4 5
  ```

- The indices corresponding to the `TRUE` elements of `x` can be selected by using logical subsetting.

  ```
  > (1:length(x))[x]
  [1] 2 4 5
  ```

Evaluation Details

$$\texttt{u} = \begin{pmatrix} 0.2875775 \\ 0.7883051 \\ 0.4089769 \\ 0.8830174 \\ 0.9404673 \end{pmatrix} \qquad \texttt{x} = (\texttt{u > .5}) = \begin{pmatrix} \texttt{FALSE} \\ \texttt{TRUE} \\ \texttt{FALSE} \\ \texttt{TRUE} \\ \texttt{TRUE} \end{pmatrix}$$

$$\texttt{1:length(x)} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}$$

Evaluation Details

$$\texttt{(1:length(x))[x]} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} \left[\begin{pmatrix} \texttt{FALSE} \\ \texttt{TRUE} \\ \texttt{FALSE} \\ \texttt{TRUE} \\ \texttt{TRUE} \end{pmatrix} \right]$$

$$= \begin{pmatrix} 2 \\ 4 \\ 5 \end{pmatrix}$$

Defining the `which` Function

- A simple version of the which function can be defined as follows.

```
> which = function(x) (1:length(x))[x]
```

- The function works as expected.

```
> which(u > .5)
[1] 2 4 5
```

Some Problematic Edge Cases

- The expression `1:length(x)` works well if `x` has non-zero length.
- It does not do what is required when `x` has length 0.

```
> x = logical(0)
> 1:length(x)
[1] 1 0
```

- To avoid this problem it is best to use the alternative expression `seq(along=x)` to generate the indices of the values in `x`.

```
> seq(along = x)
integer(0)
```

Edge Cases Continued

- The which function we have defined assumes that its argument is logical and it can produce strange answers if it is not.

```
> x = c(0, 1, 0, 1)
> which(x)
[1] 1 1
```

- It also doesn't act like the system function when there are `NA` values present.

```
> x = c(TRUE, NA, FALSE)
> which(x)
[1]  1 NA
```

Refining the `which` Function

- Our version of the `which` function can be improved as follows.

```
> which =
      function(x)
      seq(along = x)[!is.na(as.logical(x)) &
                     as.logical(x)]
```

- Repeating the tests shows the problems are fixed.

```
> which(c(0, 1, 0, 1))
[1] 2 4
> which(c("false", "true"))
[1] 2
> which(c(TRUE, NA, FALSE))
[1] 1
```

Functions with Multiple Arguments

- The monthly payment on a loan with principal, p, loan repayment period in years, y, and monthly (percentage) interest rate i, is given by the formula

$$payment = p\frac{(i/1200)(1+i/1200)^{12y}}{(1+i/1200)^{12y}-1}.$$

- The monthly payment depends on p, y and i.
- An R function that computes payments will have three arguments that contain the values of p, y and i.
- Functions with multiple arguments are defined in a very similar way to those with one argument.

The Payment Function

- Implementing the function amounts to simply writing out the formula in R syntax.

```
> payment =
      function(p, y, i)
      p * ((i/1200)*(1+i/1200)^(12*y) /
           ((1+i/1200)^(12*y) - 1))
```

- A modest house in Auckland costs on the order of $900,000 (which requires a 20% down payment of $180,000). With a loan repayment period of twenty years and the current interest rate of 5% the monthly repayments can be worked out as follows.

```
> payment(.8 * 900000, 20, 5)
[1] 4751.681
```

Depressed Yet?

- The expression defining the value of the function is vectorised because arithmetic in R is vectorised. This makes it easy to experiment.
- Let's vary the mortgage term.

```
> payment(.8 * 900000, c(20, 30, 40), 5)
[1] 4751.681 3865.116 3471.816
```

- Next we'll vary the interest rate.

```
> payment(.8 * 900000, 20, 7:2)
[1] 5582.152 5158.304 4751.681 4363.058
[5] 3993.103 3642.360
```

- Sigh! Auckland housing really isn't affordable.

Functions in General

- In general, an R function definition has the form:

  `function (`*arglist*`)` *body*

  where:

  arglist is a (comma separated) list of variable names known as the *formal arguments* of the function,

  body is an expression known as the *body* of the function.

- Functions are usually, but not always, assigned a name so that they can be used in later expressions.

- More complex functions can be constructed by making their argument lists or bodies more complicated.

More Complex Function Bodies

- Functions can be required to carry out some very complex calculations.
- Such calculations can require more that a single simple expression used such as those used in the `which` and `payment` functions.
- In order to create complicated functions we need to know more about the R language and about how to direct the flow of control.

Expressions and Compound Expressions

- R programs are made up of expressions. These can either be simple expressions (of the type we've seen already) or compound expressions consisting of simple expressions separated by semicolons or newlines and grouped within braces.

  { $expr_1$; $expr_2$; ... ; $expr_n$ }

- Every expression in R has a value and the value of the compound expression above is the value of $expr_n$. E.g.

```
> x = { 10 ; 20 }
> x
[1] 20
```

Assignments within Compound Expressions

- It is possible to have assignments within compound expressions and the values of the variables that this produces can be used in later expressions.

