Cheat sheet

Conceptos básicos de programación

Estructuras de datos

Constructores	Expresiones	Comentarios

Secuencias	`n:m`
	`seq(from, to, by)`
	`rep`	

Vectores	`c(valor1, valor2)` `vector(type, length)`	`type` corresponde al tipo base. Por ejemplo, `logical` o `numeric`. Por ejemplo: `vector("logical", 2) == c(FALSE, FALSE)`
Matrices	`matrix(data, nrow, ncol)`	
Listas	`list()` `list(nombre_1 = objeto_1, nombre_2 = objeto_2 ...)`	

Vectores y listas

Vectores	Expresiones	Comentarios
Longitud	`length(v)`	
Resizing	`length(v) <- valor`	
Filtrar elementos	`x[condición]`	Donde `condición` devuelve un vector lógico indicando los índices. Ejemplo: `x[x > 3]`
Obtener índices de elementos que cumplen una condición	`which(condición sobre x)`	Da los índices para los cuales la condición es cierta. Ejemplos: `which(LETTERS == "R")` `which((1:12)%%2 == 0) # which are even?` `which(ll <- c(TRUE, FALSE, TRUE, NA, FALSE, FALSE, TRUE)) #> 1 3 7`
Índices ordenados	`order(x)`	Tiene muchos parámetros. Merece la pena echarle un ojo
Vector ordenado	`sort(x)`	
Atributos	`mode(x)` `typeof(x)`	
Eliminar elemento en una cierta posición	`x[-indice(s)]`	Ejemplo: `x[-c(2, 3)]` devuelve el vector sin las posiciones 2 y 3
Asignar un nombre a las componentes	`names(v) <- c("nombre1", "nombre2")`	
Acceso a elementos de una lista	`lista[[1]]` `lista$nombre_objeto` `lista[['nombre_objeto]]`	Hay una pequeña diferencia entre `lista[]` y `lista[[]]`. El objeto que se devuelve es diferente.

Vectores	Expresiones	Comentarios
Crear una nueva componente para una lista	`lista$componente <- objeto` `lista[[indice]] <- objeto`	Sí, no hace falta que reserves nada.

Matrices

Matrices	Expresiones	Comentarios
Multiplicación elemento a elemento	`*`	
Multiplicación matricial	`%*%`	
Traspuesta	`t(matriz)`	
Inversa	`solve(A)`	
Resolución de sistemas	`solve(A, b)`	En un sistema de ecuaciones, `A` es la parte de la izquierda y `b` la de la derecha
Suma de filas y columnas	`rowSums()`, `colSums()`	
Sacar las diagonales de una matriz	`split(matriz, col(matriz) -+ row(matriz)`	Dependiendo de si se usa `+` o `-`, da unas diagonales u otras
Diagonal de una matriz	`diag(matriz)`	
Unir filas o columnas	`rbind`, `cbind`	

Dataframes

Concepto	Expresiones	Comentarios
Constructor	`data.frame(columna1, columna2, ...)`	
Acceso a elementos	`df[v]` `df[v1, v2]`	Como lista: selecciona columnas Como matriz: selecciona elementos.

Filtrado	`df[condición]`
	`subset(df, condición)`
	`Filter(condición o función, df)`
		El argumento `condición` es una expresión lógica indicando lo que se quiere tomar. Ejemplo: `df[df$sexo == "Hombre"]`. Puedes combinarlos de varias formas. Por ejemplo, con rangos, como `df[1:2, c("edad", "estudios")]`
		La función `subset` también sirve para otro tipo de estructuras.
		`Filter` y `filter` producen resultados diferentes.

