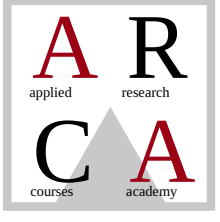
Introduzione a R

Programmazione in R



ARCA - @DPSS

Filippo Gambarota

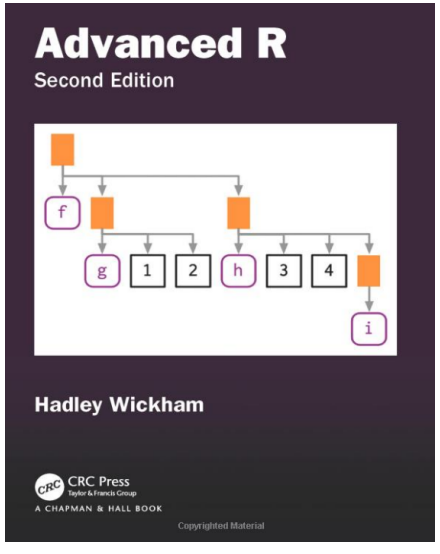
Programmazione in R

Quello che vedremo in questa sezione sono i principali **costrutti della programmazione** e la loro applicazione in R. Ci sono alcuni punti da considerare:

- Sono concetti trasversali estremamente utili
- Sono alla base di qualcunque **funzionalità già implementata in R**
- Vi permettono di fare qualunque cosa con il linguaggio

Programmazione in R - Disclaimer

Ci sono delle cose che per tempo e complessità non possiamo affrontare e che sono R specifiche. Per questi aspetti avanzati del linguaggio, il libro **Advanced R** è la cosa migliore



> https://adv-r.hadley.nz/

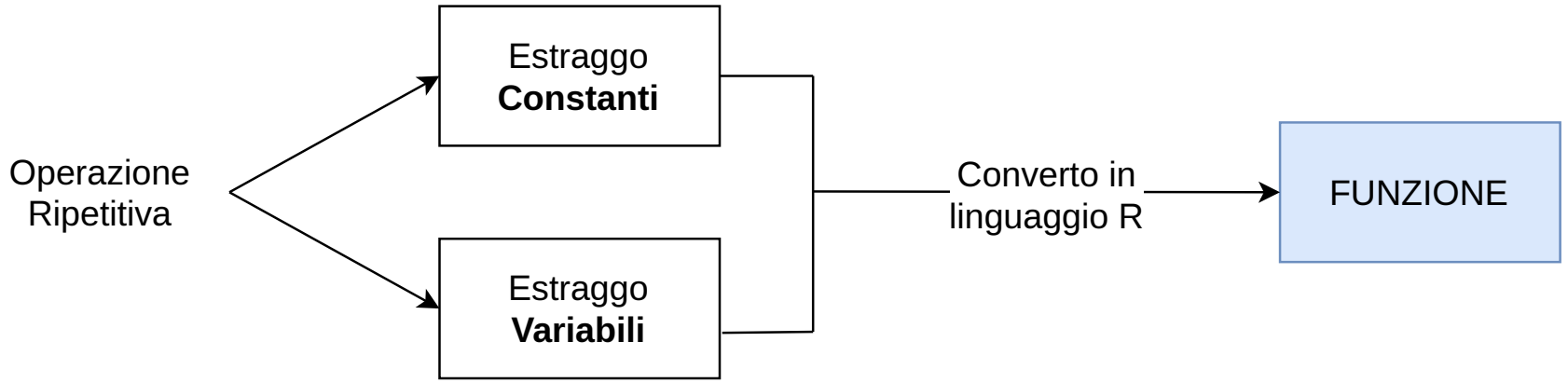
Costrutti della programmazione in R

Costrutti della programmazione in R

- **Funzioni**
- **Programmazione condizionale**
- **Programmazione iterativa**

Funzioni

Analogalmente alle *funzioni matematiche* la funzione in programmazione consiste nell' **astrarre** una serie di operazioni (nel nostro caso una porzione di codice) definendo una serie di operazioni che forniti degli *input* forniscono degli *output* eseguendo una serie di *operazioni*



Funzioni

Prendiamo un operazione ripetitiva che spesso si fa in analisi dati, **standardizzare** (trasformare in punti *z*) una variabile ovvero sottrarre da un *vettore* di osservazioni x la sua media μ_x e poi dividere per la deviazione standard σ_x:

$$x_z = \frac{x - \mu_x}{\sigma_x}$$

Seppur semplice, questa operazione può essere resa molto automatica scrivendo una funzione.