```
> z = { x = 10 ; y = x^2; x + y }

> x
[1] 10

> y
[1] 100

> z
[1] 110
```

If-Then-Else Statements

- If-then-else statements make it possible to choose between two (possibly compound) expressions depending on the value of a (logical) condition.

  `if (`*condition*`)` $expr_1$ `else` $expr_2$

- If *condition* is true then $expr_1$ is evaluated otherwise $expr_2$ is executed.

Notes

1. Only the first element in *condition* is checked.
2. The value of the whole expression is the value of which ever expression was executed.

If-Then-Else Examples

- The expression

  ```
  if (x > 0) y = sqrt(x) else y = -sqrt(-x)
  ```

  provides an example of an if-then-else statement which will look familiar to Pascal, Java, C, or C++ programmers.

- The statement can, however, be written more succinctly in R as

  ```
  y = if (x > 0) sqrt(x) else -sqrt(-x)
  ```

  which will look familiar to Lisp or Algol programmers.

If-then Statements

- There is a simplified form of if-then-else statement which is available when there is no $expression_2$ to evaluate. This statement has the general form

  `if (`*condition*`)` *expression*

  and is completely equivalent to the statement

  `if (`*condition*`)` *expression* `else NULL`

Combining Logical Conditions

- The `&` and `|` operators work elementwise on vectors.
- When programming, it is only the first element of logical vectors which is important.
- There are special logical operators `&&` and `||` which work on just the first elements of their arguments.

```
> c(TRUE, FALSE) & TRUE
[1]  TRUE FALSE

> c(TRUE, FALSE) && TRUE
[1] TRUE
```

Combining Logical Conditions

- The `&&` and `||` operate also evaluate just enough of their arguments as is necessary to determine their result.

```
> TRUE && print(TRUE)
[1] TRUE
[1] TRUE

> TRUE || print(TRUE)
[1] TRUE
```

- In the first case both arguments are evaluated and in the second only the first argument is evaluated because the entire statement is known to be true when the first argument is seen to be true.

For Loops

- As part of a computing task we often want to repeatedly carry out some computation for each element of a vector or list. In R this is done with a `for` loop.
- A `for` loop has the form:

  `for(`*variable* `in` *vector*`)` *expression*

- The effect of such a loop is to set the value of *variable* equal to each element of the *vector* in turn, each time evaluating the given *expression*.

For Loop Example 1

- Suppose we have a vector `x` which contains a set of numerical values, and we want to compute the sum of those values. One way to carry out the calculation is to initialise a variable to zero and to add each element in turn to that variable.

```
s = 0
for(i in 1:length(x))
    s = s + x[i]
```

- The effect of this calculation is to successively set the variable `i` equal to each of the values `1`, `2`, ..., `length(x)`, and for each of the successive values to evaluate the expression `s = s + x[i]`.

For Loop Example 2

- The previous example is typical of loops in many computer programming languages, but R does not need to use an integer *loop variable*.
- The loop could instead be written

```
s = 0
for(elt in x)
    s = s + elt
```

- This is both simpler and more efficient.

The "*next*" Statement

- Sometimes, when given condition are met, it is useful to be able to skip to the end of a loop, without carrying out the intervening statements. This can be done by executing a `next` statement when the conditions are met.

```
for(variable in vector) {
  expression1
  expression2
  if (condition)
    next
  expression3
  expression4
}
```

- When *condition* is true, $expression_3$ and $expression_4$ are skipped.

An Example Using "`next`"

- The following example sums just the positive elements of a vector `x`.
- It does this by skipping to the end of the loop when it encounters non-positive values.

```
s = 0
for (elt in x) {
  if (elt <= 0)
    next
  s = s + elt
}
```

- Note that this example is artifical. A better solution is to use the expression `sum(x[x > 0])`.

The "break" Statement

- The *break* statement is similar to the *next* statement but, rather than jumping to the end of the loop, it stops the execution of the loop and jumps to the following statement.

```
for(variable in vector) {
  expression1
  expression2
  if (condition)
    break
  expression3
  expression4
}
```

An Example Using "break"

- The following example sums the elements of a vector, but aborts the computation and sets the sum to `NA` if a non-positive value is encountered.

```
s = 0
for (elt in x) {
  if (elt <= 0) {
    s = NA
    break
  }
  s = s + elt
}
```

While Loops

- For-loops evaluate an expression a fixed number of times.
- It can also be useful to repeat a calculation until a particular condition is false.
- A *while-loop* provides this form of control flow.

  `while (`*condition*`)` *expression*

- Again, *condition* is an expression which must evaluate to a simple logical value, and *expression* is a simple or compound expression.

While Loop Example

- The following example shows how to carry out long division (of integers) by repeated subtraction.

```
> dividend = 22
> divisor = 5

> wholes = 0
> remainder = dividend
> while (remainder > divisor) {
      remainder = remainder - divisor
      wholes = wholes + 1
  }
> c(wholes, remainder)
[1] 4 2
```

- This shows that when 22 is divided by 5 the result is 4 with a remainder of 2.

Repeat Loops

- While loops carry out their termination test at the top of the loop. Sometimes it is necessary to carry out the test in another location.
- This can be done by using a `repeat` loop, that simply repeats an expression indefinitely.

  `repeat` *expression*

- To avoid an *infinite loop*, *expression* must contain a `break` statement to exit the loop.

Repeat and While

- The `while` loop

  `while (`*condition*`)` *expression*

  can be implemented as an equivalent `repeat` loop.

```
repeat {
  if( ! condition ) break
  expression
}
```

- This shows that `repeat` loops are more flexible that `while` loops. However, they are less used because `while` loops are clearer.

A Square Root Algorithm

- There is a very famous method for computing square roots devised by Isaac Newton.
- The method works by taking advantage of the fact that if g is a guess at the square root of x then an improved guess can be obtained by taking the average of g and x/g.
- If g is bigger than $\sqrt{x}$ then x/g will be smaller and conversely.
- By averaging the bigger and smaller values we get a value which is closer to $\sqrt{x}$ than either of them.
- Starting with a guess of 1, we keep improving the guess until it it provides a good enough approximation.

Computing Square Roots

- This small piece of code shows how the computation proceeds.

