Programação em R

Luís Torgo
FEP, Universidade do Porto
ltorgo@liacc.up.pt

13 de Dezembro de 2006

Nestes acetatos iremos abordar tópicos relaccionados com programação usando a linguagem R.

1. Interacção com o Utilizador

O R possui diversas funções para gerir a interacção com o utilizador, noameadmente: escrever no écran ou ler coisas introduzidas pelo utilizador.

Através de packages extra é ainda possível um interacção mais gráfica com o utilizador (p.ex. package rpanel).

A função `print()`, por exemplo, pode ser usada para escrever o conteúdo de qualquer objecto.

Note que, quando no prompt do R fazemos algo do género "`> x`" o que o R de facto faz é chamar a função `print()` com o objecto `x` como argumento!

Por vezes, quando os objectos são muito grandes (p. ex. grandes listas) é mais conveniente usar a função `str()` que dá uma perspectiva geral do objecto,

```
> str(list(nros=rnorm(100),dados=matrix(rnorm(10000),100,100)))
List of 2
 $ nros : num [1:100] -0.594  0.240  1.798 -1.139 -0.869 ...
 $ dados: num [1:100, 1:100] -1.472 -1.276  0.904  0.540 -1.247 ...
```

A função `cat()` também pode ser usada para escrever objectos ou constantes.

Funciona com um número qualquer de argumentos, e o que faz é transformar os seus argumentos em "strings", concatená-los, e só depois os escreve no écran,

```
> x <- 34
> cat("x tem o valor de ", x, "\t o que é estranho!\n")

x tem o valor de  34          o que é estranho!
```

Relativamente à leitura de dados introduzidos pelo utilizador também existem várias hipóteses em R.

A função `scan()`, por exemplo, permite ler dados de vário tipo,

```
> x <- scan(n=5)
1: 45 21.4 56 66.54 45.8787
Read 5 items
> x
[1] 45.0000 21.4000 56.0000 66.5400 45.8787
```

Ainda um outro exemplo,

```
> x <- scan()
1: 45 66 34.2
4: 456.7 7
6: 12.2
7:
Read 6 items
> x
[1]  45.0  66.0  34.2 456.7   7.0  12.2
```

E finalmente um outro exemplo com dados de outro tipo,

```
> scan(what=character())
1: 'erer' 'fdf'
3: '233' 44.5
5:
Read 4 items
[1] "erer" "fdf"  "233"  "44.5"
```

A função tem ainda muitas outras possíveis formas de ser usada que podem ser consultadas na respectiva ajuda.

2. Estruturas de Controlo

A linguagem R, como a qualquer linguagem de programação, possui várias instruções destinadas a alterar o curso sequencial normal de execução dos programas.

2.1. Instruções Condicionais

Permitem ao programador explicitar diferentes alternativas a serem executadas dependendo do valor de uma condição lógica.

A instrução `if`, por exemplo, permite explicitar uma condição e dois conjuntos de instruções alternativos que são executados dependendo do valor da condição. Vejamos um pequeno exemplo,

```
> (x <- rnorm(1))

[1] -1.122117


> if (x > 0) mx <- x else mx <- -x
> mx

[1] 1.122117
```

Em R tudo são funções ("coisas" que produzem resultados) logo poderiamos em alternativa fazer:

```
> mx <- if (x > 0) x else -x
> mx

[1] 1.122117
```

Existe uma versão "vectorial" desta instrução, que é a função `ifelse()`,

```
> (x <- rnorm(10))

 [1] -0.67854412  1.01647611 -0.79470257  0.16626482 -0.76535034  0.84501026
 [7]  0.17767309 -1.41100039  0.04641603  1.46653202


> mx <- ifelse(x > 0, x, -x)
> mx

 [1] 0.67854412 1.01647611 0.79470257 0.16626482 0.76535034 0.84501026
 [7] 0.17767309 1.41100039 0.04641603 1.46653202
```

Por vezes as alternativas são formadas por mais do que uma instrução.