Funzioni

Se vogliamo *astrarre* questa operazione in modo da renderla più generale e utile dobbiamo definire:

- **argomenti funzione**: quelle che in matematica sono le *variabili*
- **corpo funzione**: le **operazioni** che la funzione deve eseguire usando gli argomenti
- **output funzione**: cosa la funzione deve **restituire** come risultato

Funzioni - Argomenti

Gli **argomenti** sono quelle parti variabili della funzione che vengono definiti e poi sono necessari ad eseguire la funzione stessa. Se vogliamo *astrarre* la retta che abbiamo visto prima dobbiamo definire alcune parti come **variabili**. Nel caso della nostra funzione l'unico argomento è il vettore x in input. Possiamo analogalmente a `mean` e `sd` impostare un argomento che indichi se eliminare gli `NA`:

```
z_score <- function(x, na.rm = FALSE){ # argomenti
    # body
    # output
}
```

Funzioni - Body

Il **corpo** della funzione sono le operazioni da eseguire utilizzando gli argomenti in input. Nel nostro caso dobbiamo sottrarre la *media* da x e dividere per la *deviazione standard*

```
z_score <- function(x, na.rm = FALSE){ # argomenti
    (x - mean(x, na.rm = na.rm)) / sd(x, na.rm = na.rm)
    # output
}
```

Funzioni - Output

L'output è il **risultato che la funzione ci restituisce** dopo aver eseguito tutte le operazioni. Nel nostro caso vogliamo che la funzione restituisca il vettore x ma trasformato in punti z:

```
z_score <- function(x, na.rm = FALSE){ # argomenti
    (x - mean(x, na.rm = na.rm)) / sd(x, na.rm = na.rm)
}
```

Per essere più consistenti possiamo usare il comando `return` che esplicitamente dice alla funzione cosa restituire:

```
z_score <- function(x, na.rm = FALSE){ # argomenti
    xcen <- (x - mean(x, na.rm = na.rm)) / sd(x, na.rm = na.rm) # assegno ad una nuova variabile nell'ambiente funzione
    return(xcen)
}
```

Funzioni - Risultato finale

Ora possiamo salvare la nostra funzione come un normale oggetto ed utilizzarla come se fosse una funzione giù implementata in R:

```
z_score <- function(x, na.rm = FALSE){ # argomenti
    xcen <- (x - mean(x, na.rm = na.rm)) / sd(x, na.rm = na.rm) # assegno ad una nuova variabile nell'ambiente funzione
    return(xcen)
}

vec <- rnorm(100, 50, 10) # media 50 e deviazione standard 10

mean(vec)
```

```
## [1] 51.16137
```

```
sd(vec)
```

```
## [1] 10.16701
```

```
vec0 <- z_score(vec)
mean(vec0)
```

```
## [1] -3.284363e-16
```

```
sd(vec0)
```

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

Programmazione condizionale

Programmazione condizionale

In programmazione solitamente è necessario non solo eseguire una serie di operazione **MA** eseguire delle operazione in funzione di alcune **condizioni**

Facciamo un esempio pratico, la funzione `summary()` in R fornisce un risultato diverso in base al tipo di input. Come è possibile tutto questo? Tramite l'utilizzo di **condizioni**:

```
x <- 1:10 # vettore numerico
y <- factor(rep(c("a", "b", "c"), each = 10)) # vettore di stringhe

summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    3.25    5.50    5.50    7.75   10.00
```

```
summary(y)
```

```
##  a  b  c
## 10 10 10
```

