Eines Informàtiques per a l'Estadística: R

Gregori Guasp, Albert Ruiz, Aureli Alabert, Sundus Zafar

Grau d'Estadística Aplicada
Universitat Autònoma de Barcelona

11 d'octubre de 2021

1 Instal·lació del R

El **R** és un programari per a la manipulació, anàlisi i representació gràfica de dades. És gratuït, *open source*, i extensible a partir de paquets (*packages*) que es poden instal·lar addicionalment al sistema bàsic en el moment que es necessitin, i als quals tothom pot contribuir quan es tenen els coneixements suficients. Parlarem més endavant de l'estructura del R, les seves extensions i paquets.

Per instal·lar-lo, aneu a

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

Seguiu l'enllaç de la segona línia `download R`, que us porta a una llista de servidors rèplica (*mirrors*). Escolliu un que estigui geogràficament proper (per exemple, `https://cran.rediris.es/`, o bé `https://stat.ethz.ch/CRAN/`), i us portarà a una pàgina on veureu tres enllaços, per Linux, per MacOS, i per Windows. Escolliu el corresponent al vostre sistema operatiu.

1.1 Linux

Quan seguiu l'enllaç per a Linux haureu d'escollir la distribució de Linux que teniu. Seguiu les instruccions, que són molt específiques de cada distribució. En cas de dubte, pregunteu. En principi, és suficient instal·lar el paquet `r-base` que està en totes les distribucions, mitjançant:

```
sudo apt-get install r-base
```

1.2 MacOS

Si teniu un ordinador Apple, us heu de descarregar el fitxer `.pkg` adequat d'aquesta pàgina.

- Si el vostre processador és Intel i el sistema és 10.13 (High Sierra) o posterior, baixeu el fitxer `R-4.1.1.pkg` (el número concret de versió pot haver canviat).
- Si és un dels nous processadors Apple M1, aleshores baixeu el `R-4.1.1-arm64.pkg`.
- Si és Intel, però teniu un sistema anterior al 10.13, busqueu més avall a la pàgina la versió més recent que pugui funcionar en el vostre sistema.

Si dubteu quina versió de sistema o processador teniu, ho podeu veure en el menú `Apple > About this Mac`.

Obriu el `.pkg` que heu baixat i seguiu les instruccions de l'instal·lador. Al acabar, tindreu un fitxer `R.app` en el directori `Applications` i en el `Launchpad`.

1.3 Windows

Seguiu l'enllaç `base` i després clickeu `Download R 4.1.1 for Windows`, que us baixarà el fitxer `R-4.1.1-win.exe`. Executeu aquest fitxer. (El número concret de versió pot haver canviat.)

Us demanarà l'idioma d'instal·lació. Escolliu-lo i després accepteu totes les opcions que va preguntant tal qual s'ofereixen.

2 Instal.lació del RStudio

El **RStudio** no és imprescindible per treballar i programar en **R**, però és convenient tenir-lo, i en alguns moments el farem servir.

Independentment del vostre sistema operatiu, aneu a `RStudio.com` i seguiu els enllaços `Products`, `RStudio`, `RStudio Desktop`, `Download RStudio Desktop`, i premeu el botó `Download`. Segurament detectarà el vostre sistema automàticament, i si no, teniu totes les versions en aquesta pàgina.

Específicament per a MacOS: Observeu que la versió més recent requreix macOS 10.14 (Mojave) o posterior, i que versions de **RStudio** anteriors a la 1.4.1717 és possible que no funcionin en els processadors Apple M1. Baixeu el `RStudio-....dmg` i obriu-lo. Moveu el fitxer `RStudio.app` al directori `Applications`.

3 Arrencada

Algunes característiques visuals poden ser diferents en els diferents sistemes operatius. Però el funcionament és pràcticament idèntic en tots tres. Si no veieu les coses exactament com expliquem aquí, la diferència serà mínima i fàcilment interpretable.

Al engegar el **R** veiem una finestra mare, amb títol **RGui** (que vol dir *R Graphic User Interface*), dins la qual hi ha una finestra filla, la **R Console**[1]. A la consola veiem un missatge de benvinguda i un *prompt* (el símbol `>`), que ens indica que el sistema està preparat per rebre instruccions. Les instruccions s'acaben amb la tecla Retorn (Enter), i s'executen immediatament. Per exemple, proveu d'escriure `2+3`. La resposta normalment apareix directament sota la instrucció. Però hi ha instruccions que fan que s'obrin noves finestres dins de **RGui**, per exemple per mostrar un gràfic. El **R** és una eina molt potent per fer gràfics a partir de dades numèriques.

La finestra **RGui** té també un sistema de menús i una barra d'eines. La majoria de les accions dels menús i els botons tenen a veure amb la interacció amb el sistema operatiu, com ara obrir i tancar fitxers, i no amb l'anàlisi de dades en sí. Destaquem ara només el `Help > Manuals (in PDF)`, on hi ha diversos manuals. En particular, *An Introduction to R* us pot servir de manual de referència.[2]

Aconsellem que useu la interfície en anglès, que és l'idioma que suposarem en aquestes pràctiques, i en el que trobareu més ajuda a internet quan us calgui. Podeu forçar l'anglès de la manera següent:

Per a Linux i Windows, Obriu el menú `Edit > GUI preferences ...` ; en el quadre de diàleg que s'obre, en el camp `Language for menus and messages`, escriviu **en**; premeu

[1] En Linux i MacOS, possiblement només veureu la finestra **R Console**, en la qual apareix la barra de menús, o una barra d'eines. No té cap importància la diferència. Vegeu després el paràgraf 7 § per més explicació.

[2] En MacOS, no hi ha aquesta entrada de menú, però aquest manual el teniu en algun lloc, en el fitxer `R-intro.pdf`.

el botó `Save...` i confirmeu que voleu guardar un fitxer anomenat `Rconsole`; tanqueu i torneu a obrir el **RGui**. D'ara endavant tindreu sempre l'interfície en anglès.

Per a MacOS, heu d'anar a `Apple > System Preferences > Language & Region > Apps` i canviar aquí l'idioma.

4 Primeres passes

1 § Com heu vist abans, el **R** sap fer alguns càlculs:

```
3+2
[1] 5
```

També sap fer càlculs més complicats, com ara e^{-2}:

```
exp(-2)
[1] 0.1353353
```

Totes les funcions matemàtiques habituals són accessibles escrivint la instrucció oportuna. Per exemple, sinus, cosinus, tangent, logaritme neperià i logaritme en base 10 es calculen d'aquesta manera (proveu-ho amb aquests i altres números):

```
sin(0)
cos(0)
tan(0)
log(100)
log10(100)
```

Observacions:

- En matemàtiques log fa referència al logaritme neperià, sempre! El logaritme en base 10 es denota $\log_{10}$, i en qualsevol altre base b, s'escriu $\log_b$. És possible que ho hàgiu estudiat abans d'una altra manera; caldrà que us acostumeu al canvi. El logaritme neperià també s'escriu de vegades ln, o Ln.
- Els arguments de les funcions trigonomètriques venen donats en radians, no en graus. Aquest també és el conveni en general en matemàtiques, si no es diu el contrari. Així, podeu comprovar que `sin(90)` no dóna 1, com seria el cas si es tractés de graus.

2 § Fins aquí una calculadora de butxaca, un *smartphone* o una *tablet* ho poden fer igual de bé. Compliquem-ho una mica utilitzant *variables*, és a dir, noms que poden contenir valors numèrics concrets:

```
x <- 3
x + 2
[1] 5
x * (x+2)
[1] 15
```

La primera d'aquestes instruccions és una *assignació*. A partir d'aquest moment la variable `x` tindrà el valor `3`. La "fletxa" s'ha d'escriure amb els dos caràcters `< -` seguits; els espais blancs abans i després de la fletxa són opcionals, però queda més llegible si es posen. Acostumeu-vos que, encara que l'ordinador ho entengui igual, les instruccions també les hem de llegir els humans, i es bo que s'entenguin fàcilment.

La segona instrucció suma 2 unitats al valor contingut a la variable. El **R** ens contesta amb el resultat. Anàlogament amb la tercera instrucció.

Observeu que les *assignacions* no produeixen cap resposta. Per saber el valor que té una variable, només cal escriure una instrucció amb el seu nom. Aquestes dues instruccions produeixen el mateix resultat:

```
x
print(x)
```

Òbviament, la primera és més còmode, però a partir d'un cert moment veureu que cal usar la segona per a què les coses funcionin.

En la majoria de llenguatges de programació i sistemes del tipus del **R**, les assignacions es fan amb el símbol d'igualtat `=` en lloc de la fletxa. El **R** també admet aquest símbol en la major part de les situacions, però no sempre. Per tant, no és aconsellable usar-lo i la majoria dels experts ho considerem de mal estil.

També és admissible la "fletxa a l'inrevés". Per exemple `7 -> y` assigna el valor 7 a la variable `y`. Però queda difícil de llegir i per això tampoc no és aconsellable.

3 § A més d'assignar un valor numèric directe a una variable, podem assignar-li el resultat d'una expressió involucrant altres variables definides abans:

```
x <- 1.4
y <- x^2 + 5
z <- sqrt(x+y)
```

Observem que:

- S'utilitza un "punt decimal", no pas una coma.
- El símbol `^` serveix per *exponenciar*: elevar una base a un potència. En aquest cas, estem elevant `x` al quadrat.
- L'arrel quadrada s'obté mitjançant `sqrt()`.

Feu que el **R** escrigui el valor d'aquestes tres variables ara. Observeu que la variable `x`, que més amunt tenia el valor 3, ara té el nou valor 1.4; és a dir, una nova assignació anul·la l'assignació anterior. I veiem que aquest valor no canvia quan s'utilitza en una expressió per assignar valor a una altra variable.

Veiem que passa quan usem una variable a la qual no s'ha assignat abans un valor:

```
x <- h
```

Heu d'obtenir com a resultat el missatge: `Error: object 'h' not found`. Si imprimiu després el valor de `x` veureu que no ha canviat. La instrucció és invàlida i no fa canviar res.

Una de les coses més difícils en **R** i en altres programaris similars és entendre els missatges d'error. L'anterior missatge és força clar, però de vegades no ho és tant. Prepareu-vos per un cert grau de frustració quan no entengueu perquè el **R** no us admet una instrucció. Ens passa a tots i cal paciència. Quan tot falla, internet pot ajudar molt: proveu de posar a Google o qualsevol altre buscador `R object not found` i veureu la gran quantitat de documents que mencionen aquesta frase fent referència al **R**.

4 § Una variable no té perquè ser una lletra. Una variable pot ser qualsevol tira de caràcters amb les condicions següents:

- És una combinació de lletres, dígits, punts (`.`) i blancs subratllats (`_`).[3]
- Comença amb una lletra o un punt. Si comença amb un punt, el següent caràcter no pot ser un dígit.

Per tant, són vàlids noms com ara `B42`, `.a.b`, `O__O`. I naturalment, és molt aconsellable que els noms de les variables tinguin un sentit en el context en que s'utilitzen, per poder recordar què contenen: És millor `temperatura` i `Velocitat` que `t`, i `V`, per exemple.

Cal tenir en compte que majúscules i minúscules són lletres diferents. Comproveu-ho. Vegeu també quin és el missatge d'error quan intentem assignar un nom de variable invàlid.

Hi ha unes quantes *paraules reservades* del **R** que no és permès usar com a variables. Aquestes paraules són, bàsicament,

```
if else repeat while function for in next break
TRUE FALSE NULL Inf NaN NA NA_integer_ NA_real_ NA_complex_ NA_character_
```

Els que parlem idiomes llatins tenim una tendència natural a usar accents en els noms de variable. Segurament el **R** us admetrà noms com ara `desviació` o `alçada`. Però, per raons llargues d'explicar, és fortament aconsellable restringir-nos a usar els caràcters de l'alfabet anglès. El consell val també per a noms de fitxers, on tampoc haurien d'haver-hi mai espais blancs. A aquests efectes, `ñ` i `ç` també són lletres accentuades.

5 § Hi ha alguns noms de variable que ja tenen un valor quan arrenquem el **R**, tot i que es poden canviar si es vol. Per exemple, `pi`. Mireu quin valor té, i calculeu el sinus i el cosinus de `pi`. Més endavant veurem com podem ajustar la quantitat de decimals que ens dóna el **R** en els resultats.

El sinus de π radians (180 graus) és ben conegut que dóna zero. Segurament el **R** no us ha donat zero exactament, sinó una cosa com ara `1.224606e-16`. Això vol dir 1.224606×10^{-16}, o sigui 0.00000000000000001224606, que a tots els efectes pràctics és igual a zero.

És inevitable que hi hagi imprecisió en els càlculs matemàtics. Hi ha maneres de reduir aquesta imprecisió tant com es vulgui, però de moment no ens en preocuparem gaire.

[3] Anomenat també "barra baixa"; en anglès, *underscore*.

6 § A la consola de la **RGui** podem usar les tecles amunt i avall per recuperar instruccions anteriors i tornar-les a executar. També es pot copiar i enganxar de la consola o cap a la mateixa consola.