```
> guesses = numeric(5)
> x = 2
> g = 1
> for (i in 1:5) {
      g = .5 * (g + x/g)
      guesses[i] = g
  }
> guesses
[1] 1.500000 1.416667 1.414216 1.414214
[5] 1.414214
```

- Roughly speaking, the number of correct digits doubles with every iteration.

A Square-Root Function

- Computing square-roots by Newton's method can't be done with a single R statement, so we have to make the body of the function a compound statement, enclosed between `{` and `}`.

```
> root =
      function(x) {
          g = 1
          for (i in 1:5)
              g = .5 * (g + x/g)
          g
      }

> root(2)
[1] 1.414214
```

Improving the Function

- The `root` function always does 5 improvements of the initial guess.
- A better strategy is to keep improving the guess until it is "sufficiently close" to the square root of `x`.
- There are two possible measures of closeness: absolute difference and relative difference.

$$\text{absolute difference} = |x - g^2|,$$

$$\text{relative difference} = \frac{|x - g^2|}{x}.$$

- It is relative difference that is usually used because it always delivers a high degree of accuracy, regardless of the size of x.

A Square-Root Function

- Here is an improved version of the square root function.

```
> root =
      function(x) {
          g = 1
          while(abs(1 - g^2/x) > 1e-10)
              g = .5 * (g + x/g)
          g
      }

> root(2)
[1] 1.414214
> root(2) - sqrt(2)
[1] 1.594724e-12
```

Returning Values

- So far, our functions have used the structure of the code to decide which value will be returned (the value is that of the last expression in the function body).
- It possible to return a value from anywhere inside a function using a call to the `return` function.
- The call has the form:

  `return(`*value*`)`

- Calls to `return` function should be used sparingly (if at all) because they make it harder to understand the structure of code.
- In essence, a call to `return` amounts to performing a `goto`, which is generally frowned upon in programming.

Evaluation of Functions

- Function evaluation takes place as follows:

  (i) Temporarily create a set of variables by associating the arguments passed to the function with the variable names in *arglist*.

  (ii) Use these variable definitions to evaluate the function body.

  (iii) Remove the temporary variable definitions.

  (iv) Return the computed values.

A Simple Example

- Here is a simple function which, given the values a and b, computes the value of $\sqrt{a^2+b^2}$.

```
> hypot = function(a, b) sqrt(a^2 + b^2)
```

- The formal arguments to the function are `a` and `b`.
- The body of the function consists of the simple expression `sqrt(a^2 + b^2)`.
- The function has been assigned the name "`hypot`."

Evaluation Example

- Evaluating the function call

  ```
  hypot(3, 4)
  ```

  takes place as follows:

  (i) Temporarily create variables, `a` and `b`, that have the values `3` and `4`.

  (ii) Use these values to compute the value (5) of `sqrt(a^2 + b^2)`.

  (iii) Remove the temporary variable definitions.

  (iv) Return the value 5.

Optional Arguments

- It is possible to declare default values for arguments so that specifying values for those arguments is optional
- The default values are specified by following the argument by an = sign followed by an expression which defines the value.
- The expressions defining default values can include variables which are arguments to the function or which are defined in the body of the function.

Example

- The following function computes sums of squares (a staple quantity in statistical analyses).

```
> sumsq =
      function(x, about = mean(x))
      sum((x - about)^2)
```

- By default the function computes the sum of squared deviations around the sample mean, but the optional second argument makes other possibilities available.

```
> sumsq(1:10)
[1] 82.5

> sumsq(1:10, 0)
[1] 385
```

Lazy Evaluation

- The expressions given as the default values of arguments are not evaluated until they are required.
- This is referred to as *lazy evaluation*.
- In the example:

```
> sumsq =
      function(x, about = mean(x)) {
          x = x[!is.na(x)]
          sum((x - about)^2)
      }
```

The default value for `about` is not computed until it is needed to evaluate the expression `sum((x-about)^2)`.

Vectorisation

- In general, R functions defined on scalar values are expected to operate element-wise on vectors.
- The standard mathematical and string handling functions all obey this general rule.
- Often this kind of *vectorisation* happens naturally as a side effect of the way R operates.
- As an example, the following function which computes the normal probability density is vectorised.

```
> phi =
      function(x)
      exp(-x^2 / 2) / sqrt(2 * pi)
> phi(0:2)
[1] 0.39894228 0.24197072 0.05399097
```

Vectorising Functions

- Sometimes, functions do not vectorise in a simple way.
- As an example, the following function reverses the characters in a string.

```
> strrev =
      function(x)
      paste(substring(x,
                      nchar(x):1,
                      nchar(x):1),
            collapse = "")
```

- This function works when `x` contains a single value, but not when it has multiple values.

Vectorising Functions – Examples

```
> strrev("foobar")
[1] "raboof"

> strrev(c("foo", "bar"))
[1] "oaf"
Warning messages:
1: In nchar(x):1 :
  numerical expression has 2 elements:
  only the first used
2: In nchar(x):1 :
  numerical expression has 2 elements:
  only the first used
```

Vectorising Functions – Techniques

- A general method for vectorising functions is to run a loop computing the result element by element.
- This can done using a “helper function” or by running the loop “inline.”
- The “helper function” method is the better general purpose technique because it divides the program logic into two simple parts.
- The “inline” method can be shorter, because it uses a single function, but is usually more complicated.

Vectorising Functions – Helper Function Approach