O R permite definir um **bloco de instruções** como sendo um conjunto de instruções em linhas separadas e englobadas por chavetas.

```
> x <- rnorm(1)
> if (x > 0) {
+     cat("x é positivo.\n")
+     mx <- x
+ } else {
+     cat("x não é positivo!\n")
+     mx <- -x
+ }

x não é positivo!
```

Note-se que a claúsula `else` de qualquer `if` é opcional, podendo nós usar `if`'s sem alternativa para o caso de a condição ser falsa. Nesse caso, o R não fará nada como resultado da instrução `if`.

Podemos ainda usar várias instruções `if` aninhadas, como no exemplo seguinte:

```
> if (idade < 18) {
+     grupo <- 1
+ } else if (idade < 35) {
+     grupo <- 2
+ } else if (idade < 65) {
+     grupo <- 3
+ } else {
+     grupo <- 4
+ }
```

A instrução `switch` pode também ser usada para escolher uma de várias alternativas.

Ela consiste numa série de argumentos em que dependendo do valor do primeiro argumento, o resultado do `switch` é um dos outros argumentos.

Vejamos em primeiro lugar um exemplo em que o primeiro argumento é um número. Neste caso, o valor desse número determina qual o argumento do `switch` que é avaliado. Por exemplo,

```
> op <- 2
> vs <- rnorm(10)
> switch(op, mean(vs), median(vs))

[1] -0.3798445


> median(vs)

[1] -0.3798445
```

Na realidade, todos os argumentos a seguir ao primeiro são tomados como uma lista que pode ter nomes associados a cada componente.

No caso de não ter nomes, como no exemplo acima, é escolhida a componente da lista pelo número indicado no primeiro argumento do `switch`.

No caso de o número ser superior ao tamanho da lista, o resultado do `switch` é `NULL`.

Ao dar nomes às componentes da lista, passamos a poder usar valores textuais no primeiro argumento, como podemos ver no exemplo seguinte,

```
> semaforo <- "verde"
> switch(semaforo, verde = "continua", amarelo = "acelera", vermelho = "pára")

[1] "continua"
```

Por estes exemplos podemos ver que a instrução `switch` tem a forma genérica de

```
switch(valor,lista)
```

2.2. Instruções Iterativas

O R tem várias instruções iterativas ("ciclos") que nos permitem repetir blocos de instruções.

A instrução `while` tem a seguinte estrutura genérica,

```
while (<condição boolena>)
   <bloco de instruções>
```

A sua semântica pode ser descrita por: enquanto a condição booleana fôr verdadeira, repetir o bloco de instruções no "corpo" do ciclo. Vejamos um pequeno exemplo,

```
> x <- rnorm(1)
> while (x < -0.3) {
+  cat('x=',x,'\t')
+  x <- rnorm(1)
+ }
x= -0.8432914         x= -0.4693949
```

Note-se que as instruções no bloco dentro do ciclo podem nunca ser executadas, bastando para isso que a condição seja falsa da primeira vez que o R "chega" ao ciclo.

Em sintese podemos dizer que as instruções no bloco do ciclo `while` podem ser executadas zero ou mais vezes.

Atente-se que é um erro relativamente frequente escrever blocos de instruções que levam a que os ciclos nunca mais terminem.

Como regra geral podemos dizer que se deve escrever os ciclos de tal forma que exista sempre possibilidade de que a(s) variável(eis) que controla(m) a execução dos mesmos (no nosso exemplo a variável **x**), possa sempre ver o seu valor alterado nas instruções que formam o bloco contido no ciclo, de forma a que a condição possa vir a ser falsa, ou seja de forma a que o ciclo possa vir a terminar.

A instrução `repeat` permite mandar o R executar um bloco de instruções uma ou mais vezes. A sua forma genérica é,

```
repeat
   <bloco de instruções>
```

Repare que uma vez que não existe qualquer condição lógica a governar a execução repetida do bloco de instruções, como no caso do ciclo `while`, o R socorre-se de outras instruções que permitem parar a execução de um processo iterativo.

A instrução `break`, se executada dentro do ciclo, faz o R terminar a repetição da execução do bloco de instruções.