Proveu la instrucció `history()` (en Windows i Linux) o el botó `Show/Hide R command history` (en MacOS). Obrirà una altra finestra filla (o un panel lateral), amb la llista de les últimes instruccions escrites, i des d'on podem copiar i enganxar a la consola. La quantitat d'instruccions que queden guardades depèn del sistema operatiu, però es pot controlar amb un paràmetre (només Windows i Linux):

```
history(max.show=25)
```

//Proveu de posar altres números diferents de 25, que és el valor per omissió en Windows.

La instrucció

```
savehistory("sessio.Rhistory")
```

guarda la història d'instruccions en un fitxer anomenat `sessio.Rhistory` (el nom és arbitrari). (En MacOS, no està implementada aquesta funció, i s'ha de fer amb el botó `Save History`, després d'obrir el panel lateral). En principi, el fitxer queda en el nostre directori habitual de documents, l'anomenat directori *home* (casa) del usuari, que en Windows sol ser `C:\Users\usuari\Documents`, en MacOS `C:\Users\usuari`, i en Linux `/home/usuari`. Parlarem sobre directoris més endavant. Un cop guardat un fitxer d'història, es pot tornar a carregar un altre dia fent

```
loadhistory("sessio.Rhistory")
```

i tornarem a disposar de la història amb les tecles amunt/avall i la instrucció `history()`. En el menú `File` veureu que hi ha entrades `Load history ...` i `Save history ...` que executen aquestes instruccions. En MacOS no hi ha aquestes entrades i s'ha d'usar el botó `Load History`.

De tota manera, per raons que aniran quedant clares més endavant, aquest mecanisme de guardar i carregar història no s'utilitza gaire a la pràctica.

En el menú `Edit` trobareu l'entrada `Clear Console` que neteja la consola, sense esborrar res de la història.

7 § Des del menú `Edit > GUI preferences ...` es poden canviar alguns aspectes visuals.

Es pot escollir entre MDI (Multiple Document Interface) o SDI (Single Document Interface). En el primer cas les finestres de la consola, la història i altres s'obren totes dins d'una mateixa finestra mare, la **RGui**, que és la que conté els menús de l'aplicació, diferents segons quina de les finestres filles està activa. La instal·lació per omissió en Windows posa el **R** en aquest mode. Des d'aquí es pot canviar a SDI.

En el mode SDI, no hi ha una finestra mare, sinó que les filles es mouen independentment per l'escriptori. Les finestres filles contenen els menús pertinents. Aquesta és la instal·lació per omissió en Linux i MacOS. En MacOS, com sabeu els usuaris, els menús van integrats a la barra general.

Va totalment a gust de cadascú escollir entre una o altra opció. Si de cas, l'opció SDI fa molt còmode usar el **R** com a calculadora ràpida, sense haver d'obrir tota la interfície.

També podeu escollir en aquest quadre de diàleg el tipus de lletra i els colors.

8 § Per tancar el programa, podeu escriure a la consola `quit()`, o abreviadament `q()`, o simplement tancar la finestra de la manera ordinària. En tots els casos ens pregunta si volem guardar el *workspace*. La resposta ha de ser sempre **NO**, excepte si teniu raons de pes per fer el contrari.

Guardar el workspace vol dir guardar les variables que tenim definides i altres detalls de la sessió en un fitxer que s'anomena `.RData`. Aquest fitxer es llegeix la propera vegada que arrenquem el **R**, i ens deixa en l'estat en què vam tancar. És mala idea guardar el workspace, per la interferència que pot representar amb la sessió actual. Veurem que hi ha altres maneres molt més segures de guardar informació que es pugui reutilitzar amb fiabilitat.

9 § A l'assignaura d'Anàlisi Exploratòria de Dades veureu com treballar amb el **R** dins d'un *Integrated Development Environment* (IDE) anomenat **Rstudio**. Un IDE és una aplicació que proporciona un entorn de treball més còmode i complet per a un determinat llenguatge de programació o sistema interactiu.

En aquests apunts suposarem que esteu en la **RGui**, excepte quan es digui el contrari. De tota manera, essencialment també podeu seguir les instruccions a la consola del **Rstudio**.

En tot cas, instal·leu sempre el **Rstudio** *després* del **R**.

5 Help!

10 § La manera bàsica d'obtenir ajuda sobre una funció del **R** és usar la funció `help()`. Com a argument es posa el nom d'allò sobre el que volem ajuda.

Si no heu canviat la configuració per omissió, al executar la funció `help()` s'obre una pestanya del nostre navegador web i l'ajuda apareix en aquesta pestanya. Mireu per exemple l'ajuda d'algunes instruccions que ja hem vist:

```
help(history)
help(log10)
help(sqrt)
```

La pàgina conté una descripció, la sintaxi, la definició i opcions per omissió de tots els arguments, el valor retornat per la funció, unes quantes funcions relacionades ("see also") i uns exemples d'ús.

També podeu fer `help(help)`, que ens mostra "ajuda sobre `help()`". En ella veureu que hi ha un argument anomenat `help_type` i que pot prendre 3 valors: `"html"`, `"text"` i `"pdf"`. El primer és el valor per omissió, que com heu vist ens presenta l'ajuda com una pàgina web, amb enllaços a temes relacionats.

Proveu el segon amb qualsevol de les instruccions d'abans. Per exemple, feu `help(sqrt, help_type="text")`. Veureu que la sortida apareix en una finestra dins de la **RGui**.

La tercera opció crea un fitxer PDF i el guarda, en principi, en el nostre directori *home*. Per veure'l, cal buscar-lo allà i obrir-lo amb un visor de PDF. És més incòmode, però és llegeix molt millor si ens volem mirar una ajuda sense presses.

Una altra opció, no tan directe, és usar l'entrada del menú `Help > Html help` (o equivalentment, escriure `help.start()` a la consola), que carrega en el navegador web una pàgina principal des de la qual es pot navegar cap on es desitgi. Proveu-ho, seguint després els enllaços `Packages` i `base`. La llista que veieu és la de totes les funcions contingudes en el "paquet" anomenat *base*, que és un dels que estan "carregats en el R" de seguida que l'arrenquem. Més endavant veurem què vol dir un *package*.

11 § Les pàgines web que hem vist estan en el nostre propi ordinador. No hem anat a cap lloc d'internet a buscar-les. Per tant, l'ajuda està disponible encara que no tinguem connexió a internet. La tenim en el disc dur.

Se sap que una pàgina web està en el nostre propi ordinador quan l'adreça comença per `127.0.0.1` o per `localhost`, o quan comença amb el protocol `file:///` en lloc del típic `http://`

Per altra banda, la informació que hi ha a internet sobre el **R** és immensa. Quan estigueu aturats amb una cosa que no us funciona, no dubteu a usar Google o altre buscador amb la instrucció que no us funciona o el missatge d'error que no enteneu. La probabilitat és molt alta que en poca estona estigueu altre cop en marxa.

Al paràgraf 4 § vam veure la llista de paraules reservades que no es poden usar com a noms de variables en **R**. Aquesta llista també es pot veure fent `help(reserved)`. Proveu-ho. Veureu que a més d'aquelles paraules, tampoc es poden usar variables que comencin per dos punts seguits d'un dígit (possiblement a ningú se li hauria acudit fer tal cosa de tota manera!).

Observeu també que hi ha funcions que comparteixen pàgina d'ajuda perquè estan molt relacionades entre sí: Al demanar ajuda sobre `log10`, ens parlen també dels altres logaritmes, i de la funció exponencial. Surt la mateixa pàgina si fem `help(exp)`. El mateix passa amb la funció valor absolut `abs()` i l'arrel quadrada `sqrt()`.

6 Objectes

12 § Fins ara hem vist variables i com assignar valors numèrics a una variable, amb instruccions com ara `x <- 3`. Veurem ara que el **R** manipula en general *objectes*, i el que hem anomenat fins ara *variables* és només un dels tipus d'objectes que hi ha.

Tot objecte té un tipus o *mode*. Els modes bàsics són: "numeric", "character", "logical". Proveu:

```
1 x <- 3
2 mode(x)
3 x <- "pastanaga"
4 mode(x)
5 x <- TRUE
6 mode(x)
```

El mode és *numèric* quan la variable conté un nombre, tant sigui enter com amb decimals. El mode és *character* quan la variable conté una cadena de caràcters. Les cadenes estan sempre delimitades per cometes. El mode és *logical* quan la variable conté algun dels valors `TRUE` o `FALSE`.

Observeu que al canviar el valor assignat a una variable podem canviar el seu mode. Això pot ser convenient però també té els seus perills, en forma d'errors difícils de trobar.

13 § Un *vector* és un objecte que conté una col·lecció de valors del mateix mode. Els vectors és creen amb la funció `c()` (penseu-la com abreviatura de "concatenar"), i s'assignen a un nom de la mateixa manera que ja hem vist. Per exemple:

```
weight <- c(60, 72, 57, 90, 95, 72)
weight
[1] 60 72 57 90 95 72
mode(weight)
```

Hem creat un vector numèric que conté elements. Quan invoquem el nom del objecte, ens imprimeix els sis valors. Si voleu saber què vol dir el `[1]` que apareix abans de cada sortida, construïu un vector llarg, de manera que al imprimir-lo el resultat ocupi més d'una línia.

Una variable és a tots els efectes un vector que té un element.

Suposem que els valors que hem dipositat a la variable `weight` són els pesos, en kg, de sis persones, i que les seves alçades, en metres, i donades en el mateix ordre, són les que posarem ara en el vector `height`:

```
height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
bmi <- weight/height^2
bmi
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
```

A la segona línia estem dividint el pes pel quadrat de l'alçada, obtenint un valor conegut com a "body mass index". Observeu que la variable `bmi` on hem dipositat el resultat conté també un vector. La fórmula s'ha aplicat component a component als dos vectors involucrats. Aquesta és una característica fonamental del **R**:

Les operacions entre vectors es realitzen component a component.

Una altra característica relacionada és la anomenada "regla de reutilització":

Si, al fer operacions entre vectors, un d'ells té una longitud més curta que els altres, quan arribem al final es torna a llegir des del principi.

Per exemple, `height*c(1,-1)` produeix un vector on els components parells de `height` queden amb el signe canviat. Comproveu-ho. Aquesta característica també l'hem usada a la fórmula de la segona línia: El 2 que representa elevar al quadrat es considera com un vector d'un sol component, que anem reutilitzant fins que s'acaba el vector `height`.

Observeu que en l'expressió `height*c(1,-1)` hem usat un vector sense donar-li nom prèviament. I com que tampoc no hem assignat el resultat a un nom, el **R** ens l'imprimeix directament.

De vegades convé crear vectors buits. Ho fem amb les funcions `numeric()`, `character()`, `logical()`.

```
x <- numeric()
x
mode(x)
x <- character()
x
mode(x)
x <- logical()
x
mode(x)
```

També podem crear vectors de longitud determinada sense especificar el seu contingut. La longitud es passa com a argument a les funcions anteriors. El resultat depen del mode del vector. Observeu el que surt:

```
  x <- numeric(10)
  x
  x <- character(10)
  x
  x <- logical(10)
  x
```

14 § Una *llista* és un objecte que conté una col·lecció de valors, no necessàriament del mateix mode. Creem una llista amb la funció `list()`. Proveu:[4]

```
dades <- list("Red", c("Orange", "Green"), 42,
              c(3, 14, 16), TRUE, c(FALSE, FALSE))
```

Imprimiu l'objecte creat i observeu molt bé com és la sortida i assegureu-vos que l'enteneu. Podeu provar de construir altres llistes. En particular observeu que dins una llista poden haver-hi vectors, i de fet, altres llistes:

```
dades <- list(list("Red", 0, FALSE), list("Green", 1, TRUE))
```

Imprimiu i observeu atentament la sortida.

Exercici: Fes una llista en què cada element contingui el nom i l'edat dels membres de la teva família.

15 § Un cop construïts un vector o una llista, hem de poder accedir als seus elements individuals. En el cas dels vectors, escriurem, per exemple, `weight[3]` per accedir al tercer element. En el cas de les llistes, usarem dos parells de parèntesis quadrats: `dades[[2]]`.

Exercici: Com que aquest `dades[[2]]` és una altra llista, hem de poder accedir als seus elements individuals. Com creieu que es fa? Quina diferència hi ha entre aquestes dues instruccions?:

- `dades[[2]][1]`
- `dades[[2]][[1]]`

[4] Podeu escriure-ho en una sola línia, o bé en més d'una. En aquest últim cas, veureu que el *prompt* canvia a `+`, indicant que no hem acabat la instrucció.

16 § Trobareu que molt sovint heu de crear vectors amb seqüencies sistemàtiques de nombres. Això es pot fer amb la funció `seq()` (abreviatura de "sequence"). En la forma més bàsica, la funció pren tres arguments, `from`, `to`, `by`. Proveu com funciona amb els exemples següents:

```
y1 <- seq(from=1, to=100, by=1)
y2 <- seq(from=1, to=100, by=2)
y3 <- seq(from=50, to=100, by=10)
y4 <- seq(from=100, to=1, by=-1)
y5 <- seq(from=-5, to=5, by=0.2)
```

Si posem, per exemple, `y2 <- seq(1, 100, 2)` funciona igual (comproveu). Les paraules `from`, `to`, `by` són els *noms del arguments* de la funció `seq`. Podem posar-los o no. Si no els posem, hem de posar els arguments exactament en l'ordre anterior. En canvi, usant els noms dels arguments, els podem posar en qualsevol ordre (comproveu). A més, amb els noms posats, és segurament més fàcil d'entendre el que fa la funció. Ara no té molta importància, però ja veureu que escriure instruccions clares la tindrà. I és més fàcil recordar els noms dels arguments que l'ordre en què s'han d'especificar.