```
> strrev1 =
      function(x)
      paste(substring(x,
                      nchar(x):1,
                      nchar(x):1),
            collapse = "")
> strrev =
      function(x) {
          ans = character(length(x))
          for(i in 1:length(x))
              ans[i] = strrev1(x[i])
          ans
      }
> strrev(c("foo", "bar"))
[1] "oof" "rab"
```

Vectorising Functions – Inline Approach

```
> strrev =
      function(x) {
          ans = character(length(x))
          for(i in 1:length(x))
              ans[i] = paste(substring(x[i],
                             nchar(x[i]):1,
                             nchar(x[i]):1),
                             collapse = "")
          ans
      }

> strrev(c("foo", "bar"))
[1] "oof" "rab"
```

Vectorising Functions with Multiple Arguments

- The process of vectorising functions of multiple arguments is similar to the one argument case.
- The major complication is ensuring that the arguments satisfy the recycling rule.
- This is done by using the `rep` function to recycle the shorter arguments.

Vectorising a Two-Argument Numeric Function

Assuming that `f1` is a numerical function that is not vectorised, the following code creates a vectorised version.

```
f =
    function(x, y) {
        n = max(length(x), length(y))
        if (length(x) < n) x = rep(x, length = n)
        if (length(y) < n) y = rep(y, length = n)
        ans = numeric(n)
        for(i in seq(along = ans))
            ans[i] = f1(x[i], y[i])
        ans
    }
```

A Packaged Solution

- The function `Vectorize` can be used to create a vectorised version of a function.
- This uses some very high-level capabilities that R has to reason about code.

```
f = Vectorise(f1)
```

- It is possible to specify which arguments are to be vectorized (i.e. obey the recycling rule).

Aside: The `lapply` Function

- The function `lapply` applies a given function to each element of a list and returns the computed values in a list.

```
> lapply(list(a = 2, b = 3), sqrt)
$a
[1] 1.414214

$b
[1] 1.732051
```

- Essentially, `lapply` runs a loop to carry out its computation.

An `lapply` Implementation

```
> mylapply =
      function(lst, fun) {
          lst = as.list(lst)
          ans = vector("list", length(lst))
          names(ans) = names(lst)
          for(i in 1:length(lst))
              ans[i] = list(fun(lst[[i]]))
          ans
      }
> mylapply(list(a = 2, b = 3), sqrt)
$a
[1] 1.414214

$b
[1] 1.732051
```

The `sapply` Function

- The `sapply` function behaves just like the `apply` function, except that it tries to simplify its result into a vector or array.

```
> sapply(list(a = 2, b = 3), sqrt)
       a        b
1.414214 1.732051
```

- The result is a numeric vector with named elements.
- The `sapply` function can be used to produce a vectorised version of functions of a single variable.

Vectorization with `sapply`

- The loop in vectorised version of a function can be replaced by a call to `sapply` (which itself contains a loop).

```
> strrev1 =
      function(s)
      paste(substring(s, nchar(s):1,
                      nchar(s):1),
            collapse = "")
> strrev =
      function(x)
      sapply(x, strrev1)

> strrev(c("foo", "bar"))
  foo   bar
"oof" "rab"
```

A Special Technique for Functions of One Variable

- An alternative to using `sapply` is to use `lapply` and to simplify the result directly.

```
> strrev =
      function(x)
      unlist(lapply(x, strrev1))

> strrev(c("foo", "bar"))
[1] "oof" "rab"
```

- This is the simplest and fastest way of vectorising a function of one variable that is only defined for scalars.

  (The efficiency comes about because `lapply` is implemented in a way that avoids loop overhead.)

Statistical Computations

The formula for the standard two sample t statistic is

$$t = \frac{\bar{y}_1 - \bar{y}_2}{s_p \sqrt{\dfrac{1}{n_1} + \dfrac{1}{n_2}}},$$

where s_p is the pooled estimate of the standard deviation defined by

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}.$$

Here n_1, $\bar{y}_1$ and s_1 are the sample size, mean and standard deviation of the first sample and n_2, $\bar{y}_2$ and s_2 are the sample size, mean and standard deviation for the second sample.

Key Quantities in the Computation

The two-sample t-test can be thought of comparing the difference of the two sample means

$$\bar{y}_1 - \bar{y}_2$$

to its standard error

$$se = s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}},$$

where

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}.$$

The computation of these quantities can be thought of sub-tasks which must be carried out so that the t-statistic can be computed.

Implementing the *t* Test in R

We'll assume that we are given the two samples in vectors `y1` and `y2`. The sub-tasks can then be implemented as follows.

```
diff.means =
    function(y1, y2) mean(y1) - mean(y2)

pooled.se =
    function(y1, y2) {
        n1 = length(y1)
        n2 = length(y2)
        sqrt((((n1 - 1) * var(y1) +
               (n2 - 1) * var(y2)) /
               (n1 + n2 - 2)) * (1/n1 + 1/n2))
    }
```

Computing the t-Statistic

Because we have code which computes the required components of the t-statistic it is now very easy to compute the statistic itself.

```
tval = diff.means(y1, y2) /
       pooled.se(y1, y2)
```

By itself, the t-statistic is of little interest. We also need to compute the p-value.

Computing p Values

The p-value for the t statistic is the probability of observing a value as extreme as `tval` or more extreme for a $t_{n_1+n_2-2}$ distribution. This can be computed using one of the built-in distribution functions in R. For a two-sided test the p-value is

$$2P(T_{n_1+n_2-2} < -|\texttt{tval}|).$$

This can be computed with the R statement

```
pval = 2 * pt(- abs(tval), n1 + n2 - 2)
```

The Complete R Function

Here is the complete R function for the two-sided *t* test.