Programmazione condizionale

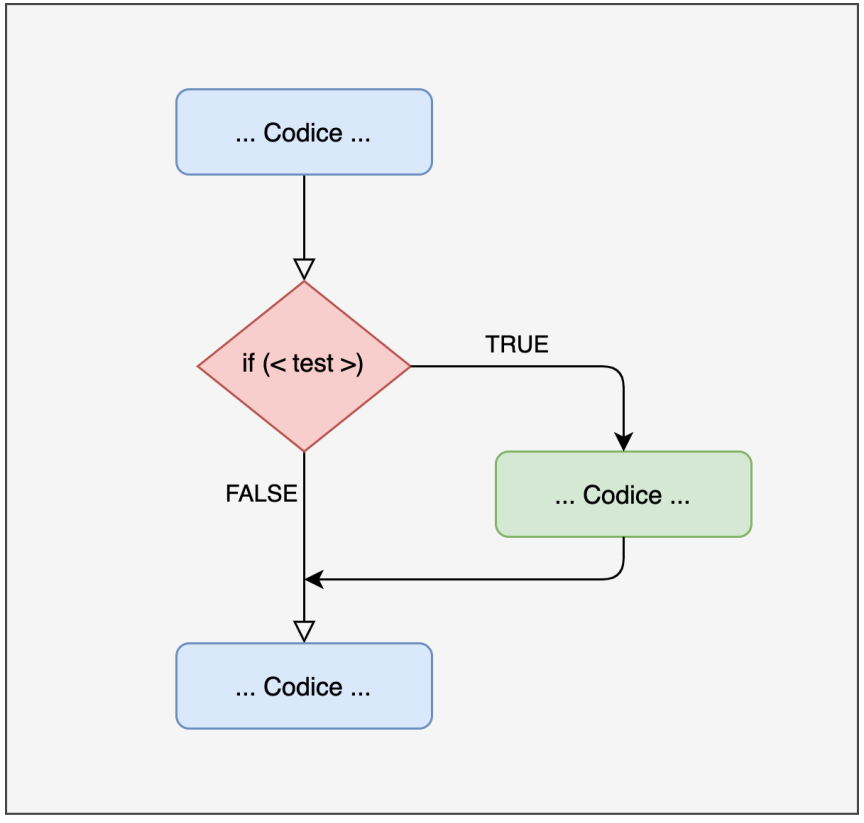
Anche se non sappiamo quali operazioni svolga la funzione `summary()` possiamo immaginare una cosa simile

```
summary <- function(argomento){

    # se l'argomento è un vettore numerico
    # esegui --> operazioni a,b,c

    # se l'argomento è un vettore stringa
    # esegui --> operazioni d,e,f

    # ...
}
```

Quindi non solo una funzione esegue lo stesso codice ogni volta che è chiamata ma può eseguire un codice specifico (o un parte) in base al contesto (condizioni)

Programmazione condizionale

Il concetto di `se <condizione> allora fai <operazione>` si traduce in programmazione tramite quelli che si chiamano `if statement`:



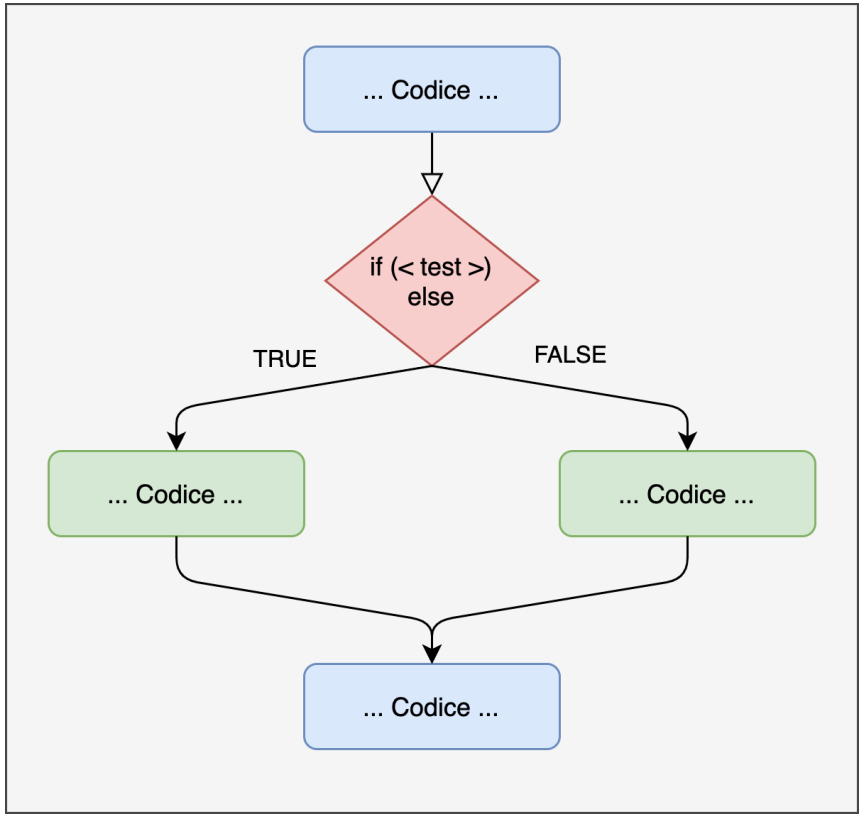
Programmazione condizionale

Per lavorare con gli `if statements` dobbiamo avere chiaro:

- il concetto di *operatori logici* ovvero `TRUE` e `FALSE`
- il concetto di *operazioni logiche* `TRUE and TRUE = TRUE`

Programmazione condizionale

Quando una sola condizione non basta...