Primeros y últimos elementos	`head(df)` `tail(df)`	

Concepto	Expresiones	Comentarios
Transformaciones	`transform(df, expresión1, expresión2, ...)`	Añade las filas calculadas por las expresiones dadas. Ejemplos: `transform(mtcars, wt = wt/2` o `transform(mtcars, columna_nueva = wt/2)`
Añadir columnas	`within(df, z <- expr)`	Añade la columna de nombre `z` calculada por la expresión. Se pueden añadir varias columnas metiéndolo en un bloque
Acceder a un elemento aleatorio de un cierto subconjunto	`sub_df <- subset(df, condición sobre df)` seguido de `sub_df[sample(rownames(sub_df), número de muestras), ]`	Si quieres aplicarlo a todo el dataset, puedes obviar la primera parte

Factores

Concepto	Expresiones	Comentarios
Crear un factor	`v.factores <- factor(array de valores)`	Por ejemplo, `array de valores = c(1, 2, 3, 2, 1, 3, 2)`.
Niveles	`levels()`	Codifica enteros a strings. Ejemplo: `sexo.f <- factor(c(1,,2,2,1,2,1),labels=c('hombre','mujer'))`

Agregando por factores

Concepto	Expresiones	Comentarios
`split`	`split(contenedor, criterio)`	Un criterio típico suele ser clasificar por factor. Por ejemplo: `split(Orange, Orange$Tree)`. Se le puede pasar también una función
`tapply`	`tapply(contenedor, INDEX = criterio(s), función)`	Muy similar a aggregate. Hace un `split` de acuerdo con el/los factores y aplica la función a cada subconjunto.
`aggregate`	`aggregate(contenedor, by = criterio(s), función)`	Aplica funciones a subconjuntos de un df determinados por *una lista* de factores. Esa lista sí puede tener un sólo elemento

Operadores

Operadores	Expresiones	Comentarios
Ariméticas	`+ - * / ^`	
Módulo	`%%`	
Divisón entera	`%/%`	
Lógicas	`!`, `&`, `&&`, `|`, `||`, `xor(x,y)`	Los operadores dobles (`&&` y `||`) cortocircuitan. Los simples funcionan vectorialmente elemento a elemento. Mejor usa los dobles que no hacen cosas raras

Flujo de control

Concepto	Expresiones	Comentarios
Condicionales	`if (cond) {expr} else if (cond) {expr} else { expr}`	Son capaces de devolver valores al final de la expresión. Eso significa que se pueden hacer asignaciones del estilo `x <- if (expr) nosequé else nosecuánto`
Condicional vectorizado	`ifelse(condición, valor verdadero, valor falso)`	
Switch	`switch(variable, posible_valor1 = retorno, posible_valor2 = retorno)`	

Bucles

Concepto	Expresiones	Comentarios
Range-based for loop	`for (variable in rango) {}`	Habitualmente, `for (i in min:max)` o `for (x in contenedor)`
While	`while (cond) {expr}`	
Salida temprana	`break`	

Funciones

Concepto	Expresiones	Comentarios
Declaración de una función	`f <- funcion(parámetros) { expresión }`	`class(f) = "function"`
Comprobar argumentos de una función	`formals(función)`	
Cuerpo de una función	`body(función)`	Tema2_2 tiene información adicional sobre reflexión en R.
Lambdas/closures/funciones anónimas	`function(parametro) {expresión}`	En esencia, como la creación de una función normal y corriente pero sin asignárselo a un objeto. Por las ideosincracias de R, **se hace captura de todo el entorno por defecto**. Esto también se aplica a las funciones habituales
Número variable de argumentos en una función	`function(parámetros obligatorios, ...)`	
Argumentos omitidos	`missing(argumento)`	Es especialmente útil cuando se cita directamente los parámetros de una función (`f(x = 1, y = 2, w = 4) => missing(z)`)

La familia `apply` y otras funciones similares

Concepto	Expresiones	Comentarios
`lappy`	`lappy(contenedor, función a aplicar)`	Devuelve una lista. De ahí la `l`.
`sapply`	`sapply(contenedor, función a aplicar)`	Simplifica el retorno. Creo que devuelve un vector
`apply`	`apply(contenedor, MARGIN, función, ..., simplify=TRUE)`	`MARGIN` controla la dimensión a la que se le aplica la función. Más info
Productorio	`prod(x)` `cumprod(x)`	
Sumatoria	`sum(x)` `cumsum(x)`	