```
ttest =
    function(y1, y2)
    {
        tval = diff.means(y1, y2) /
               pooled.se(y1, y2)
        pval = 2 * pt(- abs(tval), n1 + n2 - 2)
        list(t = tval, df = n1 + n2 - 2,
             pval = pval)
    }
```

Using The *t*-Test Function

Here is a very simple application of the *t*-test.

```
> y1 = rnorm(10)
> y2 = rnorm(10) + 2

> ttest(y1, y2)
$t
[1] -3.206807

$df
[1] 18

$pval
[1] 0.004888771
```

Sample Sizes from Percentages

An advertisement quoted a table showing the percent of IT managers considering the purchase of particular brands of computer. It showed the following table.

Rank	Vendor	Percent
1	A	14.6%
2	B	12.2%
3	C	12.2%
4	D	7.3%
5	E	7.3%

The granularity of this data and the fact that the largest value is exactly twice the smallest one suggests that the sample size was very small. Is it possible to infer what the sample size was from the values in the table?

Problem Analysis

The percentages in the table correspond to fractions f of the form i/n, where n is the sample size. We want to determine n, the denominator of the fractions and the vector of i values corresponding to the values in the table.

Let's assume that the fractions f have been rounded to d decimal places (in this case $d = 3$). In other words, the fractions are accurate to $\pm\varepsilon = 1/2 \times 10^{-d} = .0005$. In symbols

$$(f - \varepsilon) \leq \frac{i}{n} \leq (f + \varepsilon)$$

and it follows that

$$n(f - \varepsilon) \leq i \leq n(f + \varepsilon).$$

Algorithm

We can determine the sample size n by searching for the smallest value of n for which it is possible to find i values so that this expression is true for all the f values.

For $n = 1, 2, 3, \ldots$ compute i as the rounded value of $n * f$ and then test to see whether

$$n(f - \varepsilon) \leq i \leq n(f + \varepsilon).$$

Stop when the first such n is found.

The Algorithm as an R Function

```
> find.denom =
      function(f, places) {
          eps = .5 * 10^-places
          n = 1
          repeat {
              i = round(n * f)
              if (all(n * (f - eps) <= i &
                      i <= n * (f + eps)))
                  break
              n = n + 1
          }
          n
      }
```

Results

The fractions corrsponding to the percentages 14.6%, 12.2% and 7.3% are: 0.146, 0.122 and 0.073. We can run the search with these fractions as follows.

```
> find.denom(c(.146, .122, .073), 3)
[1] 41
```

The actual i values are:

```
> round(41 * c(.146, .122, .073))
[1] 6 5 3
```

Determining the Number of Digits

The function `find.denom` requires that the number of significant digits be passed as an argument. This argument isn't actually required, as the value can be computed from the fractions themselves.

Carrying out the computations can done using the function `round` which rounds values to a given number of decimal places.

```
> 1/3
[1] 0.3333333
> round(1/3, 4)
[1] 0.3333
```

Determining the Number of Digits

We could test for a specified number of digits as follows:

```
> x = .146
> round(x, 2) == x
[1] FALSE
> round(x, 3) == x
[1] TRUE
```

There is a danger that this test of equality will fail because of roundoff errors (which we'll discuss more later). A better test uses a small numerical threshold.

```
> eps = 1e-7
> abs(x - round(x, 2)) < eps
[1] FALSE
> abs(x - round(x, 3)) < eps
[1] TRUE
```

An R Function

```
> places =
      function(f, eps = 1e-7) {
          places = 0
          repeat {
              if (all(abs(f - round(f, places))
                      < eps))
                  break
              places = places + 1
          }
          places
      }

> places(.146)
[1] 3
```

An Improved `find.denom` Function

```
> find.denom =
      function(f) {
          eps = .5 * 10^-places(f)
          n = 1
          repeat {
              i = round(n * f)
              if (all((f - eps) * n <= i &
                      (f + eps) * n >= i))
                  break
              n = n + 1
          }
          n
      }
```

Improving Performance

In most cases there is very little point in starting the denominator search with `n = 1`. It is possible to start the search at the value

```
n = floor(1/min(f) + eps)
```

which corresponds to `min(f)` of `1/n`. In the example, this would be

```
> floor(1/.073 + .0005)
[1] 13
```

which would correspond to a saving of 25% in the computation time. Even better savings are possible.

Invisible Return Values

All R functions return a value. It is possible to make the value returned by a function be non-printing by returning it as the value of the `invisible` function.

```
> no.print =
      function(x)
      invisible(x^2)

> no.print(1:10)

> x = no.print(1:10)
> x
 [1]   1   4   9  16  25  36  49  64  81 100
```

Variable Numbers of Arguments

R functions can be defined to take a variable number of arguments. The special argument name `...` will match any number of arguments in the call (which are not matched by any other arguments).

The mean function computes the mean of the values in a single vector. We can easily create an equivalent function which will compute the mean of all the values in all its arguments.

```
> mean.of.all =
      function(...)
      mean(c(...))

> mean.of.all(1:3, 1:5)
[1] 2.625
```

Restrictions on . . .

- Ony one . . . can be used in the formal argument list for a function.
- The only thing which can be done with . . . inside a function is to pass it as an argument to a function call.
- Arguments which follow . . . in a formal argument list most be specified by name, and the name cannot be abbreviated.

Using ...

The following function assembles its arguments (twice) into a vector.

```
> c2 =
      function(...)
      c(..., ...)


> c2(1, 2, 3)
[1] 1 2 3 1 2 3
```

Notice that argument names are passed along with ....