Programmazione condizionale

Per poter capire quale struttura condizionale utilizzare è importante capire bene il problema che dobbiamo risolvere.

Ritornando all'esempio della funzione `summary()`, immaginiamo di avere 2 tipi di dati in R; stringhe e numeri.

In questo caso è sufficiente avere un `if statement` che controlla se l'elemento è una stringa/numero e per tutto il resto applicare l'opposto.

Programmazione condizionale - Tip

Esiste una famiglia di funzioni con prefisso `is.*` che fornisce `TRUE` quando la tipologia di oggetto corrisponde a quella richiesta e `FALSE` in caso contrario.

```
x <- 1:10

is.numeric(x)
```

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

```
is.factor(x)
```

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

```
is.character(x)
```

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

Possiamo usare queste funzioni per creare un flusso condizionale nella nostra funzione `summary()`

Programmazione condizionale

Scriviamo una funzione che restituisca la `media` quando il vettore è numerico e la tabella di frequenza (con la funzione `table()`)

```
my_summary <- function(x){

    # testiamo la condizione

    if(is.numeric(x)){
        return(mean(x))
    }else{
        return(table(x))
    }
}

x <- 1:10
my_summary(x)
```

```
## [1] 5.5
```

```
x <- rep(c("a","b","c"), c(10, 2, 8))
my_summary(x)
```

```
## x
##  a  b  c
## 10  2  8
```

`ifelse()`

Un limite di usare gli `if statements` riguarda il fatto che funzionano solo su un singolo valore (i.e. non sono **vettorizzati**):

```
x <- 1:10
if(x < 5){
    print("x è minore di 5")
}else{
    print("x è maggiore di 5")
}
```

```
## Error in if (x < 5) {: the condition has length > 1
```

La versione vettorizzata è la funzione `ifelse(test, yes, no)`:

```
ifelse(x < 5, "x è minore di 5", "x è maggiore di 5")
```

```
##  [1] "x è minore di 5"   "x è minore di 5"   "x è minore di 5"   "x è minore di 5"
##  [5] "x è maggiore di 5" "x è maggiore di 5" "x è maggiore di 5" "x è maggiore di 5"
##  [9] "x è maggiore di 5" "x è maggiore di 5"
```

`ifelse()`

Come anche per gli `if statements` normali, posso creare degli `ifelse() nested` quando ho bisogno di testare più alternative. Immaginiamo di avere una colonna/vettore `age` e voler creare un altro vettore dove l'età è divisa in 3 fascie, bambino, adulto, anziano:

```
age <- round(runif(50, 3, 80))
age_ifelse <- ifelse(age < 18,
       yes = "bambino",
       no = ifelse(
           age >= 18 & age < 60,
           "adulto",
           "anziano"
       ))
```

`dplyr::case_when()`

Quando le condizioni da testare sono numerose (indicativamente > 3) può essere tedioso scrivere molti `ifelse()` multipli. Possiamo allora usare la funzione `dplyr::case_when()` del pacchetto `dplyr` che è una generalizzazione di `ifelse()`:

```
age_case_when <- case_when(age < 18 ~ "bambino",
          age >= 18 & age < 60 ~ "adulto",
          TRUE ~ "anziano") # con TRUE si identifica "tutto il resto" in modo da non lasciare valori scoperti (ATTENZIONE)
```

I due risultati sono identici:

```
all.equal(age_case_when, age_ifelse)
```

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

Esempio con `dplyr::case_when()`

Ricodificare i valori di una variabile come ad esempio "girare" gli item di un questionario è un operazione facilmente eseguibile in con `dplyr::case_when()`:

```
item <- sample(1:5, 20, replace = TRUE) # simuliamo delle risposte ad un item
item
```

```
##  [1] 1 2 3 5 4 5 3 4 5 3 2 3 3 1 3 2 2 3 1 4
```

```
# ricodifichiamo con 1 = 5, 2 = 4, 3 = 3, 4 = 2, 5 = 1
item_rec <- case_when(
    item == 1 ~ 5,
    item == 2 ~ 4,
    item == 3 ~ 3,
    item == 4 ~ 2,
    item == 5 ~ 1
)

item_rec
```

```
##  [1] 5 4 3 1 2 1 3 2 1 3 4 3 3 5 3 4 4 3 5 2
```

Se usate spesso dei questionari potete scrivervi la vostra funzione che fa lo scoring in automatico 😏

Programmazione iterativa

Programmazione iterativa

Il concetto di *iterazione* è alla base di qualsiasi operazione nei linguaggi di programmazione.