Vejamos um pequeno exemplo que lê frases introduzidas pelo utilizador até este introduzir uma frase vazia,

```
> texto <- c()
> repeat {
+   cat('Introduza uma frase ? (frase vazia termina) ')
+   fr <- readLines(n=1)
+   if (fr == '') break else texto <- c(texto,fr)
+ }
```

O próximo exemplo ilustra o uso de outra instrução que pode ser usada para controlar o que é feito num ciclo, a instrução `next`,

```
> pos <- c()
> repeat {
+   cat('Introduza um nro positivo ? (zero termina) ')
+   nro <- scan(n=1)
+   if (nro < 0) next
+   if (nro == 0) break
+   pos <- c(pos,nro)
+ }
```

O R tem ainda a instrução `for` que permite controlar o número de vezes que um ciclo é executado através de uma variável de controlo que vai tomar uma série de valores pré-definidos em cada iteração do ciclo. A sua sintaxe genérica é,

```
for(<var> in <conjunto>)
   <bloco de instruções>
```

Vejamos um pequeno exemplo de utilização destes ciclos,

```
> x <- rnorm(10)
> k <- 0
> for(v in x) {
+   if (v > 0) y <- v else y <- 0
+   k <- k + y
+ }
```

Claro que

```
> k <- sum(x[x > 0])
```

era mais fácil ;-) , além de ser muito mais eficiente em termos de tempo de execução, embora para um vector com este tamanho a diferença seja irrelevante. O mesmo já não se poderá dizer para vectores maiores como se pode constatar nesta pequena experiência,

```
> x <- rnorm(100000)
> t <- Sys.time()
> k <- 0
> for(v in x) {
+   if (v > 0) y <- v else y <- 0
+   k <- k + y
+ }
> Sys.time() - t
Time difference of 0.351563 secs
> t <- Sys.time()
> k <- sum(x[x> 0])
> Sys.time() - t
Time difference of 0.01090908 secs
```

2.3. Evitando ciclos

O R possui, como a maioria das linguagens, objectos que permitem armazenar conjuntos de valores (p. ex. vectores, matrizes, etc.).

Existem inúmeros problemas onde para os resolver, precisamos de percorrer todos os elementos desses objectos.

Na maioria das linguagens isso faz-se com o recurso a ciclos.

No R, temos felizmentes várias alternativas bastante mais práticas e sintéticas, para além de muito mais eficientes.

Uma das formas mais comuns de evitar os ciclos consiste em tirar partido do facto de muitas funções do R serem vectorizáveis.

Suponhamos que pretendemos somar todos os elementos de uma matriz **m**.

Solução "tradicional":

```
> s <- 0
> for(c in 1:ncol(m))
+   for(l in 1:nrow(m))
+     s <- s + m[l,c]
```

Solução "à R":

```
sum(m)
```

O R tem ainda outras funções que podem ser explicitamente usadas para vectorizar operações.

Suponhamos que pretendemos saber o valor mínimo de cada coluna de uma matriz. A nossa "conhecida" função `apply()` pode-nos dar uma grande ajuda,

```
> m <- matrix(rnorm(100), 10, 10)
> apply(m, 2, min)

 [1] -1.5412378 -1.4971939 -2.6028208 -1.6066139 -0.9027720 -1.2944808
 [7] -1.9815078 -1.3819544 -1.0988993 -0.8120515
```

Para sentir "na pele" as vantagens da linguagem R, resolva o mesmo problema destas duas formas:

1. Assumindo que não existe a função `apply()`
2. Assumindo que não existem nem a função `apply()` nem a função `min()`.

A função `tapply()` permite executar operações semelhantes mas em sub-grupos dos dados, determinados por valores de factores.