L'expressió `1:15` és una manera abreviada de fer `seq(from=1, to=15, by=1)`. Proveu:

```
1:15
2*1:15
2*(1:15)
(2*1):15
15:1
```

Observeu que els dos punts (`:`), tenen *precedència* respecte la multiplicació. Per desfer la precedència, cal usar parèntesis. En cas de dubte, millor sempre usar parèntesis. A més, l'expressió `2*(1:15)` és més clara que `2*1:15`.

La funció `rep()` (de "repeat") s'utilitza per construir un vector repetint un objecte una quantitat especificada de vegades.

```
rep(8, times=5)
x <- 1:4
rep(x, times=5)
```

Aplicat a llistes, `rep()` "concatena" les llistes en una nova llista. No fa una "llista de llistes":

```
salutacions <- list("hola", "adéu")
rep(salutacions, times=3)
```

La funció `length()` retorna la longitud d'un vector o llista.

```
length(y5)
length(salutacions)
length(rep(salutacions, times=3))
```

La longitud, igual que el mode, és un atribut que tenen tots els objectes del **R**.

17 § Hem vist abans que hi ha objectes de mode *logical*. Anem a treballar amb ells. Recordeu les definicions dels vectors `weight`, `height`, `bmi`, del paràgraf 13 §. Comproveu que encara els teniu definits o torneu-los a entrar.

Es diu que una persona té sobrepès si el seu índex de massa corporal (el `bmi`) és superior a 22.5. L'expressió `bmi > 22.5` té un valor `TRUE` o `FALSE` segons l'expressió sigui veritat o mentida. Com que `bmi` és un vector, el resultat serà un vector *logical*, que ens dirà si cadascuna de les persones de la llista té sobrepès o no.

```
sobrepes <- bmi > 22.5
bmi[sobrepes]
```

Què ha fet la segona instrucció? Dins dels parèntesis quadrats hem posat un vector lògic. El resultat és seleccionar només aquells valors en què el vector lògic té el valor `TRUE`. Queda un vector de longitud més petita.

Hi ha altres maneres de seleccionar només una part d'un vector. Per exemple, proveu i mireu d'entendre què fan:

```
bmi[1:4]
bmi[2:5]
bmi[c(1,1,1,6,6,6)]
bmi[-6]
bmi[-(1:3)]
```

De la mateixa manera que seleccionem una part d'un vector per construir un altre vector, també podem canviar els valors del vector original. Experimenteu:

```
x <- 1:20
x[3] <- 10
x[7:9] <- 0
```

i tot el que se us pugui acudir.

El símbol `>` és un exemple de *operador lògic*. Representa la pregunta "és el que hi ha a l'esquerra més gran que el que hi ha a la dreta?", i la resposta només pot ser, naturalment, `TRUE` o `FALSE`.

```
4 > 3
3 > 4
3 > 3
x <- 1:5
x > 3
y <- 3:7
x > y
```

Reflexioneu sobre l'últim resultat fins que el tingueu clar. Recordeu que les operacions entre vectors es fan "component a component".

Experimentem més:

```
x <- 1:5
length(x)
y <- -5:14
length(y)
x > y
```

Reflexioneu sobre l'últim resultat recordant la "regla de la reutilització".

Fem encara un altre experiment:

```
x <- 1:5
length(x)
y <- -5:12
length(y)
x > y
```

Obtenim un resultat, però també un "warning". El **R** no considera que sigui un error, però ens avisa que el que estem fent és estrany i potser no és el que volíem. El vector `y` té una longitud que no és múltiple de la longitud de `x`. Per tant, quan anem reutilitzant `x` arriba un moment que el vector `y` s'acaba i no hem acabat el `x`. El resultat és un vector més curt. Les dues últimes comparacions no s'han fet.

Hi ha altres operadors lògics:

`<` `<=` `>` `>=` `==` `!=`

que pregunten, respectivament,

- és més petit que?
- és més petit o igual que?
- és més gran que?
- és més gran o igual que?
- és igual que?
- és diferent que?

Els dos últims són els que tenen una sintaxi menys intuïtiva, i és molt fàcil equivocar-se amb ells. Si fem `x = 3` no estem preguntant. Això és una assignació, equivalent a `x <- 3` com vam comentar al paràgraf 2 §. Acostumeu-vos a no usar mai el `=` sol, i reduireu la probabilitat d'errors.

Exercici: Experimenteu amb tots ells per comprovar que els enteneu.

Hi ha encara uns altres operadors lògics que operen entre valors o vectors lògics. Si `p` i `q` són vectors lògics, `p & q` és la seva *conjunció*, `p | q` és la seva *disjunció*, i `!p` és la negació de `p`.

- La conjunció de dos valors lògics és `TRUE` si i només si tots dos valors són `TRUE`.
- La disjunció és `TRUE` si i només si almenys un dels dos valors és `TRUE`.
- La negació és `TRUE` si i només si el valor original és `FALSE`.

Exercici: Intenteu endevinar els valors que sortiran en les tres últimes instruccions, abans d'executar-les:

```
p <- c(TRUE, TRUE, FALSE, FALSE)
q <- c(TRUE, FALSE, TRUE, FALSE)
!p | q
!(p & !q) | p
(p & !q) & !p
```

18 § Un vector numèric o lògic pot contenir també el valor `NA`, que es llegeix com "Not Available". És un valor indefinit, i normalment una operació que involucri un `NA` donarà com a resultat un `NA`. Per exemple:

```
x <- 1:5
x <- c(x, NA)
x > 3
is.na(x)
```

A la segona línia hem afegit al final del vector `x` un valor `NA`. Veiem com el resultat de la comparació és també `NA` per aquest valor. A l'última línia hem usat una funció que retorna `TRUE` o `FALSE` segons si el l'argument és `NA` o no.

Els valors `NA` es poden produir com a resultat de certes manipulacions. Per exemple, si tenim un vector de 5 components, com el `x` anterior, i fem `x[6] <- 0` estem afegint un sisé element (és equivalent a `x <- c(x, 6)`). Si fem `x[10] <- 0`, ens afegirà uns quants valors `NA` per completar els deu elements que ara tindrà el vector. Comproveu totes aquestes afirmacions.

Hi ha un altre valor especial que es pot produir com a resultat d'un càlcul: El valor `NaN`, que es llegeix com "Not a Number". Per exemple, intenteu fer l'arrel quadrada d'un número negatiu.

També produeixen `NaN`'s les operacions `0/0` i `Inf - Inf`. En canvi no ho fan les operacions `1/0`, `-1/0`, `Inf-5`, `2*Inf`, `(-1)*Inf`. Comproveu-ho. `Inf` representa un valor més gran que qualsevol nombre real, i que habitualment anomenem *infinit*. (Si heu estudiat *límits*, veureu que les operacions anteriors coincideixen amb les idees que ja teniu.)

Podem usar el valor `Inf` per exemple en `history(max.show=Inf)`.

La funció `is.na()` que hem vist abans retorna `TRUE` tant si l'argument és un `NA` com un `NaN`. Per distingir-los hi ha la funció `is.nan()`, que només retorna `TRUE` per `NaN`'s.

Exercici: Intenteu predir quin serà el resultat de les següents operacions i comproveu-ho:

```
y <- c(3, -2, 5, NA, 4, -1, 7)
y[y>0]
y[!is.na(y) & y>0]
y[!is.na(y) & y<0] <- -y[!is.na(y) & y<0]
print(y)
```

7 Scripts

19 § El **R** permet treballar interactivament com hem fet fins ara, però també és útil per fer *scripts*, que són programes fets d'instruccions de **R**. Els scripts es guarden fitxers en el disc dur, i es poden executar sempre que calgui.

Des del menú de la **RGui**, feu `File > New script`. S'obrirà una nova finestra, amb títol **R Editor**, que és un editor bàsic (com veureu, el **Rstudio** té un editor més complet, però de moment no ens cal).

Escriviu, en aquesta finestra:

```
weight <- c(60, 72, 57, 90, 95, 72)
height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
weight
height
bmi <- weight/height^2
bmi
bmi[bmi>22.5]
which(bmi>22.5)
```

Bàsicament, són instruccions que hem usat abans, excepte la última: La funció `which()` actua sobre un vector lògic i ens diu quines components del vector tenen valor `TRUE`.

En el menú, feu `Edit > Run all`. Les instruccions s'executaran totes a la vegada en la consola.

Feu ara `File > Save as...` Possiblement, se us obrirà el quadre de diàleg per guardar un fitxer en el vostre directori *home*. Guardeu-lo en tot cas allà on el **R** us suggereix. Més tard ja el moureu al lloc que vulgueu. Anomeneu-lo `script1.R` .

Tanqueu la finestra de l'editor. Feu `File > Open script...` i busqueu i seleccioneu el fitxer que acabeu de guardar. Ara el tornareu a tenir a la finestra **R Editor** i podeu tornar a executar totes les instruccions juntes, o fer alguna modificació i tornar-lo a guardar (si es volen conservar les dues versions).

El que hem fet és un programa, molt senzill, en què totes les instruccions s'executen automàticament una rere l'altra, talment com si les escrivíssim a la consola. Anem a fer-ho d'una manera diferent.

20 § Feu

```
source("script1.R")
```

Si no hi ha cap missatge d'error, acabeu d'executar el mateix script d'abans. Per què no surt res a la consola? Perquè ens ha faltat usar la funció `print()`, que és imprescindible quan volem executar scripts d'aquesta manera (que en el futur serà la majoria de vegades!)

Torneu a obrir, si no el teniu ja, el fitxer `script1.R` des de `File > Open script...` i escriviu els `print()` explícits que manquen.

```
weight <- c(60, 72, 57, 90, 95, 72)
height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
print(weight)
print(height)
bmi <- weight/height^2
print(bmi)
print(bmi[bmi>22.5])
print(which(bmi>22.5))
```

Guardeu el fitxer amb nom `script2.R`, i feu `source("script2.R")`.

Ara surten els resultats. Però com que no està clar què és cada cosa, és convenient ajudar-se amb més `print()`. Aprofitem per introduir alguna informació més:

```
weight <- c(60, 72, 57, 90, 95, 72)
height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
print("nombre de persones:")
print(length(weight))
print("pesos:")
print(weight)
print("alçades:")
print(height)
bmi <- weight/height^2
print("index de massa corporal (bmi):")
print(bmi)
print("quants tenen bmi excessiu:")
print(length(bmi[bmi>22.5]))
print("valor dels bmi excessius:")
print(bmi[bmi>22.5])
print("persones amb bmi excessiu:")
print(which(bmi>22.5))
```

Guardeu-ho com a `script3.R` i executeu-lo amb `source("script3.R")`.

La instrucció que hem introduït ara per imprimir la quantitat de persones amb `bmi` per sobre de 22.5, té una alternativa amb truc: La funció `sum()` suma totes les components d'un vector. Els vectors lògics són com vectors numèrics on `TRUE` és igual a 1, i `FALSE` és igual a zero. Per tant, també podem escriure

```
print(sum(bmi>22.5))
```

21 § El **R** manté en tot moment un *working directory*, el directori de treball. Podem saber quin és fent `getwd()` (abreviatura de "get working directory"). Veureu que és el lloc on heu guardat abans els scripts.

Per poder executar scripts en altres directoris, podem fer dues coses, i quina és millor depèn de les circumstàncies.

1. Escriure com a argument de `source()` el camí complet on és el nostre fitxer. Per exemple `source("D:/UAB/CursR/script3.R")`.
2. Canviar el working directory amb `setwd()` (abreviatura de 'set working directory'). Per exemple `setwd("D:/UAB/CursR/")`, seguit de `source("script3.R")`.

Fixeu-vos, sobretot els que useu Windows, que les barres inclinades són aquestes (`/`) i no aquestes (`\`).

22 § Si volem saber els noms dels objectes que tenim definits, podem usar la funció `objects()` o, més curt, `ls()`, abreviatura de 'list'. Proveu-ho ara.

Per "esborrar" un nom d'objecte (o sigui, tornar-lo a deixar indefinit) s'usa la funció `rm()` (penseu-ho com abreviatura de 'remove'). L'argument ha de ser un nom de variable o una llista de noms de variable. Feu per exemple

```
rm(x)
x
```

Quan tanqueu la sessió de R, recordeu de no guardar la "workspace image". Així tindreu una sessió neta, sense objectes, la propera vegada que l'arrenqueu.

8 Més objectes: matrius i data frames

23 § Una *matriu* és un altre tipus d'objecte del R. És un conjunt d'elements del mateix mode, com els vectors, però disposats en un rectangle. És un concepte de moltíssima utilitat en Estadística. Una matriu té un cert nombre de files i un cert nombre de columnes. Es creen amb la funció `matrix()`, a partir d'un vector, donant la quantitat de files i de columnes.

```
x <- 1:24
print(x)
m <- matrix(x, nrow = 6, ncol = 4, byrow = FALSE)
print(m)
m <- matrix(x, nrow = 6, ncol = 4, byrow = TRUE)
```

Si no s'especifica `byrow`, pren el valor `FALSE`, i per tant la matriu s'especifica per columnes. Es diu que `FALSE` és el valor *per omissió* del paràmetre `byrow`. I com que la quantitat de columnes es dedueix de la quantitat de files, i viceversa, només cal posar una de les dues coses (però millor posar les dues, per claredat i per detectar possibles errors).