Lectura de ficheros

Concepto	Expresiones	Comentarios
Directorio de trabajo	`getwd()` `setwd("path/al/directorio")`	
Leer tabla o dataframe	`read.table()`	`as.is` o `stringsAsFactors` para leerlas como factores. Si el fichero incluye los nombres de las columnas, puedes hacer `read.table(..., header = TRUE)` Se especifica el separador con `sep = ","` `col.names` y `row.names` indican los nombres de las filas y las columnas
Leer cosas más genéricas	`scan()`	Se come todo lo que le eches. Puedes escribir en el script literalmente lo que va a leer. Por defecto separa por espacios en blanco: (`' '`). Para saltarse headers, se pasa el argumento `skip = 1`
Cargar datos de un paquete	`data(dataset, package="nombre_del_paquete")`	Por ejemplo, `data(coal, package="boot")` carga los datos del dataset `coal` del paquete `boot`
Escribir a ficheros	`write.table("path/al/archivo")` `write`	

Distribuciones estadísticas, aleatoriedad y otros conceptos similares

Concepto	Expresiones	Comentarios
Media	`mean(x)`	
Cuasivarianza	`var(x)`	Sí, es la cuasivarianza, y no la varianza (°—°〃)
Cuasi-desviación típica	`sd(x)`	
Distribuciones	`rnorm(n, mean = 0, sd = 1)` `runif(n, min, max)`	
Dibujar funciones	`plot`, `abline`, `contour`	

Redondeo	`trunc`
	`floor`
	`ceiling`	

Gamma	`gamma`	
Tomar muestras	`sample(contenedor, elementos, replace = TRUE/FALSE)`	

Logging y debugging

Concepto	Expresiones	Comentarios
Mostrar las variables definidas en el entorno	`ls()`	
Mensajes	`stop()`, `warning()`, `message()`	
Traza	Tras una ejecución fallida, `traceback()`	
Punto de ruptura *(breakpoint)*	`browser()`	Cuando entra en el ambiente de debug, con `get("var")` saca las variables locales (¿A veces? R hace cosas RaRas), y `Q` sale del debuggeo
Debug	`debug(función)`	Para salir, `undebug(función)`

Plotting

Orden de dibujado `plot` es "high level"
`abline`, `curve` etc. se le llaman "low level".
Las "low level" dibujan sobre un gráfico ya creado anteriormente por `plot`

Algunos argumentos de `plot`

Concepto	Expresiones	Comentarios
Lo básico	`plot(y)` `plot(x, y)`	En el primer caso, R usa "Index plotting": `x = 1:length(y)`
Estilo	`type` `col`	Tipo de gráfico (puntos `p`, líneas `l`, histograma `h`) Color

Anotaciones	`main`
	`sub`
	`xlab, ylab`
		Título
		Subtítulo
		Etiquetas de ejes

Otras funciones de plotting "high level"

Concepto	Expresiones	Comentarios
Curvas	`curve(función)`	
Formas	`symbols`	Dibujar polígonos, estrellas, círculos, etc.

Plotting "low level"

Dibujando encima de plots

Concepto	Expresiones	Comentarios
Puntos	`points(x,y, col, ...)`	Dibujar puntos (o formas) en las coordenadas indicadas.
Líneas	`lines(x,y, col, ...)`	Como points pero dibuja líneas entre las coordenadas.
Recta	`abline(h=, v=, a=, b=)`	Dibuja rectas, horizontales, verticales o dadas por su valor en 0 y pendiente. También admite como argumento un `lm`
Texto	`text(x, y, texto)`	Escribe el texto en las coordenadas indicadas.
Leyenda	`legend(x, y, texto, col, lty)`	`texto`, `col` y `lty` son vectores indicando cada componente de la leyenda.
Título	`title`	

Documentos

Configuración de vscode

1. Instalación de paquetes: `install.packages(c("markdown", "rmarkdown", "knitr"))`
2. Entorno de ejecución de los chunks. Por defecto es el del documento. Para cambiarlo a la raiz del entorno de trabajo (eso es, donde se suelen ejecutar los comandos de los scripts):
    1. En settings.json: `"r.rmarkdown.knit.defaults.knitWorkingDirectory": "workspace root"` para que al compilar desde la interfaz de vscode se haga desde root.