Vejamos um exemplo usando um conjunto de dados que vem com o R para efeitos ilustrativos,

```
> data(warpbreaks)
> head(warpbreaks)

  breaks wool tension
1     26    A       L
2     30    A       L
3     54    A       L
4     25    A       L
5     70    A       L
6     52    A       L


> tapply(warpbreaks$breaks, warpbreaks[, -1], sum)

    tension
wool   L   M   H
   A 401 216 221
   B 254 259 169
```

Uma outra função interessante é a função `sapply()`, que permite aplicar uma função a todos os elementos de um vector ou lista. Vejamos dois exemplos ilustrativos,

```
> x <- sample(10)
> sapply(x, log)

 [1] 2.3025851 0.6931472 0.0000000 1.3862944 1.9459101 1.7917595 1.0986123
 [8] 1.6094379 2.1972246 2.0794415


> l <- list(f1 = sample(20), f2 = c(2.3, 1.3, 4.5, 2.4), f3 = rnorm(100))
> sapply(l, quantile)

        f1    f2         f3
0%    1.00 1.300 -2.0498868
25%   5.75 2.050 -0.6350390
50%  10.50 2.350 -0.0304650
75%  15.25 2.925  0.6731329
100% 20.00 4.500  2.4331582
```

3. Funções

Na linguagem R as funções são também objectos e podem ser manipuladas de forma semelhante aos outros objectos que estudamos até agora.

Uma função tem três características principais: uma lista de argumentos, o corpo da função e o ambiente onde ela é definida.

A lista de argumentos é uma lista de símbolos (os argumentos) separada por vírgulas, que podem ter valores por defeito.

O corpo de uma função é formado por um conjunto de instruções da linguagem R, sendo que normalmente é um bloco de instruções.

Quanto ao ambiente da função, trata-se do ambiente activo quando ela foi criada e isso determina que tipo de objectos são visíveis pela função conforme vermos mais à frente.

3.1. Criar funções

A criação de uma função consiste na atribuição do conteúdo de uma função a um nome, como qualquer outro objecto do R.

Esse conteúdo é a lista dos seus argumentos e as instrução que formam o corpo da função.

Vejamos um exemplo simples de uma função que recebe como argumento uma temperatura em graus Celsius e a converte para graus Fahrenheit,

```
> cel2far <- function(cel) {
+     res <- 9/5 * cel + 32
+     res
+ }
> cel2far(27.4)

[1] 81.32


> cel2far(c(0, -34.2, 35.6, 43.2))

[1]  32.00 -29.56  96.08 109.76
```

O corpo desta função está desnecessariamente complexo, podendo nós em alternativa definir a função como,

```
> cel2far <- function(cel) 9/5 * cel + 32
```

Em resumo, a criação de uma função é uma atribuição com a forma genérica,

```
<nome da função> <- function(<lista de argumentos>) <corpo da função>
```

O valor retornado como resultado da função **é o resultado da última computação** efectuada na função.

Existe uma função específica, a função `return()`, que permite retornar um determinado valor numa função. Isso quer dizer que essa função se executada no corpo de uma função vai fazer com que a execução da mesma termine aí imediatamente. Vejamos um exemplo,

```
calculo <- function(x) {
  if (x<0) return(NULL)
  y <- 34*sqrt(x)/5.2
  y
}
```

3.2. Ambientes e "scope" de variáveis

Quando começamos a nossa interacção com o R, estamos a fazê-lo num ambiente chamado o *workspace*, ou o ambiente de topo.

Sempre que fazemos `ls()` por exemplo, para saber os objectos que temos na memória, o que obtemos é a lista de objectos neste ambiente de topo.

Existem situações que levam à criação de novos ambientes, levando a uma hierarquia de ambientes com a forma de uma árvore invertida, aninhados uns nos outros, até ao ambiente de topo.

A importância destas noções reside nas regras de "scope" ou visibilidade dos objectos em R que dependem do ambiente em que eles são definidos.

Quando criamos uma função ela, como objecto que é, fica associada ao ambiente onde foi criada.

Quando chamamos essa função o R vai executá-la num novo ambiente "por baixo" do ambiente onde ela é chamada.

Quer isto dizer que o ambiente onde são executadas as instruções que formam o corpo de uma função não é o mesmo ambiente onde a função é chamada.

Isto tem impacto nos objectos que são visíveis em cada um destes dois ambientes.