Fixeu-vos en les etiquetes de files i columnes que posa el R en la sortida. És el que s'ha d'escriure per seleccionar com a vector una fila o columna. Per exemple:

```
print(m[2,])
print(m[,3])
```

Una matriu té dues "dimensions". Un *array* és una generalització a més dimensions. En lloc de files i columnes, especifiquem un vector donant la magnitud de cada dimensió:

```
x <- 1:24
m <- array(x, dim=c(3, 4, 2))
print(m)
```

Ara, `m` és un array amb 3 "files", 4 "columnes" i 2 "capes" (o com vulgueu anomenar-ho). Fixeu-vos com ho presenta el R quan li demanem que escrigui `m`.

Les regles del reciclatge s'apliquen com sempre. Anem a construir un array 3-dimensional amb les mateixes dimensions que el `m`, contenint el número 2 a tot arreu. Després farem la suma i el producte dels dos arrays, que com veureu són operacions que es fan element a element[5].

```
n <- array(2, dim=c(3,4,2))
print(n)
m_mes_n <- m+n
print(m_mes_n)
m_per_n <- m*n
print(m_per_n)
```

[5]Si heu estudiat matrius, sabreu que aquest producte no és el que habitualment s'anomena en matemàtiques "producte de matrius", que és molt més útil.

24 § Al 14 § hem parlat de llistes. Els elements d'una llista poden tenir noms, per tal d'accedir-hi sense necessitat de saber quin és l'ordre de la llista, que sol ser irrellevant.

```
Simpsons <- list(home="Homer", dona="Marge",
                 nombre.fills=3, edat.pares=c(36, 34),
                 edat.fills= c(10, 8, 1))
```

Ara, en lloc de posar `Simpsons[[1]]`, podem posar `Simpsons[["home"]]`. I encara hi ha una manera més convenient i més usada: `Simpsons$home`.

Per exemple:

```
Simpsons
Simpsons$dona
Simpsons$edat.fills
Simpsons$edat.fills[2]
n_fills <- "nombre.fills"
Simpsons[[n_fills]]
Simpsons <- c(Simpsons, list(nom.fills=c("Bart", "Lisa", "Maggie")))
Simpsons$nom.fills
```

Les matrius també poden tenir noms a les files i les columnes: Tornem a la matriu `m` del paràgraf 23 §:

```
m <- matrix(1:24, nrow = 6, ncol = 4, byrow = FALSE)
colnames(m) <- paste("C", 1:4, sep="")
rownames(m) <- paste("F", 1:6, sep="")
m
m[ ,"C2"]
m["F4", ]
```

Per posar noms a les columnes i files hem usat la funció `paste()`, que enganxa diferents objectes. Feu `help(paste)` per veure la idea. Aquí, per exemple, estem enganxant la cadena `"C"` amb cadascun dels números 1,2,3,4 (recordeu que els vectors es "reciclen"), sense deixar espai de separació. El resultat són les cadenes `"C1"`, `"C2"`, etc. La separació per omissió és un espai (`sep=" "`).

Veiem en les últimes instruccions com podem usar els noms de files i columnes per seleccionar una determinada fila o columna. També és possible seleccionar una submatriu: `m[ , c("C2", "C4")]`

25 § Un altre tipus d'objecte de **R** és el *data frame*. Escriviu a la consola

```
airquality
```

El **R** ve amb unes quantes dades d'exemple, que es poden cridar directament pel seu nom a la consola. (Més endavant parlem d'aquests *datasets*.) El `airquality` és una d'elles.

Veureu que us ha sortit un conjunt de números organitzat en files i columnes. Aquesta és la manera habitual com es presenten les dades en Estadística. Típicament, cada fila és un "cas" o "observació" (un individu d'una població), i cada columna representa una "variable" (una característica de cada individu). En aquest cas, els individus són dies concrets en què es van prendre unes mesures de qualitat de l'aire a la ciutat de Nova

York, i les variables són: La proporció d'ozó, la radiació solar, el vent, la temperatura, el mes, i el dia del mes.

Aquesta taula és un exemple de data frame. Els data frames es poden pensar com a llistes, els components de les quals són vectors (les columnes del data frame). Per exemple, la instrucció `airquality$Ozone` ens retorna el vector numèric de les mesures d'ozó. Els `NA` (vegeu el 18 §) corresponen segurament a dies en què no es va poder fer la mesura o es van perdre després les dades. En aquests casos es parla en general de *missing data*.

Els data frames també es poden pensar com arrays 2-dimensionals (matrius), i per tant també es pot accedir a les dades com en els arrays. En el tros de codi següent, seleccionem la columna de les temperatures i calculem la seva mitjana amb la funció `mean()`; després fem el mateix però agafant només les temperatures del més d'agost; la última instrucció fa el mateix que la tercera, fent la selecció a l'estil dels arrays.

```
temperatures <- airquality$Temp[ !is.na( airquality$Temp)]
mean(temperatures)
temperatures_ago <- airquality$Temp[airquality$Month == 8]
mean(temperatures_ago)
airquality[airquality[ , 5] == 8, 4]
```

Exercici: Què passa si volem fer així la mitjana de la radiació solar el mes d'agost? Com li traiem els `NA`, seleccionem agost i calculem la mitjana en una sola instrucció?

Mireu les dades anomenades `swiss` i feu `help(swiss)` per veure què són. També estan organitzades en un *data frame*, i la única diferència apreciable amb el de `airquality` és que els casos (és a dir, les files) tenen noms en lloc d'un número d'ordre. Aquest noms formen un vector de caràcters i s'hi accedeix amb la funció `row.names()`.

```
row.names(swiss)
swiss[row.names(swiss)== "Courtelary", ]
```

En algunes circumstàncies pot ser còmode veure el *data frame* en una graella.

Podem fer-ho amb `edit(swiss)`, que obre una nova finestra **Data Editor**. De fet, podem canviar les dades en aquesta graella i que quedin guardades si fem una assignació, com ara `swiss <- edit(swiss)`. De tota manera, no és aconsellable editar un data frame d'aquesta manera, perquè no queda constància de

9 Gràfics i funcions

26 § El **R** és molt bo fent gràfics de tota mena. Aquí veurem només com fer la gràfica d'una funció matemàtica $f(x)$: La instrucció `plot(x,y)`, on `x` i `y` són dos vectors de la mateixa longitud dibuixa el punts $(x[1], y[1])$, $(x[2], y[2])$, etc, en uns eixos de coordenades. Proveu:

```
x <- c(1,2,3,4)
y <- c(4,3,5,3)
plot(x,y)
```

Hi ha molts arguments addicionals per ajustar colors, símbol per als punts, rang dels eixos horitzontal i vertical, afegir text, etc.

Anem a dibuixar la funció $f(x) = \sqrt{x} + \sin(x)$ entre 0 i 10. Per això, primer creem un vector que contingui el valors del 0 al 10 de centèsima en centèsima; després creem un vector amb les imatges de tots aquests punts; finalment fem el dibuix.

```
x <- seq(from=0, to=10, by=0.01)
y <- sqrt(x)+sin(x)
plot(x, y)
```

Quedarà més bonic si poseu `plot(x, y, type="l")`, que uneix els punts amb una línia fina en lloc de dibuixar els punts en sí. Proveu-ho.

Quan teniu seleccionada la finestra del gràfic, en el menú podeu seleccionar `File > Save as...` per guardar-lo en diferents formats gràfics.

27 § Us haureu fixat que en diem *funcions* de **R** a expressions com ara `getwd()`, `plot(x,y)`, `mean(x)`, on apareixen parèntesis amb (potser) alguns objectes de **R** dins.

El **R** és de fet un llenguatge de programació, com ho són el Python, el C, el Java, etc. Un *programa* és una seqüència d'instruccions donades en un determinat llenguatge, que nosaltres escrivim i entenem, i que després es tradueixen a un llenguatge que les màquines entenen (simplificant molt, podem dir que es tradueixen a "zeros i uns").

Tot llenguatge de programació modern implementa el concepte de funció: Una *funció* és un conjunt d'instruccions del llenguatge que s'executaran totes en una sola línia (la *crida a la funció*). És útil definir una funció quan aquest conjunt d'instruccions fan una tasca concreta, que volem poder fer repetidament sense haver de reescriure cada cop les instruccions.

Per exemple, està clar que és més pràctic usar `mean(x)` i `var(x)` que no pas escriure cada cop la fórmula de la mitjana i la variància, adaptant-la a més cada vegada a si el vector en qüestió és diu `x` o es diu d'una altra manera i a la seva longitud concreta.

El **R** ens proporciona una bona quantitat de funcions predefinides des que arrenca. Algunes codifiquen fórmules matemàtiques senzilles, com la de la mitjana i la variància; d'altres són més complicades, com ara la funció `seq()` per crear un vector; i d'altres requereixen molts coneixements informàtics per crear-les: canviar el directori de treball amb `setwd()`, carregar un script amb `source()`, fer sortir un resultat a la consola amb `print()`, o obrir una finestra i fer un gràfic amb `plot()`.

El que hi ha dins dels parèntesis quan cridem una funció és la *llista d'arguments*. Poden ser un, com en `mean(x)`, més d'un, com en `plot(x, y, type="l")`, o cap, com en `getwd()`. Fins i tot la mateixa funció pot acceptar una quantitat variable d'arguments, com hem vist al dibuixar la gràfica, i els arguments poden ser de diferents tipus[6].

Molts cops, els arguments tenen *nom d'argument* i els podem usar dins els parèntesis. Ho hem vist amb la funció `seq()` al paràgraf 16 §. Si no posem els noms d'argument, aquests han d'anar en un ordre determinat; si els posem, els podem especificar en qualsevol ordre. Proveu:

```
seq(10, 100, 2)
seq(from=10, to=100, by=2)
seq(by=2, from=10, to=100)
```

[6] Tècnicament, són funcions diferents, encara que tinguin el mateix nom, però no cal entrar en aquest detall.

A més, els arguments poden tenir *valors per omissió*. Deduïu-los:

```
seq(from=1, to=100)
seq(to=100, by=2)
seq(from=1, by=2)
seq(to=100)
seq()
```

La funció `seq()` té a més altres arguments alternatius: podem donar la longitud del vector a crear, bé directament amb l'argument `length`, bé copiant la longitud d'un altre vector, amb l'argument `along.with`

```
seq(from=10, length=30)
seq(from=10, length=30, by=2)
x <- 1:15
seq(from=10, along.with=x, by=2)
```

Les funcions, normalment, *retornen* un resultat, i per tant es pot assignar directament la crida a una funció a un objecte de **R**. Per exemple, la instrucció `m <- mean(x)` crida a la funció `mean()` amb l'argument `x`; aquesta retorna un número, i aquest número s'assigna a la variable `m`.

28 § Nosaltres podem definir funcions noves fàcilment: La sintaxi és

```
nom_funcio <- function( llista_parametres ) { cos_funcio }
```

on nosaltres hem d'especificar el nom de la funció, la llista de paràmetres i el cos de la funció.

- El nom de la funció és arbitrari. Tan sols tingueu en compte que si hi ha algun altre objecte definit amb el mateix nom l'estarem canviant de valor.
- La *llista de paràmetres* són els noms que usarem dins de la funció per als arguments amb què cridem la funció. S'han de correspondre en el mateix ordre.
- El *cos de la funció* són les instruccions que la funció executarà.

Comencem amb un exemple molt simple:

```
quadrat <- function(x){ x^2 }
```

Aquesta funció eleva al quadrat allò que se li passi com a argument. Usem-la:

```
quadrat(3)
quadrat(1:5)
y <- 10
quadrat(y)
x <- 100
quadrat(x)
print(x)
```

Observeu que l'objecte `x` que hem definit com `x <- 100`, no té a veure amb el paràmetre `x` que s'utilitza *dins de la funció*. Quan la funció acaba, `x` segueix tenint el mateix valor 100.

La funció anterior té un sol argument. Fem-ne una amb dos arguments:

```
potencia <- function(x, p){ x^p }
potencia(2, 10)
potencia(1:5, 2)
y <- 1:10
potencia(y, 3)
```

En aquests exemples, el cos de la funció només té una instrucció, que no és el més habitual. Si n'hi ha més d'una, les podem anar escrivint a la consola, però és molt més pràctic fer un script. Feu `File > New Script` i escriviu les línies següents i executeu el script amb `Edit > Run all`.

```
resum <- function(x){
  x_bar <- mean(x)
  sigma2 <- var(x)
  paste("mitjana=", x_bar,", variancia=", sigma2)
  }
```

(La funció `paste()`, com veieu, permet enganxar diferents objectes en una sola cadena i imprimir resultats de manera més endreçada.)

Podem establir un *valor per omissió* d'alguns paràmetres. Per exemple, podem fer que la funció `potencia()` elevi al quadrat si no se li especifica el paràmetre `p`.

```
potencia <- function(x, p=2){ x^p }
potencia(4)
```

Si no donem valors per omissió, i els arguments no encaixen en quantitat i ordre amb els paràmetres, ens sortirà un error.