```
> c2(a = 1, b = 2)
a b a b
1 2 1 2
```

Example

The following function computes the (trimmed) mean of the means of several vectors.

```
> mean.of.means =
      function(..., trim = 0) {
          x = list(...)
          n = length(x)
          means = numeric(n)
          for (i in 1:n)
              means[i] = mean(x[[i]], trim = trim)
          mean(means)
      }

> mean.of.means(1:3, 1:5)
[1] 2.5
```

Abandoning Computations

Sometimes a situation arises during a computation where it is necessary to simply give up and abandon the computation.

There is an R function called `stop` which makes this easy.

The argument to `stop` is a character string which explains why the computation is being stopped.

```
> fake.fun =
      function(x)
      if (x > 10) stop("bad x value") else x

> y = fake.fun(15)
Error in fake.fun(15) : bad x value
```

Warnings

Occasionally, situations arise where it could be that something has gone wrong, but it isn't completely clear that it has. In such a situation it can be useful to continue the computation, but to also alert the user that there is a potential problem. The `warning` function can be used to issue warnings in these cases.

```
if (any(wts < 0))
    warning("negative weights encountered")
```

R Programming

Palindromes

An Extended Example: Palindromic Numbers

- A word is *palindrome* if does not change when its letters are reversed.
- Common examples are: *civic*, *level*, *rotator*, *rotor*, *kayak*, *racecar*.
- Phrases and sentences can also be palindromes (in these cases spaces, letter case and punctuation are ignored).
- Two famous examples are:

  Able was I ere I saw Elba.
  A man, a plan, a canal – Panama!

- An (integer) number is a palindrome if reversing its digits produces and identical value.
- The values *1*, *121*, *24642*, etc., are palindromes.

Checking for Palindromes

- There is a simple way to check whether a word is a palindrome:

  > Reverse the letters in the word and see if the result is the same as the original word.

- This can be done in steps.
  - Extract the letters from the word.
  - Reverse their order.
  - Paste them back together into a word.
  - Compare with the original word.
- This can be done with the functions `substring` and `paste`.

Palindromes Checking Code

```
> x = "foobar"

> substring(x, 1:nchar(x), 1:nchar(x))
[1] "f" "o" "o" "b" "a" "r"

> substring(x, nchar(x):1, nchar(x):1)
[1] "r" "a" "b" "o" "o" "f"

> paste(c("f", "o", "o", "b", "a", "r"),
        collapse = "")
[1] "foobar"

> paste(substring(x, nchar(x):1, nchar(x):1),
        collapse = "")
[1] "raboof"
```

A Palindrome Checking Function

```
> strrev =
      function(x)
      paste(substring(x, nchar(x):1, nchar(x):1),
            collapse = "")

> is.palindrome =
      function(x)
      x[1] == strrev(x[1])

> is.palindrome("foobar")
[1] FALSE

> is.palindrome("racecar")
[1] TRUE
```

Checking for Numeric Palindromes

- The `is.palindrome` function is designed to work on character strings but, because of the magic of automatic coercion, it also works on numbers.

```
> is.palindrome(123)
[1] FALSE

> is.palindrome(12321)
[1] TRUE
```

- This happens because the `substring` function converts its first argument into a string and because the comparison `x[1] == strrev(x[1])`, between a number and a character string, converts the number to a character string before the comparison is made.

Code for Checking Numerical Palindromes

- The following code is designed to check whether a number is a palindrome.

```
> revdigits =
      function(n)
      as.numeric(strrev(as.numeric(n)))

> is.palindrome =
      function(n)
      n[1] == revdigits(n[1])

> is.palindrome(123)
[1] FALSE

> is.palindrome(121)
[1] TRUE
```

The Palindromic Level of Numbers

- If a number is not a palindrome, we can try to convert it into one by adding it to the value obtained by reversing its digits.

$$19 + 91 \rightarrow 110$$

- If this does not produce a palindrome, the process can be repeated.

$$110 + 011 \rightarrow 121$$

- The *palindromic level* of a number is the number of times that digit reversing and addition must be carried out before a palindrome is produced.
- The palindromic level values such as 2 or 121 is 0.
- The palindromic level of 19 is 2 because two reversals are required to produce a palindrome.

Computing the Palindromic Level

Writing a function to compute the palindromic level of a number is relatively easy.

```
> plevel =
      function(n) {
          level = 0
          while(n != revdigits(n)) {
              level = level + 1
              n = n + revdigits(n)
          }
          level
      }

> plevel(19)
[1] 2
```

A Vectorised Function

- The `plevel` function will only work for a single number which is in conflict with the general R philosophy of making functions work for vectors.
- It is easy to vectorise the function using a `for` loop.

```
> palindromic.level =
      function(n) {
          levels = numeric(length(n))
          for(i in 1:length(n))
              levels[i] = plevel(n[i])
          levels
      }