Vejamos um exemplo das implicações e importância desta noção de ambientes. Consideremos o seguinte código, que inclui a definição de uma função, e que assumimos ser introduzido no ambiente de topo,

```
> x <- 2
> z <- 56.3
> f <- function(x) {
+   a <- 34
+   y <- x / 4 * a * z
+   y
+ }
> f(21)
[1] 10049.55
> a
Error: Object "a" not found
```

A regra fundamental na linguagem R relativamente à visibilidade do objectos, é conhecida por *lexical scoping*, e diz-nos que:

> Quando um objecto é necessário numa avaliação, ele é procurado pelo R no ambiente em que foi pedido. Caso seja encontrado um objecto com esse nome nesse ambiente, ele é usado. Caso tal não aconteça o R procura-o no ambiente acima, e assim sucessivamente até ao ambiente de topo. Se neste processo for encontrado um objecto com esse nome ele é usado. De contrário o R dá um erro dizendo que tal objecto não existe.

3.3. Argumentos de funções

Quando criamos uma função indicamos normalmente a sua lista de argumentos.

Nessa altura a linguagem R permite-nos também explicitar, caso queiramos, um valor por defeito para cada um desses argumentos.

O uso desta facilidade, vai permitir ao utilizador das nossas funções, evitar incluir um valor para esses argumentos caso pretenda usar o valor por defeito.

Isto é particularmente útil em funções com muitos argumentos, alguns dos quais só usados em situações muito particulares.

```
> mean(c(21, 45.3, 342.4, 54.3, 65.3, 1000.2))

[1] 254.75
```

A maioria das vezes usamos a função `mean()` deste modo. No entanto, se consultarmos a ajuda desta função iremos observar que ela tem outros dois argumentos, *trim* e *na.rm*, cada um deles com um valor por defeito, 0 e *FALSE*, respectivamente.

Se não pretendessemos usar estes valores, teriamos que o explicitar na chamada à função, como no exemplo a seguir,

```
> mean(c(21, 45.3, 342.4, 54.3, 65.3, 1000.2), trim = 0.2)

[1] 126.825
```

Use a ajuda da função para ver se entende o que foi calculado com este valor do parâmetro *trim*.

Ao criarmos uma função podemos indicar valores por defeito para alguns dos seus parâmetros, bastanto para isso usar o sinal igual seguido do valor à frente do nome do argumento, conforme ilustrado neste pequeno exemplo,

```
> valor.central <- function(x, estat = "mean") {
+     if (estat == "mean")
+         return(mean(x))
+     else if (estat == "median")
+         return(median(x))
+     else return(NULL)
+ }
> x <- rnorm(10)
> valor.central(x)

[1] 0.3707976


> valor.central(x, estat = "median")

[1] 0.4078363
```

Uma outra facilidade bastante conveniente da linguagem R é a possibilidade de criar funções com número variável de parâmetros.

Isso é conseguido por um parâmetro especial que é indicado por "...".

Este parâmetro especial é de facto uma lista que pode agregar um número qualquer de parâmetros usados na chamada da função.

Uma utilização frequente desta facilidade é na criação de funções que chamam outras funções e ao fazê-lo pretendem passar-lhes parâmetros que só a elas lhes interessam.

Vejamos um exemplo,

```
> valor.central <- function(x, estat = "mean", ...) {
+     if (estat == "mean")
+         return(mean(x, ...))
+     else if (estat == "median")
+         return(median(x, ...))
+     else return(NULL)
+ }
> x <- rnorm(10)
> valor.central(x)

[1] -0.02620797


> valor.central(x, trim = 0.2)

[1] -0.1278131


> valor.central(x, estat = "median", na.rm = T)

[1] -0.004502386
```

Para além das questões ligadas aos **parâmetros formais** (os parâmetros usados na definição da função), que descrevemos até agora, existem ainda questões ligadas aos **parâmetros actuais** (os usados nas chamadas às funções), que agora detalhamos.

Podemos indicar os parâmetros actuais de 2 formas: através de posição, ou por nome.

Uma chamada por posição ocorre quando pretendemos que o valor indicado na posição x da lista de argumentos da função seja associado ao argumento formal na mesma posição.

Por exemplo, ao executar `seq(10,23)` o R vai assumir que o valor 10 é para ser atribuído ao primeiro argumento formal da função **seq** (o argumento *from* no caso, conforme pode confirma na ajuda da função), e o valor 23 no segundo argumento formal (o argumento *to*).