Com que els arguments tenen nom, al cridar la funció els podem especificar en qualsevol ordre, usant aquests noms.

```
potencia(p=2, x=10)
```

29 § El valor que retorna una funció és, en principi, el resultat de la última instrucció de la funció. Però també es pot posar explícitament una instrucció `return()`, i de vegades és necessari. Ja veurem quan.

L'objecte retornat pot ser de qualsevol tipus (llista, gràfic, ...). Completem la funció anterior i fem que la sortida sigui una llista:

```
resum <- function(x){
  x_bar <- mean(x)
  sigma2 <- var(x)
  m <- min(x)
  M <- max(x)
  list(mitjana=x_bar, variancia=sigma2, minim=m, maxim=M)
  }
```

L'executem passant-li un vector com a paràmetre i assignem el resultat a un altre objecte:

```
resultats <- resum(5:15)
print(resultats)
```

Gairebé sempre fem funcions per tenir-les guardades i reutilitzar-les quan calgui. Guardeu el codi de la funció `resum()` en un script amb nom (per exemple) `MesuresBasiques.R`. Tanqueu el **R** (sense guardar el workspace!, com sempre), i torneu-lo a obrir. Executeu

```
source("MesuresBasiques.R")
x <- 1:15
resultats <- resum(x)
print(resultats)
```

Podria haver més d'una funció definida en el script i les tindríem totes disponibles al carregar-lo amb `source()`.

10 Paquets

30 § Un *package* (paquet) de **R** és una col·lecció de funcions i/o dades. Quan el **R** arrenca, "carrega" en memòria uns quants paquets fonamentals, i per tant les seves funcions i dades les tenim disponibles de seguida. Totes les funcions que hem usat fins ara (excepte les que hem creat nosaltres mateixos) i les dades `airquality` que vam obrir al estudiar els data frames, estan en aquests paquets "precarregats".

Per veure quins són aquests paquets, feu `search()`. Veureu una sortida com la següent:

```
[1] ".GlobalEnv" "package:stats" "package:graphics" "package:grDevices"
[5] "package:utils" "package:datasets" "package:methods" "Autoloads"
[9] "package:base"
```

Sense entrar en detalls tècnics, podeu pensar aquesta llista de la manera següent: La primera entrada, `.GlobalEnv`, és el "lloc" on estan definits els objectes que nosaltres hem definit. És a dir, si feu ara `ls()` o `ls(.GlobalEnv)`, us ha de donar el mateix. En cas que el **R** no trobi un objecte que nosaltres referenciem, el buscarà en el paquet `stats`. Si feu `ls("package:stats")` veureu els objectes que hi ha definits en aquest paquet (per exemple, veureu `"var"`, que fa referència a la funció amb aquest nom que hem usat fa una estona).

> **Opcional:** Podem "veure" el objecte `var` escrivint simplement el seu nom (sense els parèntesis). El que veurem és la definició de la funció `var()`. No obstant, ens quedarem a mitges, perquè, després d'unes quantes instruccions de **R**, la funció acaba amb una instrucció `.Call`, que executa una funció escrita en llenguatge **C** i ja compilada (o sigui, traduïda a llenguatge màquina) i no es veu el seu codi. Moltes funcions bàsiques del **R** estan escrites en **C** i compilades. La seva execució és molt més ràpida perquè no s'han de traduir a llenguatge màquina mentre s'executen.

En cas que l'objecte que referenciem no estigui tampoc en `stats`, el buscarà en el següent paquet de la llista, i així successivament. L'últim que consulta és el paquet `base`. Aquí veureu noms que us seran familiars: `source`, `setwd`, `mode`, `mean`, `c`, ... En cas que no trobi aquí el nom, sortirà l'error `object not found`.

31 § Podem obtenir una llista de tots els paquets que tenim instal·lats, demanar una petita descripció del paquet, o fer la llista de tots els objectes que conté, amb una petita explicació. Proveu:

```
library()
help(stats)
library(help=stats)
```

Amb l'última modalitat, podeu veure els objectes que conté un paquet que està instal·lat, encara que no estiui carregat. Per exemple, podeu provar `library(help=MASS)`. El paquet `MASS` va ser creat per acompanyar un llibre[7] (ja una mica antic), i conté funcions i dades d'exemple.

Podem instal.lar nous paquets i carregar-los de la manera següent.

```
install.packages("combinat")
library(combinat)
library(help=combinat)
```

És molt possible que el primer cop que instal·leu un paquet us pregunti si voleu crear una biblioteca personal. Dieu-li que sí.

També podeu fer la instal·lació amb el menú `Packages > Install packages...`, seleccionant un *mirror* i després el paquet que voleu.

> **Opcional:** En **RStudio**, podem gestionar els paquets de la manera següent: A la pestanya `Packages` d'un dels panels, tenim la llista de tots els paquets instal·lats. Quan marquem el quadradet del costat del nom d'un paquet, es carrega (marcar el quadrat de `combinat` equival a escriure `library("combinat")`). Si fem click en el nom d'un paquet, obtenim informació del paquet a la pestanya `Help`. Per instal·lar un paquet cliquem al botó `Install`.

D'on estem baixant els paquets que instal·lem? Els paquets estan en *repositoris*, el més important dels quals és el CRAN *(Comprehensive R Archive Network)*, que es troba a `http://cran.r-project.org`. És el que s'utilitza per omissió.

32 § Amb els paquets precarregats quan arrenquem el **R**, n'hi ha un que s'anomena `datasets`. És l'únic d'aquests paquets que conté dades. I només conté dades; no funcions. Són dades "clàssiques" que han aparegut en articles i estudis científics; només tenen interès actual com a exemples per jugar i aprendre.

Com hem vist abans, un dels objectes que conté és `airquality`, que és un data frame. Un altre exemple és `AirPassengers`, amb dades del nombre de viatgers en vols internacionals, mensualment, entre 1949 i 1960. Mireu-lo escrivint el seu nom. Aquest no és un data frame, sinó una altra classe d'objecte que s'anomena *sèrie temporal* (*time series*), que no estudiarem en aquest curs de **R**. Podeu veure la *classe* d'un objecte amb la funció `class()`.

```
class(airquality)
class(AirPassengers)
```

Què passa si hi ha un objecte d'un paquet que té un nom igual a un objecte que nosaltres hem definit? No serà habitual, però cal saber què pot passar. Com veiem amb la funció

[7] W. N. Venables, B. D. Ripley: *Modern Applied Statistics with S*, Springer 2002. Per què "S"? Resulta que **S** és el nom original del llenguatge, i **R** és una "implementació" del llenguatge **S**. Però no és moment d'entretenir-se en aquest detall.

`search()`, el `.GlobalEnv`, que és on "viuen" els nostres objectes, va per davant de tots els altrespaquets. En particular, per davant del paquet `datasets`. Si nosaltres fem, per exemple, `airquality <- 3` (feu-ho!), a partir d'aquest moment ja no podem veure l'objecte `airquality` del `datasets` (comproveu-ho!). El tornarem a veure si esborrem el nostre objecte fent `rm(airquality)` (vegeu el paràgraf 22 §).

Anem a carregar un nou paquet, que conté dades i funcions. Feu

```
library(MASS)
data()
search()
```

La instrucció `data()` ens ensenya totes les dades que hi ha en els paquets que estan carregats. Se'ns obre una finestra amb títol **R Data Sets** on tenim, separades per paquets (que ara són `datasets` i `MASS`), totes les dades disponibles.

La instrucció `search()` ens mostra que `.GlobalEnv` segueix estant per davant del nou paquet carregat. Per tant, igual que abans, si ara definim un objecte amb nom coincident amb un dels dels paquets carregats, el nostre té precedència. Si l'objecte el teníem definit abans que es carregui el paquet, la situació és la mateixa, i en aquest cas a més el **R** ens avisa. Anem a comprovar-ho. La primera instrucció serveix per revertir el `library(MASS)` ("des-carregar" el paquet). Després definim un objecte `cats` que existeix en el paquet `MASS`. Després tornem a carregar el `MASS`.

```
detach(package:MASS, unload=TRUE)
cats <- 5
library(MASS)
```

Veurem l'avís: `The following object is masked _by_ '.GlobalEnv': cats`

El nostre objecte segueix tenint precedència i no podem usar el `cats` del `MASS`. Després de fer `rm(cats)` sí que hi podrem accedir. Proveu-ho.

Si és imprescindible accedir als dos objectes a la vegada, aleshores podem accedir al del paquet fent-hi referència com `MASS::cats`. Comproveu-ho.

Finalment, si fem `data(cats)`, el que estem fent és posar l'objecte `cats` dins el nostre lloc `.GlobalEnv`, i per tant, canviant el valor que pogués tenir un objecte anteriorment definit amb el mateix nom.

33 § Per tenir una informació preliminar d'unes dades tenim la funció `help()`; per veure els "noms" que conté (per exemple, les columnes d'un data frame), podem fer `names()`; per visualitzar les n primeres o últimes files podem usar `head()` i `tail()`; la dimensió ens sortirà amb `dim()`.

```
help(cats)
dim(cats)
head(cats)
tail(cats)
names(cats)
```

Podem també accedir i carregar al `.GlobalEnv` dades d'un paquet sense carregar-lo tot:

```
detach(package:MASS, unload=TRUE)
help(eagles, package="MASS")
data(eagles, package="MASS")
print(eagles)
```

Ara tenim el data frame anomenat `eagles`, però cap altre objecte del paquet `MASS`.

34 § En alguna ocasió voldrem esborrar tots els objectes definits en una sessió per començar alguna cosa completament diferent sense interferències. La funció `rm()` admet un argument opcional anomenat `list` una llista d'objectes (feu `help(rm)` per confirmar-ho); per altra banda, la funció `ls()` retorna una llista de tots els objectes en el `.GlobalEnv`. Per tant, una manera d'esborrar tots els nostres objectes és `rm(list=ls())`.

35 § Quan fem scripts per guardar ens agradaria deixar escrita alguna observació o comentari sobre el significat d'una expressió, o el motiu pel que fem una cosa en lloc d'una altra, o posar un títol a un tros de codi que fa una cosa específica. Tot això és útil per quan nosaltres, o una altra persona, ha d'entendre o modificar un script fet fa temps. Es fa usant el símbol *hashmark* `#`, després del qual el **R** ignora la resta de la línia. Per exemple,

```
detach(package:MASS, unload=TRUE) # descarreguem el paquet MASS
help(eagles, package="MASS") # mirem l'objecte eagles
data(eagles, package="MASS") # carreguem l'objecte eagles en
                             # el .GlobalEnv sense carregar
                             # tot el paquet MASS
print(eagles)
```

11 Llegir i escriure fitxers amb dades

36 § Baixeu del Moodle el fitxer `Biblioteques.csv` i obriu-lo amb un editor de text qualsevol; per exemple, amb el **RStudio** mateix.

L'extensió `.csv` vol dir *comma-separated values*, i típicament són fitxers de text pur, on cada línia té una sèrie de camps, separats per uns caràcters que poden ser comes, punts i coma, blancs, tabuladors, etc. El fitxer que esteu veient, concretament, té els valors separats per punts i coma, i són dades sobre biblioteques de Catalunya, publicades al IDESCAT.

Per nosaltres és un format pesat de llegir, però al ser un fitxer de text pur, és perfecte per intercanviar informació i fer-la perdurable, perquè no té cap dificultat fer un programa que el pugui llegir.

El que ens interessa es transformar aquestes dades a un objecte de la classe "data frame", per poder treballar-lo dins el **R**. Això ho fem amb la funció `read.table()`. Abans d'aplicar-la ens hem de fixar en unes quantes coses del fitxer:

1. Hi ha una capçalera amb el nom dels camps? En aquest fitxer veiem que sí: hi ha una primera línia que és clarament diferent de la resta i que sembla que descriu

què és cada camp. (Si l'editor amb què esteu veient el fitxer té activada la opció d'ajust de text (*text wrapping*), pot ser que veieu més d'una línia de capçalera, però en realitat és una sola; el caràcter (invisible) de final de línia només està darrera la paraula "social".)

2. Quin és el símbol separador de camps? Com hem dit, pot ser qualsevol símbol que no aparegui dins dels camps en sí. Típicament, trobem: comes, punts i coma, dos punts, barres verticals, espais en blanc, tabuladors (caràcter invisible, però diferent dels espais en blanc). En aquest cas, tenim punts i coma.

3. Quin símbol s'utilitza per a separar la part entera de la part decimal? Típicament, és un punt. Però també s'usa la coma. L'ús de l'apòstrof o "coma alta", comú a Espanya fins els anys 1980's en l'escriptura manual, està en desús. Naturalment, si un fitxer `.csv` usa la coma en aquest sentit, aleshores no la pot usar com a separador. En el nostre fitxer, tot són nombres enters.

4. Hi ha valors perduts? Si hi camps buits, s'incorporaran al data frame com a `NA`. Però també poden haver-hi altres símbols que denotin que no es coneix un valor, com ara un guió (`-`), un interrogant (`?`), etc. En el nostre fitxer, a la última línia hi falta el primer valor.