> palindromic.level(1:20)
 [1] 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 1 1 2 1
```

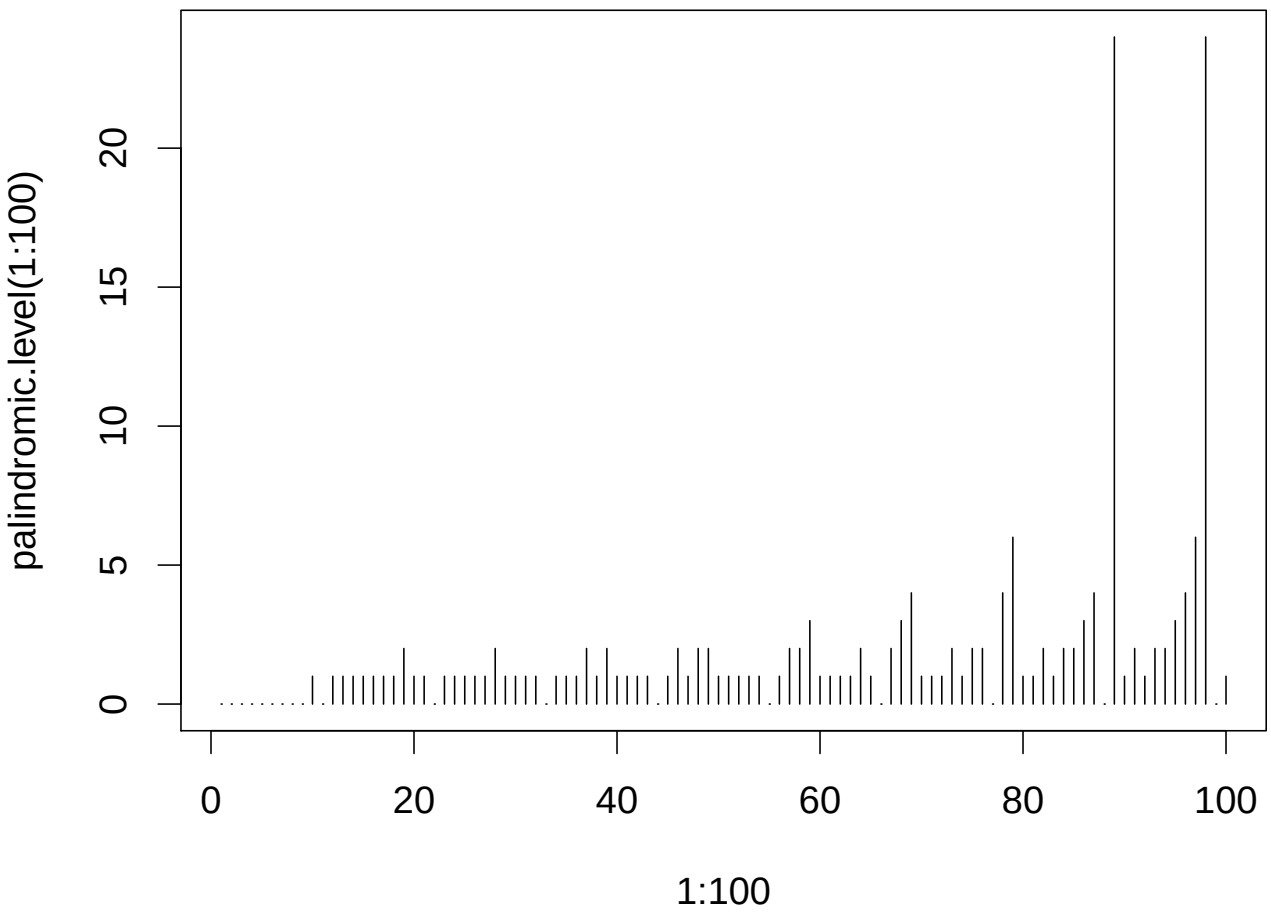
Palindromic Levels

- Using the function we've written it is easy to compute the palindromic levels of the first 100 integers.

```
> palindromic.level(1:100)
 [1]  0  0  0  0  0  0  0  0  0  1  0  1  1
[14]  1  1  1  1  1  2  1  1  0  1  1  1  1
[27]  1  2  1  1  1  1  0  1  1  1  2  1  2
[40]  1  1  1  1  0  1  2  1  2  2  1  1  1
[53]  1  1  0  1  2  2  3  1  1  1  1  2  1
[66]  0  2  3  4  1  1  1  2  1  2  2  0  4
[79]  6  1  1  2  1  2  2  3  4  0 24  1  2
[92]  1  2  2  3  4  6 24  0  1
```

- Plotting the levels can be informative.

```
> plot(1:100, palindromic.level(1:100),
       type = "h")
```


palindromic.level(1:100)
0
5
10
15
20
0
20
40
60
80
100
1:100

Computational Problems

- Although the `palindromic.level` works fine for small values, it has clear problems with larger ones.

```
> palindromic.level(1:200)
Error in while (n != revdigits(n)) { :
  missing value where TRUE/FALSE needed
Calls: palindromic.level -> plevel
In addition: Warning message:
In revdigits(n) : NAs introduced by coercion
```

- There is clearly a problem in `revdigits`, at least some arguments.

Conversion of Numbers to Character Strings

- The process of reversing and adding can produce very big numbers.
- These are converted to scientific notation by `as.character`.

```
> x = 12345678901234567890123456789O
> as.character(x)
[1] "1.23456789012346e+29"
```

- To ensure that scientific notation is not used, a different conversion function must be used.

```
> format(x, scientific = FALSE)
[1] "123456789012345677877719597056"
```

Computational Limits

- The process of reversing and adding can produce very large numbers.
- This is a problem because numbers bigger than 2^{53} may not be stored accurately in the computer.

```
> 2^53 == 2^53 + 1
[1] TRUE
```

- Because of this, we need to introduce a check to see whether numbers have grown too large and, if they have, to return a value that indicates this.
- The test can be implemented as follows:

```
> is.too.big =
      function(n) n >= 2^53
```

Program Modifications

- There are a number of changes which we can make to improve the `plevel` function.
  - Both the integer value and its reversed value must be checked to ensure that they are both accurate. If they are not, an `NA` can be returned.
  - The previous version of `plevel`, reversed the digits in the number twice. By saving the first value, the second reversal can be avoided.

Modified Code

```
> plevel =
      function(n) {
          level = 0
          while(n != (r = revdigits(n))) {
              if (is.too.big(n) ||
                  is.too.big(r)) {
                  level = NA
                  break
              }
              level = level + 1
              n = n + r
          }
          level
      }
```

Palindromic Levels

- The palindromic levels for the numbers between 101 and 200 can computed as follows.