2. `opts_knit$set(root.dir = ".")` Hace que se ejecuten en root workspace al ejecutarse en la consola. NOTA: se debería hacer cada vez que se abre vscode

Resumen

Documento inicial	Documento intermedio	Expresiones	Comentarios
.rnw	.tex	`knit("ej.Rnw")`	
.rmd	.md	`knit("ej.rmd")`	
.rmd	.HTML	`render("ej2.rmd")`	
.ext1	.ext2	`render("ej2.ext1", output_file = "ej2.ext2")`	

Documento intermedio	Documento final	Expresiones	Comentarios
.md	.html	`render("ej.md")`	
.tex	.pdf	`tools::texi2pdf("ej.tex")`	
.HTML		`browseURL(url="ej2.HTML")`	

Distribuciones aleatorias

Sintaxis

A partir de la combinación de un prefijo indicando qué tipo de valores y un sufijo indicando la distribución se nombra a toda una familia de funciones de R.

Prefijo	`r`	`d`	`p`	`q`
Resultado	Muestra	Función de densidad	Función de distribución	Cuantiles

Sufijo	`unif`	`norm`	`pois`	`t`	`f`	`chisq`	`binom`	`geom`	`exp`	`gamma`	`weibull`
Distribución			Poisson			Chi cuadrado					

Otras movidas que no sé dónde poner ahora mismo

Concepto	Expresiones	Comentarios
Regla de reciclaje		En general, R siempre va a intentar aplicar lo que le pidas como buenamente pueda. Eso significa que si faltan elementos en un vector, o se le aplica una función escalar a un vector, se aplicará al resto de dimensiones volviendo a usar los valores existentes. Por ejemplo, `log(vector)` aplica el logaritmo a cada componente.
Concatenación extraña de cadenas	`paste(string, string, sep)`	Ejemplo: `paste(c("X", "Y"), 1:10, sep = "") == ["X1", "Y2", "X3"...]`
Familia de funciones `is.()`	`is.na()` `is.integer()` `is.numeric()`	
Casting	`as.()`	Ejemplos: `as.numeric(x)`, `as.factor(x)`
Comparación de números en coma flotante	`all.equal()`	Se salta los problemas de redondeo de números en coma flotante. Por ejemplo, `all.equal(0.3 - 0.2, 0.1) == TRUE`, `all.equal(sqrt(3)^2, 3) == TRUE`
Otros	`str` `summary`	

Snippets de código útiles

Rmarkdown

Celda para exportar un `.rmd` a `.html`:

```
# El espacio entre las comillas sobra!
# Si no lo pongo, se rompe el fichero markdown
``` `{r purl=FALSE, echo=FALSE, results=FALSE, message=FALSE, fig.show='hide', warning=FALSE, eval=FALSE}
rmarkdown::render("./path/a/archivo", "html_document")
``` `
```

Integración de Monte Carlo

Primero, definir la función a integrar. En este caso,

$$\int_0^1 \frac{1}{1+x^2} dx$$

```
f <- function(x) {
  1 / (1 + x^2)
}

curve(f, 0, 1)
```

Tomar muestras y calcular su esperanza:

```
N <- 3000
x <- runif(N)
f_x <- sapply(x, f)

mean(f_x)
```