5. Quina és la codificació del fitxer? Si el fitxer conté caràcters accentuats (o com ara `ñ` o `ç`), saber la codificació del fitxer és important. Hi ha essencialment dues codificacions en ús actualment en la nostra zona geogràfica: "ISO-8859-1" i "UFT-8". Si un caràcter accentuat ha quedat substituït per un símbol estrany, és que l'hem obert amb la codificació equivocada. En el **Rstudio**, podem usar el menú `File > Reopen with encoding...` per obrir-lo suposant una altra codificació. En el nostre cas, el fitxer és veu bé en UTF-8.

Per carregar aquest fitxer al **R** com un data frame, usarem la instrucció

```
bib <- read.table("Biblioteques.csv", header=TRUE,
                  sep=";", encoding="UTF-8")
print(bib)
```

L'argument `header=TRUE` indica que hi ha una capçalera (per omissió, el valor és `FALSE`); `sep` indica el caràcter separador (per omissió, és `sep=""`, que s'interpreta com "espais en blanc"); finalment, especifiquem la codificació correcta. Si no s'especifica, el **R** agafa la que tingui el sistema operatiu per omissió; en el cas de Windows, és el ISO-8859-1, que és coneix també com a latin1, i s'especifica així en aquest paràmetre: `encoding="latin1"`.

Si veiem que en el fitxer hi ha per exemple els símbols `-` i `?` per especificar valors perduts, afegirem l'argument `na.string=c("-", "?")`. Si no especifiquem res, només els camps en blanc passen a ser `NA` en el data frame.

Exercici: Observeu què passa quan especifiquem cadascun dels paràmetres anteriors de manera equivocada. És important veure el resultat per ajudar-nos a detectar ràpidament que alguna cosa no ha funcionat bé en una altra ocasió.

Exercici: Baixeu del Moodle el fitxer `2018_taxa_natalitat.csv`. Examineu-lo, i després feu l'assignació `natalitat <- read.table()` amb els arguments correctes.

37 § Com haureu vist en els exemples, el **R** afegeix un número de línia al data frame resultant. Podem fer que una de les variables actuï com a nom de les files. Poseu `row.names="Literal"`, per exemple, en la lectura del data frame `bib`, per tal que el nom de les comarques passi a ser el nom de cada fila. No ens funcionaria `row.names="Codi"` perquè hi ha un valor que falta a la columna `Codi`.

Alguns fitxers de dades tenen a la capçalera un camp menys que en les files de dades. El conveni és que aleshores la primera columna actua com a nom de les files, i així ho interpreta la funció `read.table()`.

Quan no hi ha capçalera, podrem especificar el nom dels camps al fer la importació, amb l'argument `col.names`. Si no ho especifiquem, per omissió les columnes s'anomenen amb la lletra `V` seguida d'un número d'ordre.

Per omissió, les columnes que contenen cadenes de caràcters, es converteixen a *factors* (un tipus d'objecte que no estudiarem aquí). Si no es vol la conversió, s'especifica l'argument `stringsAsFactors=FALSE`. Hi ha altres arguments per tenir un control més fi de quines columnes es converteixen i quines no.

En lloc del nom d'un fitxer local es pot baixar directament un fitxer amb la seva adreça d'internet, la *url*. Per exemple:

```
url <-
 "https://vincentarelbundock.github.io/Rdatasets/csv/AER/EuroEnergy.csv"
read.table(url, header=TRUE, sep=",", encoding="UTF-8")
```

En general, el format de les dades pot ser variat i el que vulguem fer amb elles també. Quan hi ha complicacions, convé mirar l'ajuda de la funció `read.table()` o buscar a internet.

38 § Anàlogament, tenim el camí invers: Guardar un data frame en un fitxer de valors separats per comes `.csv`. La funció que ho fa és `write.table()`. Proveu de guardar el data frame `natalitat` que heu creat a l'exercici del 36 § a un nou fitxer `naixements.csv`, però separant els camps per tabuladors en lloc de comes.

```
write.table(natalitat, file="naixements.csv",
            sep="\t", fileEncoding="UTF-8")
```

El fitxer `naixements.csv` estarà en el vostre directori de treball. Obriu-lo amb el **Rstudio** o altre editor de text i observeu que el separador és ara el caràcter tabulador (invisible, però que en l'editor ocupa més d'un espai, típicament 2 o 4).

Paràmetres addicionals de `write.table()`:

- `quote`: Posant `FALSE`, no es guarden les cometes al voltant de les cadenes de text que en tinguin.
- `sep`: Especifica la cadena de caràcters que farà de separador de camps.
- `na`: Especifica la cadena de caràcters per representar les dades que falten (per omissió, és `"NA"`)
- `dec`: Especifica el caràcter a usar com a punt decimal.

- `row.names`: Posant `FALSE`, no es guardaran els noms de les files.

- `col.names`: Posant `FALSE`, no es guardaran els noms de les columnes.

Exercici: Sabem que un data frame es pot indexar com una matriu. Useu-ho per crear un nou data frame on no hi siguin les columnes corresponents a `Any` i `Taxa_mil_hab` (que són inútils perquè totes les files tenen el mateix valor). Després guardeu-lo a un fitxer, usant la barra vertical `|` amb un espai a cada banda com a separador, traient-li les cometes als camps de text, i usant la coma en lloc del punt decimal. Comproveu el resultat en l'editor de text.

39 § És possible llegir fitxers de Excel (extensions `.xls` i `.xlsx`) i crear data frames. I també exportar un data frame a un fitxer de Excel. No obstant, la manera més segura, i la única que necessiteu per ara, és la de passar pel format `csv`:

Per convertir un fitxer `.xl`| o `.xlsx` a `.csv` des de Microsoft Excel només cal fer `File > Export`. En el desplegable del diàleg, escollir CSV UTF-8 (Comma delimited).

Des de Libre Office, fer `Save as`, i en el desplegable `Save As type`, escollir 'Text CSV'. Abans d'acabar de guardar preguntarà la codificació, quin separador volem, i si volem posar cometes al voltant dels camps de caràcter.

Podeu provar-ho amb el fitxer `lungcancer.xls` que trobareu al Moodle.

40 § Opcionalment, si voleu, proveu d'importar/exportar directament a fitxers de Excel. Uns paquets moderns que fan aquesta feina (n'hi ha d'altres) són `readxl` i `writexl`, que possiblement haureu d'instal·lar.

Un cop instal·lats i carregats (vegeu paràgraf 31 §), podeu fer

```
1 as.data.frame(read_excel("lungcancer.xls"))
```

(El `as.data.frame()` és necessari perquè l'objecte que retorna el `read_excel()` no és exactament un data frame, però es converteix fàcilment).

Aquests paquets formen part d'un "univers de paquets de **R**" recent, que s'anomena *tidyverse*, molt potent, i que segur que haureu d'aprendre aviat.

12 Estructures algorísmiques

41 § El **R** és un llenguatge de programació complet, en el sentit que amb ell es pot codificar qualsevol *algorisme*, és a dir, qualsevol procediment de càlcul per produir un resultat a partir d'unes dades.

Hem vist que es poden executar diverses instruccions de **R** d'una sola tirada, fent un script, que es pot executar amb els menús de la **RGui**, o bé carregant el fitxer del script amb `source()`. Igual funciona en **Rstudio**, on a més podem tenir-lo en el panel de l'editor, i executar-lo prement el botó `source` del mateix panel.

Però a més de la pura execució seqüencial, una rere l'altre, de les instruccions, calen unes *estructures algorísmiques*, o *estructures de control*. Anem a veure com són.

En aquest apartat estareu més còmodes treballant amb el **Rstudio** en lloc de la **RGui**. Però també podeu seguir usant-la sense problema, amb l'editor incorporat **R Editor**.

42 § Alternativa `if`:

```
if ( condició ){
  # Instruccions si la condició és certa
}
```

La `condició` és una expressió que dóna un valor lògic, o sigui, `TRUE` o `FALSE`. Si és `TRUE`, s'executaran les instruccions dins de les claus. Si és `FALSE`, no s'executaran, i es continuarà executant el programa en la línia següent a la clau que tanca.

Obriu un script nou i aneu escrivint les instruccions dels exemples que anem veient.

Si esteu en el **Rstudio**, podeu anar prement el botó `Source` per executar tot, o només executar la part que seleccioneu amb el botó `Run`. Els exemples seran independents entre ells. Si esteu en l'editor de la **RGui**, podeu anar escrivint en la pantalla del script i executant amb el menú `Run all` o `Run selection`. I fins i tot podeu usar el **Rstudio** (o un altre programa) només com a editor, guardant els canvis, i després fent `source()` del fitxer des de la consola de la **RGui**.

Exemple:

```
x <- 10
if (x < 5){
  print(x)
}
if (x > 5){
  print(x+1)
}
```

La condició del primer `if` és `FALSE`, i no s'imprimeix res. La del segon és `TRUE`, i ens imprimeix el resultat de `x+1`.

Si només hi ha una instrucció a executar dins de les claus, no cal posar les claus. La posició de les claus també és arbitrària (podeu provar altres disposicions), però s'ha de procurar que el codi sigui llegible per nosaltres. La posició que veieu a l'exemple és força estàndard entre els programadors.

Així doncs, aquí podíem haver escrit també

```
x <- 10
if (x < 5) print(x)
if (x > 5) print(x+1)
```

43 § Alternativa `if ... else`:

```
if (condició){
  # Instruccions si la condició és certa
} else {
  # Instruccions si la condició és falsa
}
```

Si la condició és `TRUE`, s'executen només les instruccions dins de les primeres claus. Si és `FALSE` s'executen només les instruccions de les segones claus. Després es continua executant el programa en la línia següent de la última clau. Exemple:

```
x <- 10
if (x %% 2 == 1){
  print(paste(x, " és imparell"))
} else {
  print(paste(x, " és parell"))
}
```

L'operador %% calcula el residu de la divisió entera. Si el residu de dividir per 2 és 1, vol dir que el nombre és imparell, i ho escrivim a la pantalla. Si el residu és zero, el nombre és parell.

44 § Repetició `for`:

```
for(x in vector){
  # Instruccions a repetir
}
```

La repetició s'utilitza principalment per iterar un mateix bloc d'instruccions sobre els elements d'un vector, en l'ordre en què apareixen. El vector pot ser de qualsevol tipus. Un cop acabats els elements del vector, l'execució del programa continua en la instrucció següent després de la clau que tanca. Exemple: Calculem el quadrat de tots els números del 1 al 15 i imprimim els residus de dividir el quadrat per 5.

```
x <- 1:15
for(i in x){
  j <- i^2
  print(j %% 5)
}
```

Aquesta estructura és útil però millor si es pot evitar en favor d'aplicar al vector directament la operació que volem, perquè és molt més ràpid: `(1:15)^2 %% 5` dóna el mateix resultat.

Un altre exemple amb vectors de mode caràcter:

```
x <- c("pomes", "taronges", "maduixes", "llimones")
for (i in x){
  magraden <- paste("M'agraden les", i)
  print(magraden)
}
```

45 § Iteració `while`:

```
while( condicio ){
  # Instruccions a iterar
}
```

Les instruccions dins les claus es van repetint, mentre la condició sigui `TRUE`. Quan sigui `FALSE`, el programa continua en la instrucció següent després de la clau que tanca. Es necessari assegurar-se que la condició sigui falsa en algun moment, perquè si no el nostre

programa no acabaria mai. Afortunadament, tant des de la **RGui** com des del **Rstudio**, hi ha mecanismes per aturar el programa en cas de necessitat.

Exemple simple:

```
i <- 1
while(i < 10){
  print(i)
  i <- i + 1
}
```

A la primera iteració, la variable `i` val 1, i per tant compleix la condició. S'imprimeix el seu valor, i després la incrementem en una unitat. Tornem a avaluar la condició. Ara `i` val 2, i la condició segueix essent `TRUE`. S'imprimeix el 2, i després `i` passa a valer 3. I així successivament, fins la iteració en què s'imprimeix un 9, i la variable `i` passa a valer 10. Ara la condició dóna `FALSE`, i ja no s'executen més iteracions.

Aquest no és un bon exemple d'aplicació de l'estructura `while`, perquè sabem d'entrada quantes iteracions farem. En tal cas, és més lògic i elegant utilitzar la repetició `for`:

```
for(i in 1:9) print(i)
```

L'estructura `while` és més propi utilitzar-la quan no sabem quants cops es repetiran les instruccions. Farem un exemple una mica més elaborat per il·lustrar-ho: Suposem que volem la solució de l'equació $x^4 - 60 = 0$. Veiem a ull que la funció de l'esquerra dóna negatiu per $x = 0$, i dóna positiu per $x = 10$. A més, és una funció creixent per $x > 0$. Per tant, hi ha un únic punt $x > 0$ on la funció creua l'eix horitzontal, i aquest punt és la solució de l'equació.