Já a chamada `seq(10,length=23)` tem um resultado completamente diferente, uma vez que o segundo valor é indicado por nome e não por posição como no exemplo anterior. Isto quer dizer que o número 23 não é atribuído ao argumento na segunda posição, como anteriormente, mas sim ao argumento formal com o nome *length*.

3.4. Lazy evaluation

Conforme já foi mencionado , sempre que é chamada uma função o R cria um novo ambiente por baixo do ambiente onde a chamada foi efectuada.

Nessa altura as expressões dos argumentos actuais são verificadas sintaticamente, **mas não são avaliadas**.

A este procedimento chama-se *lazy evaluation*.

Os argumentos actuais só são avaliados quando são necessários pela primeira vez no corpo da função. Esta regra pode ter por vezes implicações inexperadas conforme ilustrado nestes pequenos exemplos:

```
> f1 <- function(a1, a2 = sqrt(a1)) {
+     a1 <- a1^2
+     a2
+ }
> f1(4)

[1] 4


> f2 <- function(a1, a2 = sqrt(a1)) {
+     z <- a2/a1
+     a1 <- a1^2
+     a2
+ }
> f2(4)

[1] 2
```

3.5. Algumas funções úteis

Algumas estatísticas básicas

`sum(x)`	Soma dos elementos do vector `x`.
`max(x)`	Máximo dos elementos de `x`.
`min(x)`	Mínimo dos elementos de `x`.
`which.max(x)`	O índice do maior valor em `x`.
`which.min(x)`	O índice do menor valor em `x`.
`range(x)`	O *range* de valores em `x` (produz o mesmo resultado que `c(min(x),max(x))`).
`length(x)`	O número de elementos em `x`.
`mean(x)`	A média dos valores em `x`.
`median(x)`	A mediana dos valores em `x`.
`sd(x)`	O desvio padrão dos elementos em `x`.
`var(x)`	A variância dos elementos em `x`.
`quantile(x)`	Os quartis de `x`.
`scale(x)`	Normaliza os elementos em `x`, *i.e.* subtrai a média e divide pelo desvio-padrão. Faz esta operação por cada coluna de `x` (i.e. `x` tem que ser uma matriz. Também funciona com *data frames* (só as colunas numéricas, obviamente).

Algumas operações vectoriais e matemáticas

`sort(x)`	Elementos de `x` ordenados.
`rev(x)`	Inverte a ordem dos elementos em `x`.
`rank(x)`	Faz o *ranking* dos elementos de `x`.
`log(x,base)`	Calcula o logaritmo na base "`base`" de todos os elementos de `x`.
`exp(x)`	Calcula o exponencial dos elementos de `x`.
`sqrt(x)`	Raiz quadrada dos elementos de `x`.
`abs(x)`	Valor absoluto dos elementos de `x`.
`round(x,n)`	Arredonda os valores em `x` para `n` casas decimais.
`cumsum(x)`	Obtém um vector em que o elemento *i* é a soma dos elementos $x[1]$ até $x[i]$.
`cumprod(x)`	O mesmo para o produto.
`match(x,s)`	Obtém um vector com o mesmo tamanho de `x`, contendo os elementos de `x` que pertencem a `s`. Os elementos que não pertencem a `s` aparecem no resultado assinalados com o valor NA.
`union(x,y)`	Obtém um vector com a união dos vectores `x` e `y`.
`intersect(x,y)`	Obtém um vector com a intersecção dos vectores `x` e `y`.
`setdiff(x,y)`	Obtém um vector resultante de retirar os elementos de `y` do vector `x`.
`is.element(x,y)`	Retorna o valor TRUE se `x` está contido no vector `y`.
`choose(n,k)`	Calcula o número de combinações `k`, `n` a `n`.