Gráfico con el error:

```
valor_exacto <- pi / 4

# Calcular la media y su error
estimacion <- cumsum(f_x) / (1:N)
error <- sqrt(cumsum((f_x - estimacion)^2)) / (1:N)

plot(1:N, estimacion,
  type = "l",
  ylab = "Aproximación y límites del error (1 - alpha = 0.975)",
  xlab = "Número de simulaciones",
)
z <- qnorm(0.025, lower.tail = FALSE)
lines(estimacion - z * error, col = "blue", lwd = 2, lty = 3)
lines(estimacion + z * error, col = "blue", lwd = 2, lty = 3)
abline(h = valor_exacto, col = 2)
```

Método de la transformada inversa

Definir la función de distribución inversa que te deberían haber proporcionado. Tendrás que cambiar los parámetros de la función. En ete caso, se utiliza la distribución de Pareto:

```r
F_inversa <- function(xi, a, b) {
  b / (1 - xi)^(1 / a)
}
```

Tomar muestras y sacar los gráficos:

```r
N <- 1000

xi <- runif(N)
a <- 5
b <- 4

x <- F_inversa(xi, a, b)

hist(x, freq = FALSE, breaks = "FD", main = "Método de la inversa para la distribución de ...", ylim = c(0, 1.2))
lines(density(x), col = "blue")
```

Aplicar el test de Kolmogorov-Smirnov para comprobar cómo de bien se aproxima. Para ello, necesitarás la función de ditribución original:

```r
pdf_pareto <- function(x, a, b) {
  ifelse(x >= b,
    (a * b^a) / (x^(a + 1)),
    0
  )
}

distrib_pareto <- function(x, a, b) {
  ifelse(x >= b,
    1 - (b / x)^a,
    0
  )
}

ks.test(x, distrib_pareto, a = a, b = b)
```

Método de aceptación-rechazo

Queremos sacar muestras de una distribución con función de densidad p_X a partir de una con función de densidad p_Y. Para este ejemplo,

$$p_X(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}, \quad 0 \leq x \leq 1; a = 2, b = 6$$
$$p_Y(x) = 1, \quad 0 \leq x \leq 1$$

El algoritmo necesita un $M \in [1, \infty)$ tal que

$$p_X(x) \leq M p_Y(x) \quad \forall x \in \mathbb{R}$$

Así que, para empezar, define tus parámetros:

```r
a <- 2
b <- 6

p_X <- function(x) dbeta(x, shape1 = a, shape2 = b)
p_Y <- function(x) 1

```

Buscamos el M óptimo. En este caso, se puede tomar

$$M = \sup_x \frac{p_Y(x)}{p_X(x)} = \sup_x p_X(x)$$

```r
resultado <- optimize(
    #                    Te va a tocar cambiar esto
    #                vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
    f = function(x) { dbeta(x, shape1 = a, shape2 = b) },
    maximum = TRUE,
    interval = c(0, 1)
)

M <- resultado$objective
```

Enseñar la curva del algoritmo:

```r
#          Cambia este parámetro!
#     vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
curve(dbeta(x, shape1 = a, shape2 = b), 0, 1)

#        Igual aquí
#         vvvvvvvv
curve(M * dunif(x), 0, 1, add = TRUE, col = 2, lty = 2)
legend('right',
    legend = c('p_X(x)', 'p_Y * M'),
    col = c(1, 2), lty = c(1, 2), bty = 'n'
)
```

Tomar muestras:

```r
N <- 1000
x <- double(N)

valores_generados <- 0

for (i in 1:N) {
    xi <- runif(1)
    y <- runif(1)   # Depende de p_Y

    valores_generados <- valores_generados + 1

    while (xi > p_X(y) / (M * p_Y(y))) {
        # Seguir generando hasta que aceptemos uno
        xi <- runif(1)
        y <- runif(1)
        valores_generados <- valores_generados + 1
    }

    # Aceptar el valor
    x[i] <- y
}

valores_generados
```

Mostrar gráfica con el resultado:

```r
hist(x, freq = FALSE, breaks = 'FD', main = 'Método del rechazo')
lines(density(x), col = 'blue')

#             Cambiar
#     vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
curve(dbeta(x, shape1 = a, shape2 = b), add = TRUE, col = 2)
```