Podem trobar aquest punt amb el procediment següent: Avaluem la funció en el punt mig de l'interval $[0, 10]$. Si dóna negatiu, sabem, pel mateix argument, que la solució estarà en l'interval $[5, 10]$; si dóna positiu, estarà en $[0, 5]$. I tornem a iterar. L'interval on sabem que estarà la solució serà la meitat de gran a cada iteració. Ens aturarem quan la diferència entre l'extrem dret i l'extrem esquerra de l'interval sigui més petit que una tolerància que nosaltres fixarem. Al final agafarem el punt mig de l'últim interval i podrem afirmar que el resultat és correcte amb un error més petit que la tolerància.

```
esquerra <- 0
dreta <- 10
tolerancia <- 10e-5
while(dreta-esquerra > tolerancia){
  x <- (esquerra + dreta) / 2
  if(x^4-60 < 0){
    esquerra <- x
  } else {
    dreta <- x
  }
}
x <- (esquerra + dreta) / 2
print(x)
```

Observeu que:

- Dins del `while` tenim un `if...else`. Podem tenir diferents estructures unes dins de les altres, fins al nivell de profunditat que calgui.

- No sabem d'entrada quantes iteracions farem. Cal anar avaluant la condició cada cop per saber si hem d'entrar dins el `while` o no.
- `10e-5` és una manera equivalent d'escriure `10^(-5)`, o sigui `0.00001`, habitual en informàtica.
- Si voleu veure com va evolucionant el valor de l'aproximació `x` a cada iteració, poseu un `print(x)` com a última instrucció dins el `while`.

46 § Hi ha dues instruccions especials que es poden posar dins de l'estructura `while` i que modifiquen el flux de l'execució. La primera és `break`, que atura les iteracions immediatament, sense acabar la que s'estigui fent. La segona és `next`, que atura la iteració actual i passa a la següent.

No són imprescindibles, en el sentit que sempre es pot reescriure el programa d'una altra manera per evitar-les. Però de vegades venen molt bé. Es poden usar dins de `for`, `while` i l'estructura `repeat`, que veurem després.

Per exemple, suposem que en el càlcul de la solució de $x^4 - 60 = 0$ que hem fet, volem introduir una segona tolerància, de manera que si la diferència entre $x^2 - 60$ i zero és més petita que 10^{-2}, l'aproximació ja ens val i aturem el càlcul.

```
esquerra <- 0
dreta <- 10
tolerancia <- 10e-5
tolerancia2 <- 10e-2
while(dreta-esquerra > tolerancia){
  x <- (esquerra + dreta) / 2
  if(x^4-60 < 0){
    esquerra <- x
  } else {
    dreta <- x
  }
  if(abs(x^4-60)<tolerancia2) break
}
print(x)
```

Podeu veure l'evolució del valor de $x^2 - 60$ posant un `print(x^2-60)` abans de l'últim `if`.

En aquest cas, l'ús del `break` és fàcilment evitable. Podem posar les dues condicions d'aturada juntes en el parèntesi del `while`

```
esquerra <- 0
dreta <- 10
tolerancia <- 10e-5
tolerancia2 <- 10e-2
x <- (esquerra + dreta)/2
while(dreta-esquerra > tolerancia & abs(x^4-60) > tolerancia2){
  if(x^4-60 < 0){
    esquerra <- x
  } else {
    dreta <- x
  }
  x <- (esquerra + dreta) / 2
}
print(x)
```

Ara s'han de complir les dues condicions del `while` per continuar iterant. Quan en falla una, sortim immediatament. Observeu els lleugers canvis en la resta de les instruccions, segons quina opció adoptem. Però els dos programes donen el mateix resultat. Podeu comprovar que al final del programa almenys una de les dues condicions d'aturada s'ha acomplert: O be `dreta-esquerra` $\leq 10^{-5}$, o bé $|x^4 - 60| \leq 10^{-2}$.

47 § Iteració `repeat`:

```
repeat{
  # Instruccions a iterar fins que arribi un break
}
```

Les instruccions dins les claus es repeteixen infinitament en bucle. Per tant és imprescindible que hi hagi en algun lloc una instrucció `break` que ens faci sortir del bucle. Exemple senzill: volem trobar el nombre enter k més petit tal que $k^3 > 50\,000$.

```
k <- 1
repeat{
  cub <- k^3
  if (cub <= 5*10^{4}) {
    print(paste(cub, "és més petit que 50 000. Seguirem iterant."))
    k <- k + 1
  } else {
    print(paste(cub, "és més gran que 50 000. Sortim del bucle."))
    break
  }
}
print(paste("El número és:", k))
```

Les estructures iteratives `while` i `repeat` s'assemblen, i el que es pot fer amb una es pot fer amb l'altre, canviant el que calgui canviar. Una diferència fonamental és que les instruccions del `while` pot ser que no s'executin mai, si la condició no es compleix d'entrada, mentre que les instruccions del `repeat` s'executen segur una vegada, almenys fins el `break`.

Repetim l'exemple de trobar la solució de $x^4 - 60 = 0$ (la última versió) usant `repeat` en comptes de `while`. Observeu que és natural posar la condició a l'inrevés, i al final del bloc.

```
esquerra <- 0
dreta <- 10
tolerancia <- 10e-5
tolerancia2 <- 10e-2
repeat{
  x <- (esquerra + dreta)/2
  if(x^4-60 < 0){
    esquerra <- x
  } else {
    dreta <- x
  }
  if (dreta-esquerra <= tolerancia | abs(x^4-60) <= tolerancia2) break
}
print(x)
```

13 Les funcions `apply()` i família

48 § La família "apply" és una col·lecció de funcions que faciliten certes manipulacions de arrays, llistes i data frames, tot evitant haver d'usar estructures repetitives tipus `for`.

La funció bàsica és `apply()` que actua sobre les files o columnes d'una matriu, o més en general, sobre qualsevol "llesca" d'un array de més dimensions.

Obriu un fitxer nou amb el **Rstudio**, o feu `File > New script` en el **RGui** per escriure més còmodament, i feu:

```
x <- array(1:12, dim=c(3,4))
print(x)
apply(x, 1, sum)
apply(x, 2, sum)
```

Hem "aplicat" la funció `sum()` (suma dels elements d'un vector) a cadascuna de les files de la matriu (quan el segon argument és 1), i a cadascuna de les columnes (quan el segon argument és 2). Podem utilitzar altres funcions, com ara `mean()` o `var()`, que també retornen un número per cada fila o columna de la matriu.

Apliquem-ho a un array de més dimensions:

```
x <- array( 1:24, dim=c(3,4,2))
print(x)
apply(x, 1, sum)
apply(x, 2, sum)
apply(x, 3, sum)
apply(x, c(1,2), sum)
```

Quan posem `1`, estem sumant per "files" (la primera dimensió). Observeu que la primera fila està formada per `1, 4, 7, 10, 13, 16, 19, 22`, i la suma és 92.

Quan posem un `2`, estem sumant per "columnes" (la segona dimensió). La primera columna, per exemple, té els sis elements `1, 2, 3, 13, 14, 15`.

Quan posem un `3`, estem sumant per la tercera dimensió. La primera "llesca" del array en aquesta dimensió té el números del 1 al 12 i suma 78. La segona llesca té els números del 13 al 24, i suma 222.

També podem fer `apply(x, c(1,2), sum)`, per exemple. El resultat serà la suma per a cada combinació de (fila, columna); per tant una matriu. Cada element d'aquesta matriu és la suma de dos números. Per exemple, el primer element és la suma de 1 i 13. L'últim element és la suma de 12 i 24.

49 § També podem aplicar `apply()` a un data frame. Per exemple, usant les dades `mtcars` del paquet **datasets**, amb

```
mtcars
apply(mtcars, 2, mean)
```

calculem la mitjana de totes les variables d'aquest data frame. El resultat és un vector "amb noms", els mateixos que obteniu si feu `names(mtcars)`.

La funció que apliquem també pot retornar més d'un número. Per exemple, apliqueu la funció `summary()`. El resultat ara, en lloc d'un vector, és una matriu "amb noms".

50 § La funció `lapply()` s'aplica a llistes i retorna una llista. No cal indicar-li cap "dimensió":

```
lapply(mtcars, mean)
```

Aquí l'hem aplicada a un data frame, però recordeu que el R veu els data frame també com a llistes.

Apliqueu també la funció `summary()` amb `lapply()`. Us ha de sortir una llista, els elements de la qual són vectors amb nom.

La funció `sapply()` fa el mateix que `lapply()`, però el resultat el dóna en forma de vector o matriu en lloc de llista. Si ho apliquem a `mtcars`, veiem que el resultat és idèntic a `apply()`, però sense haver d'especificar la dimensió. Proveu-ho.

51 § La funció `tapply()` s'aplica sobre un vector, partint-lo en trossos segons una altra variable, per tal d'aplicar la funció del tercer argument per separat a cadascun dels trossos.

Seguim agafant com exemple les dades `mtcars`. La segona variable `cyl` és el nombre de cilindres del motor, que poden ser 4, 6 o 8. Ens agradaria calcular la mitjana del consum (primera variable, `mpg`), i el temps per fer el quart de milla (variable `qsec`), separant per nombre de cilindres. Això ho podem fer simplement amb

```
tapply(mtcars$mpg, mtcars$cyl, mean)
tapply(mtcars$qsec, mtcars$cyl, mean)
```

En lloc de la "dimensió" hi posem el vector que ens determina les diferents classes. Proveu també de canviar `mean()` per `summary()`.

52 § La funció que posem com a tercer argument pot ser una funció estàndard del R o una funció que nosaltres definim. Podem per exemple aprofitar això per convertir la variable `mpg` (milles per galó) a litres per cada 100 quilòmetres, que és una mesura més comuna a Europa.

```
mean_consum_EU <- function(x) {
  y <- 282.4811/x
  mean(y)
}
tapply(mtcars$mpg, mtcars$cyl, mean_consum_EU)
```

14 Entorns

53 § Recordeu que dèiem a l'Apartat 10, paràgraf 30 §, que la funció `search()` retorna la llista de paquets que tenim carregats, en un ordre que és en el que "busca" un objecte quan nosaltres hi fem referència. Primer el busca en el "lloc" on nosaltres definim objectes, que és el `.GlobalEnv`; si no el troba busca en el paquet `stats`; si no el troba el busca en el següent de la llista, etc.

Anem a ser ara una mica més precisos amb la terminologia. La llista que veieu al fer `search()` és una llista de *entorns* (*environments*), començant per l'entorn "global" (el `.GlobalEnv`).

Feu `rm(list=ls())` per esborrar tot el que tenim en el `.GlobalEnv`, i executeu les instruccions següents:

```
x <- 3
y <- 5
quadrat <- function(z) { z^2 }
ls()
environment()
```

Hem llistat els objectes de l'entorn global. Apareixen `x`, `y`, `quadrat`, que són els que hem definit. No apareix `z`, que és un objecte intern de la funció `quadrat()`; no està en el nostre entorn global. La instrucció `environment()` ens diu que estem en l'entorn global (`R_GlobalEnv` és el mateix que `.GlobalEnv`)

La funció `quadrat()` crea un nou entorn, que és on viu `z`. És un entorn *fill* de `.GlobalEnv`; `.GlobalEnv` és el seu entorn *pare*. Torneu a eliminar tots els objectes de l'entorn global amb `rm(list=ls())`, i feu un script amb aquest codi:

```
f <- function(x){
  g <- function(y){
    print("Dins de g:")
    print(environment())
    print(ls())
    }
  g(1)
  print("Dins de f:")
  print(environment())
  print(ls())
  }
f(0)
print(environment())
print(ls())
```

Analitzem atentament el que hem fet:

- Al nivell més alt (entorn global), hem definit una funció `f`, que després hem cridat amb `f(0)`.
- Dins la funció `f`, hem definit una funció `g`, que després hem cridat amb `g(1)`.
- Dins de la funció `g`, hem escrit el nom del seu entorn i els objectes que hi ha en aquest entorn. Això és el primer que surt imprés a la sortida. El nom de l'entorn s'assigna automàticament i no és important. Només interessa que *no és* l'entorn global. Conté un sol objecte, que és el paràmetre `y`, amb valor 1, com podeu comprovar si afegiu la instrucció `print(y)` allà dins.
- Un cop hem retornat de `g`, continuem amb `f`. Escrivim el nom del seu entorn i els objectes que conté. Veiem que és un entorn diferent del de `g` i diferent del global, i que conté el paràmetre `x`, que val 0, i la funció `g`.
- Finalment, les dues últimes instruccions del programa escriuen l'entorn, que ara sí és el global, i l'únic objecte que conté, que és la funció `f`.

Deduïm que des de l'entorn global no podem cridar a la funció `g` ni accedir als objectes `x`, `y`.

54 § Un cop més esborreu tots els objectes de l'entorn global: `rm(list=ls())`. Com que estem esborrant `f`, també estem eliminant el seu entorn i en conseqüència també la funció `g` i el seu entorn. Feu ara:

```
f <- function(x){
  g <- function(y){
    c <- 30
    a+b+c
    print("Dins de g:")
    print(ls())
  }
  b <- 20
  a+b
  g(1)
  print("Dins de f:")
  print(ls())
}
a <- 10
f(0)
print(ls())
```

Observeu ara que, apart dels paràmetres `x` i `y` de les funcions, tenim unes variables `a`, `b`, `c` definides en diferents entorns.

La variable `a` està definida a l'entorn global, i és accessible des de dins de `f` i des de dins de `g`. Aquestes variables s'anomenen *globals* i estan accessibles a tot arreu en el programa des del moment en què estan definides.

Però cal tenir en compte que la globalitat d'una variable s'ha de pensar des del punt de vista de cada funció. Des del punt de vist de `g`, la variable `b` també és global, en el sentit que està definida en un entorn superior.