Álgebra Matricial

`diag(x,nrow,ncol)`	Constrói uma matriz diagonal com `nrow` linhas e `ncol` colunas, usando o número `x`. Também pode ser usada para extrair ou substituir os elementos na diagonal de uma matriz (ver exemplos fazendo "`? diag`").
`t(x)`	A matriz transposta de `x`.
`nrow(x)`	Número de linhas de `x`.
`ncol(x)`	Número de colunas de `x`.
`A %*% B`	Multiplicação matricial de `A` por `B`.
`solve(A,b)`	Resolve o sistema de equações lineares $Ax = b$. Com um único argumento (uma matriz) (*e.g.* `solve(A)`) calcula a sua inversa.
`qr(x)`	Decomposição QR da matriz `x`.
`svd(x)`	Decomposição SVD (*singular value decomposition*) da matriz `x`.
`eigen(x)`	Valores e vectores próprios da matriz quadrada `x`.
`det(x)`	O determinante da matriz quadrada `x`.

4. Objectos, Classes e Métodos

O R é uma linguagem de programação orientada aos objectos. Tudo em R são objectos de maior ou menor complexidade.

Quando alguém cria uma classe de objectos (por exemplo uma matriz), também cria uma série de métodos específicos a esta classe, para realizar as funções mais comuns com esses objectos.

Muitas das funções do R "escondem" métodos específicos para diferentes classes de objectos.

Por exemplo, a função `summary()` produz um sumário do conteúdo do objecto que lhe é fornecido como argumento.

O significado deste sumário, bem como a forma como ele é obtido, está em grande parte dependente do tipo de objecto de que queremos o sumário. No entanto o utilizador não precisa de saber isso. Vejamos um exemplo concreto deste comportamento,

```
> x <- rnorm(10)
> summary(x)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-1.09800 -0.68910  0.03718 -0.07999  0.55590  0.84780
```

```
> summary(matrix(x, 5, 2))
```

```
       X1                X2
 Min.   :-0.7458   Min.   :-1.0980
 1st Qu.:-0.5188   1st Qu.:-1.0838
 Median : 0.5196   Median :-0.3818
 Mean   : 0.1342   Mean   :-0.2941
 3rd Qu.: 0.5680   3rd Qu.: 0.4562
 Max.   : 0.8478   Max.   : 0.6368
```

Duas chamadas à mesma função (na realidade só aparentemente a mesma), produzem resultados diferentes dependendo da classe do objecto dado como argumento, embora conceptualmente ambos sejam sumários desse argumento.

De facto, a função `summary()` é uma função genérica, que basicamente o que faz é ver a classe do objecto que lhe é passado como argumento, e depois "despacha" a tarefa de produzir o sumário para funções especializadas nessa classe de objectos.

```
> summary

function (object, ...)
UseMethod("summary")
<environment: namespace:base>


> methods(summary)

 [1] summary.aov            summary.aovlist        summary.connection
 [4] summary.data.frame     summary.Date           summary.default
 [7] summary.ecdf*          summary.factor         summary.glm
[10] summary.infl           summary.lm             summary.loess*
[13] summary.manova         summary.matrix         summary.mlm
[16] summary.nls*           summary.packageStatus* summary.pessoa
[19] summary.POSIXct        summary.POSIXlt        summary.ppr*
[22] summary.prcomp*        summary.princomp*      summary.stepfun
[25] summary.stl*           summary.table          summary.tukeysmooth*

   Non-visible functions are asterisked
```

A grande vantagem para o utilizador está em só ter que conhecer a função `summary()`!

A função `class()` pode ser usada para obter a classe (ou classes) de um objecto:

```
> class(x)

[1] "numeric"


> class(matrix(x, 5, 2))

[1] "matrix"


> class(as.POSIXct("2006-9-23"))

[1] "POSIXt"  "POSIXct"
```

Vejamos agora como criar uma nova classe. Suponhamos que queriamos criar uma classe de objectos para armazenar dados de pessoas:

```
> pessoa <- function(n, p, d, s) {
+     p <- list(nome = n, pais = p, nascimento = d, sexo = s)
+     class(p) <- "pessoa"
+     p
+ }
> chefe <- pessoa("Ana", list(pai = "Carlos", mae = "Joana"), as.Date("1970-03-3"),
+     "f")
> class(chefe)

[1] "pessoa"
```

Poderiamos agora defenir uma função específica para sumariar um objecto deste tipo:

```
> summary.pessoa <- function(p) cat("O meu nome é ", p$nome, " e sou ",
+     ifelse(p$sexo == "f", "filha", "filho"), " de ", p$pais$pai,
+     " e ", p$pais$mae, " tendo nascido a ", format(p$nascimento,
+         "%d %b %Y"), "\n")
> summary(chefe)

O meu nome é  Ana  e sou  filha  de  Carlos  e  Joana  tendo nascido a  03 Mar 1970
```

5. Depuração de Programas

A depuração de programas em qualquer linguagem de programação tem como objectivo principal encontrar a fonte de erros ou comportamentos imprevistos nos programas que desenvolvemos.