```
> palindromic.level(101:200)
  [1]  0  1  1  1  1  1  1  1  2  1  0  1  1
 [14]  1  1  1  1  1  2  1  0  1  1  1  1  1
 [27]  1  1  2  1  0  1  1  1  1  1  1  1  2
 [40]  1  0  1  1  1  1  1  1  1  2  2  0  2
 [53]  2  2  3  3  3  3  2  2  0  2  2  3  3
 [66]  5 11  3  2  2  0  2  2  4  4  5 15  3
 [79]  2  3  0  6  4  3  3  3 23  7  2  7  0
 [92]  4  8  3  4 NA  7  5  3  1
```

- Notice that the level for 196 could not be computed.

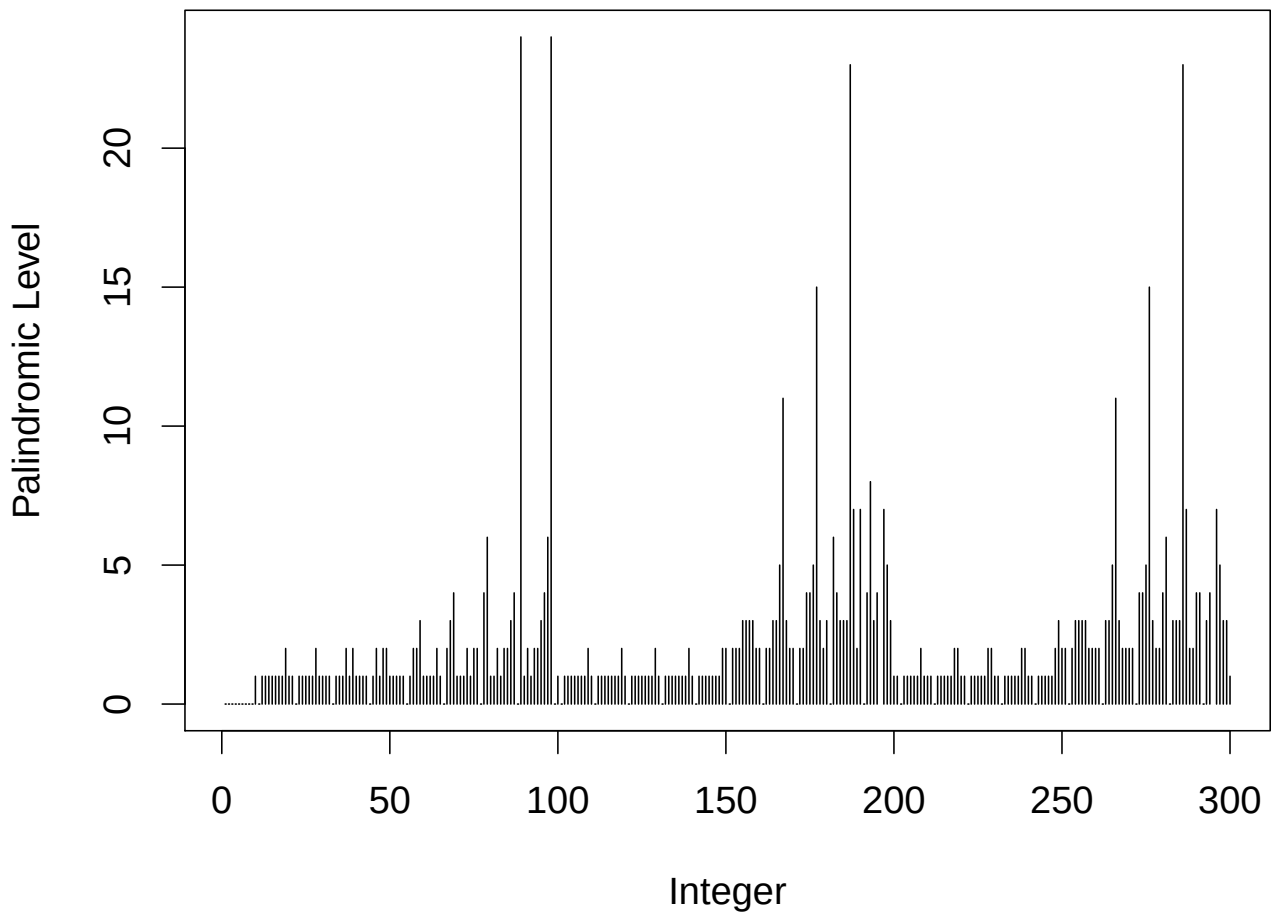
Lychrel Numbers

- Our program failed to compute the palindromic level of 196 because the values being computed during the computation became too big for R to handle accurately.
- In fact even with programs capable of working with numbers of up to 300,000,000 digits it has proved impossible to compute the palindromic level of 196.
- It has been conjectured that the palindromic level of 196 is infinite, but it is beyond the capabilities of present day mathematics to prove it.
- Numbers whose palindromic level is infinite are known as *Lychrel* numbers.

The Pattern of Palindromic Levels

- It is interesting to plot the pattern of palindromic levels for the first few hundred integers.

```
> plot(1:300, palindromic.level(1:300),
       type = "h",
       xlab = "Integer",
       ylab = "Palindromic Level")
```


Palindromic Level
0
5
10
15
20
0
50
100
150
200
250
300
Integer