En canvi, des de l'entorn de la funció `f` no es pot accedir la variable `c`, i des de l'entorn global no es pot accedir a les variables `a` i `b`. Podeu provar-ho i veureu com surt un error. Aquestes variables són *locals*. Concretament, la variable `c` és local de l'entorn de `g`; la variable `b` és local de l'entorn de `f`.

55 § Què passa si hi ha noms de variables de diferents entorns que coincideixen? Després d'esborrar-ho tot amb `rm(list=ls())`, fem:

```
f <- function(){
  a <- 20
  g <- function(){
    a <- 30
    print(a)
  }
  g()
  print(a)
}
a <- 10
f()
print(a)
```

Després de l'assignació a nivell global `a <- 1`, entrem a la funció `f` i canviem el valor de `a`. Entrem a `g` i el tornem a canviar. No obstant, al tornar als entorns exteriors, la variable segueix tenint el valor que tenia abans. Concloem que:

- Una variable global és accessible, es pot usar, i es pot canviar, dins d'una funció.
- Al acabar la funció, la variable global no ha canviat.

La manera fàcil d'entendre-ho és pensar que l'entorn de la funció crea una còpia apart de la variable i treballa amb ella com a variable local. No toca la variable global.

56 § No és impossible canviar una variable global des de dins d'una funció. Cal usar l'operador de "superassignació" `<<-`. Esborrem tot altre cop i fem:

```
f <- function(){
  a <<- 20
  print(a)
}
a <- 10
f()
print(a)
```

Específicament, aquesta assignació amb fletxa de dues puntes el que fa és: Buscar la variable en el entorn pare; si no la troba, el busca en el següent nivell; i així successivament fins a l'entorn global. Si a l'entorn global la variable no existeix, la crea *en aquest entorn global.*

Exercici: Intenteu predir què passarà amb aquest codi

```
rm(list=ls())
f <- function(){
  g <- function(){
    a <<- 30
    print(a)
  }
  a <- 20
  g()
  print(a)
}
a <- 10
f()
print(a)
```

57 § Els informàtics aconsellen fortament no usar variables globals dins de les funcions, en cap llenguatge de programació. Hi ha bones raons. Una funció ben feta i reutilitzable hauria d'interaccionar amb el seu entorn pare només a través dels arguments que se li passen al cridar-la, i del valor que retorna. Per tant, no hauria d'utilitzar variables globals i menys encara modificar-les. No obstant, cal admetre que de vegades aquesta possibilitat ens permet acabar ràpidament una prova, que ja arreglarem després per tenir una funció ben polida i reutilitzable, si cal.

15 Precisió en els càlculs

58 § La funció `options()` permet veure i modificar algunes opcions molt bàsiques del funcionament del **R**. Per exemple, si no us agrada el símbol `>` que apareix a la consola

quan espera una instrucció (el *prompt*), el podeu canviar per alguna altra cosa. Proveu per exemple `options(prompt="Què vols? ")`.

L'opció més interessant de conèixer és `digits`, que per omissió té el valor 7. Això vol dir que els resultats dels càlculs surten amb set xifres significatives. Proveu:

```
x <- 10.87667748768532147
print(x)
```

Només ens dóna 7 xifres, arrodonint correctament. Però la resta no s'ha perdut:

```
options(digits=10)
print(x)
options(digits=15)
print(x)
```

Cada cop arrodoneix correctament a les xifres que se li donin (comptant part entera i part decimal). El valor màxim que podem posar és `digits=22`, però en aquest cas no ens donarà més de 17 xifres, i de fet la última pot estar mal arrodonida. Comproveu-ho.

El que succeeix és que el **R** emmagatzema els números en un espai limitat de memòria. Concretament, usa una *mantissa de 53 bits*[8], que correspon aproximadament a 16 xifres bones en base 10.

```
options(digits=20)
100/998
```

El **R** ens dóna el resultat amb 20 xifres, però les 3 últimes estan malament! Ho podeu comprovar amb el **SageMath**, que sí que fa el càlcul amb nombres enters amb total precisió, amb la instrucció `(100/998).n(digits=20)`.

Normalment, aquesta imprecisió del **R** és un problema molt menor. Un error a la dissetena xifra significativa vol dir un error relatiu de com a molt 10^{-16}, que sol ser negligible.

59 § De tota manera, pot haver-hi situacions en què vulguem treballar amb nombres el més exactes possible. Per exemple, suposem que volem saber de quantes maneres es pot ordenar un conjunt de 30 elements. Ho podem calcular amb la funció `factorial()`:

```
factorial(30)
```

El resultat és un nombre enter de 33 xifres. Però encara que hàgim posat el màxim `options(digits=22)`, només obtenim les 17 xifres més significatives, i la última segurament no és bona. En efecte, amb **SageMath** obtindreu `265252859812191058636308480000000`. L'última xifra del resultat del **R** està equivocada, i si s'ha d'arrodonir a 16 xifres, la setzena hauria de ser un 1, no un zero, cosa que no podem deduir de la sortida.

Anem a veure un altre exemple. Farem una funció que trobi les solucions reals de l'equació de segon grau $ax^2 + bx + c = 0$.

[8] també anomenada *mantissa de doble precisió*.

```
eqGrau2 <- function (a, b, c){
  discriminant <- b^2 - 4*a*c
  if(discriminant >= 0){
    radical <- sqrt(discriminant)
    print(c(-b - radical, -b + radical) / (2*a))
  } else {
    print("no hi ha solucions reals")
  }
}

a <- 1; b <- -5; c <- 6
eqGrau2(a, b, c)
```

El resultat és perfectament correcte. Les solucions exactes són $x = 2$, $x = 3$, i ens dóna el mateix tant si ho fem amb 7 dígits com amb 17 dígits.

Si dividim per 3 tots els coeficients de l'equació, el resultat hauria de ser el mateix. Però...

```
a <- 1/3; b <- -5/3; c <- 6/3
options(digits=7)
eqGrau2(a, b, c)
options(digits=17)
eqGrau2(a, b, c)
```

Ups! Ara el resultat és correcte quan demanem 7 dígits, i no ho és quan demanem 17. La setzena xifra ja està malament. Si feu `1000*eqGrau2(a, b, c)` i `100000*eqGrau2(a, b, c)`, veureu més coses estranyes.

Aquests resultats són conseqüència del fet que el **R** no pot guardar exactament números com $1/3$ o $-5/3$. Els guarda en un cert nombre de bits (zeros i uns, en sistema binari), arrodonint correctament, però no de manera exacte. No és possible representar 'un terç' en una quantitat finita de decimals. Quan després fem operacions amb aquests números, els errors de representació interna es van propagant, i a la llarga es poden acumular i ser molt més grans.

Un últim exemple, més dramàtic: Useu la funció `eqGrau2()` per resoldre l'equació $x^2 - 1000.001x + 1 = 0$. Les solucions exactes són $x = 0.001$ i $x = 1000$. Comproveu que surten moltes menys xifres bones. El quart decimal d'una de les solucions està completament malament.

L'explicació és que certs valors del coeficients (aquells pels quals b^2 és molt més gran que $4ac$), provoquen la resta de dos números molt propers, i a conseqüència d'això es perden moltes xifres significatives. S'ha d'evitar restar valors semblants. El programa que hem fet, usant la fórmula clàssica, no és un bon programa per resoldre qualsevol equació de segon grau.

60 § Si necessitem fer càlculs amb més precisió, tenim un paquet de **R**, anomenat `Rmpfr`, que ens ho permet. Està basat en la biblioteca *Multiple Precision Floating-Point Reliable Library* del llenguatge **C**, que permet fer càlculs amb precisió arbitrària.

Instal·leu i carregueu aquest paquet. Feu aleshores:

```
factorial(mpfr(30, prec=120))
```

el resultat ara és exacte. L'argument `prec` indica la precisió en bits amb què volem fer el càlcul. Recordeu que hem dit en el 58 § que en principi el **R** utilitza sempre 53 bits.

Com sabem quin és el valor de `prec` que hem de posar? Si `bits` és el que hem de posar, i `dígits` són les xifres significatives desitjades, les dues quantitats estan relacionades per la fórmula

$$2^{\texttt{bits}} = 10^{\texttt{dígits}}$$

d'on es dedueix que

$$\texttt{bits} = \texttt{dígits} \times \log_2(10)$$

En aquest cas concret, veiem de la resposta aproximada que necessitarem 33 dígits. Calculem

```
33*log2(10)
[1] 109.6236
```

i veiem que 120 bits és més que suficient.

Exercici: Amb una sola instrucció, feu que el **R** ens doni els factorials correctes i exactes dels 24 primers nombres naturals. Podeu usar `prec=120`, que ja hem vist que és suficient per aquestes quantitats.

Exercici: Calculeu de quantes maneres exactament poden quedar ordenades les 52 cartes d'una baralla, calculant primer un valor raonable de `prec`.

16 Rmarkdown

61 § Els resultats d'una anàlisi estadística no són només números i gràfics. El destinatari dels informes estadístics és freqüentment algú que ha de prendre decisions. Cal per tant *redactar*, desenvolupar la història que els resultats expliquen, de manera correcta i el més estètica possible.

Veurem que el LaTeX ens ajuda a produir fitxers de gran qualitat en format PDF, amb el nostre text barrejat amb les anàlisis, els resultats i els gràfics.

Però hi ha una manera més ràpida i fàcil de crear informes força presentables, en format HTML. Sense entrar en detalls de quins programes intervenen i com, a efectes pràctics el que farem és instal·lar i carregar un paquet que s'anomena `rmarkdown`.

Un cop instal·lat aquest paquet, podrem crear, en el **Rstudio** mateix, fitxers que barregen:

- el text que volem explicar,
- unes marques per assenyalar títols, apartats, llistes, fórmules, ...,
- codi en **R** que volem executar.

Veiem directament un exemple: En el menú del **Rstudio**, feu `File > New File > R Markdown...` . Veureu un diàleg amb opcions. Podeu deixar-lo tal com està i acceptar. Us crearà un nou fitxer amb una plantilla de document. Feu `File > Save As...` i guardeu-lo amb un nom acabat en `.Rmd`, com ara `test.Rmd`.

Ara premeu el desplegable `Knitr` en el panel de l'editor, i feu `Knit to HTML`. Se us obrirà una pestanya `Viewer`, amb el document processat. Aquest viewer ens està ensenyant un fitxer HTML, que podem obrir amb el nostre navegador tocant el botonet `Show in new window` de sobre el panel. Acabeu de produir un informe HTML senzill però força agradable de llegir.

Tornant al **Rstudio**, anem a examinar el codi del fitxer:

- El que hi ha entre els guions `---` és una capçalera (*header*), que conté títol, autor, data, i on es poden posar determinades opcions.
- Trossos (*chunks*) de codi **R** entre tres *backticks* ```.
- Títols d'apartats en línies que comencen amb `##`.
- La resta és text que surt tal qual en el HTML, però veiem que: L'adreça web que hi ha entre parèntesis angulars `<` i `>` s'ha tornat "activa"; la paraula emmarcada en doble estrelles `**` ha sortit en negreta; els caràcters entre backticks simples `` ` `` han quedat emmarcats en un fons diferent i en lletra diferent i monoespaiada.

Examinem ara el codi **R** entre triple backticks, començant pel segon:

````
```{r cars}
summary(cars)
```
````

A la primera línia hi ha, entre claus, la lletra `r`, que indica que el que segueix és codi **R**, i la paraula `cars`. Aquesta paraula és simplement una etiqueta per si més tard volem referir-nos a aquest *chunk*. La podeu eliminar i veureu que no passa res en aquest exemple. En canvi, si traieu la `{r}`, s'escriurà el codi com abans, però no s'avaluarà, i el resultat no surt.

Examinem ara l'últim chunk.

````
```{r pressure, echo=FALSE}
plot(pressure)
```
````

Després del l'etiqueta opcional `pressure`, i separades per comes, poden haver-hi algunes opcions, que s'apliquen només a aquest chunk. L'opció `echo=FALSE` impedeix que surti el codi, però l'execució es fa i el resultat es presenta.

En el primer chunk

````
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
````

s'estableixen unes opcions que seran vàlides per tots el chunks, si no es diu el contrari. L'opció `include=FALSE` d'aquest particular chunk indica que el **R** l'avaluarà, però no s'escriurà el resultat. Això és útil per executar codi preliminar, però que no interessa explicitar en l'informe.

Una altra opció habitual és `eval=FALSE`. El codi no s'executarà, però sí que es mostrarà si `include=TRUE`, que és el valor per omissió.

62 § Alguns recursos per profunditzar més en el `Rmarkdown`:

- Un exemple en castellà, una mica més complet que el que hem vist aquí:
  `http://www.unavarra.es/personal/tgoicoa/ESTADISTICA_RMarkdown_tomas/basicRmarkdown/index.html`
- Una guia de referència, lligada al **Rstudio**: `https://rmarkdown.rstudio.com/`
- Una "cheatsheet" de dues pàgines, per tenir a mà, i que es pot descarregar des del **Rstudio**: Aneu al menú `Help > Cheatsheets > R Markdown Cheat Sheet`.
- Una altra "cheatsheet" més senzilla:
  `https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf`
- I n'hi ha moltes més per internet ...

Índex