O R possui várias funções que podem ser utilizadas para ajudar nesta tarefa.

Uma delas é a função `traceback()` que permite ao utilizador obter a sequência de funções que foram chamadas até o erro ocorrer.

Se tal não fôr suficiente para identificar a fonte do erro, ou então se localizamos a sua possível origem mas não entendemos o porquê do erro, poderemos também socorrer-mo-nos da função `browser()`.

Quando o R encontra esta função, executa todas as instruções seguintes passo a passo e unicamente sob a nossa ordem.

Em cada uma destas paragens o R vai mostrar um *prompt* especial onde o utilizador poderá fazer quase tudo o que pode fazer no *prompt* normal do R (por exemplo ver o conteúdo de objectos da função a ser depurada), mas onde existem ainda certos comandos especiais ligados à função `browser()`.

Vejamos um pequeno exemplo deste tipo de depuração. Suponhamos que temos a seguinte função, que aparenta estar sem problemas,

```
> er <- function(n, x2) {
+     s <- 0
+     for (i in 1:n) {
+         s <- s + sqrt(x2)
+         x2 <- x2 - 1
+     }
+     s
+ }
> er(2, 4)

[1] 3.732051
```

Suponhamos agora que fazemos a seguinte chamada que inesperadamente produz um aviso sobre um erro numérico (**NaN** significa *Not A Number*) na execução da função,

```
> er(3,1)
[1] NaN
Warning message:
NaNs produced in: sqrt(x2)
```

Vamos incluir uma chamada à função `browser()` no corpo da função antes do local que suspeitamos ser a fonte do aviso.

```
> er <- function(n,x2) {
+   s <- 0
+   browser()
+   for(i in 1:n) {
+     s <- s + sqrt(x2)
+     x2 <- x2-1
+   }
+   s
+ }
> er(3,1)
Called from: er(3, 1)
Browse[1]> x2
[1] 1
Browse[1]> print(n)
[1] 3
Browse[1]> n
debug: for (i in 1:n) {
    s <- s + sqrt(x2)
    x2 <- x2 - 1
}
Browse[1]> n
debug: i
Browse[1]> i
NULL
Browse[1]> n
```

```
debug: s <- s + sqrt(x2)
Browse[1]> i
[1] 1
Browse[1]> s
[1] 0
Browse[1]> sqrt(x2)
[1] 1
Browse[1]>
debug: x2 <- x2 - 1
Browse[1]> s
[1] 1
Browse[1]>
debug: i
Browse[1]> x2
[1] 0
Browse[1]>
debug: s <- s + sqrt(x2)
Browse[1]> i
[1] 2
Browse[1]> sqrt(x2)
[1] 0
Browse[1]>
debug: x2 <- x2 - 1
Browse[1]>
debug: i
Browse[1]>
debug: s <- s + sqrt(x2)
Browse[1]> sqrt(x2)
[1] NaN
Warning message:
NaNs produced in: sqrt(x2)
Browse[1]> x2
[1] -1
Browse[1]> Q
>
```

Uma alternativa à função `browser()` que obriga a laterar o nosso código, é a função `debug()`.

No nosso exemplo poderiamos ter feito `er(3,1)`. A intereacção que se segue é em tudo idêntica à que vimos acima.

Note-se que até fazermos `undebug(er)`, sempre que voltamos a chamar a função `er()` o R entra em modo de *debugging*.

A vantagem da função `browser()` em relação a esta alternativa é que podemos iniciar o *debugging* no ponto da função que queremos, enquanto que com esta alternativa ele começa logo deste a primeira instrução da função.