In R molte delle operazioni sono **vettorizzate**. Questo rende il linguaggio più efficiente e pulito MA nasconde il concetto di **iterazione**. Ad esempio la funzione `sum()` permette di sommare un vettore di numeri. Ma cosa si nasconde sotto?

```
sum(1:100)
```

```
## [1] 5050
```

```
# come è possibile?
```

Programmazione iterativa

Esempio: se io vi chiedo di usare la funzione `print()` per scrivere `"hello world"` nella console 5 volte, come fate?

```
msg <- "Hello World"
print(msg) # 1
```

```
## [1] "Hello World"
```

```
print(msg) # 2
```

```
## [1] "Hello World"
```

```
print(msg) # 3
```

```
## [1] "Hello World"
```

```
print(msg) # 4
```

```
## [1] "Hello World"
```

```
print(msg) # 5
```

```
## [1] "Hello World"
```

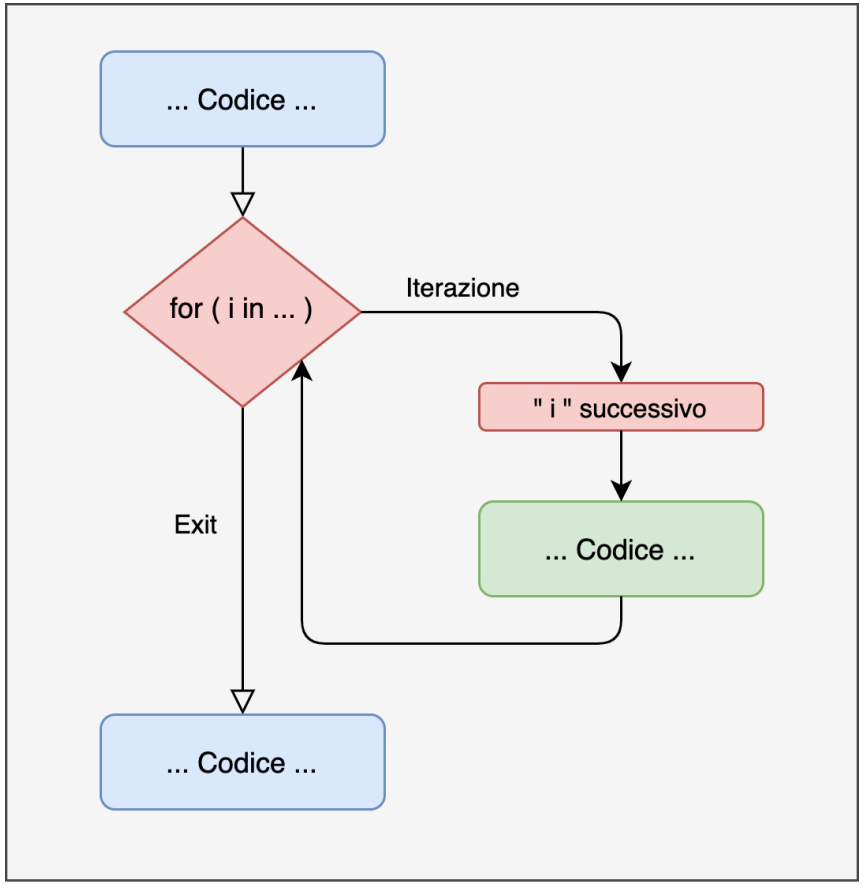
Programmazione iterativa

Quello che ci manca è un modo di ripetere una certa operazione, senza effettivamente ripetere il codice manualmente.

Ci sono vari costrutti che ci permettono di ripetere operazioni:

- Cicli `for`
- Cicli `while`
- `*apply` family
- altri

Il ciclo `for`



For

Il ciclo `for` è una struttura che permette di ripetere un numero *finito* e *pre-determinato* di volte una certa porzione di codice:

La scrittura di un ciclo `for` è:

```
for(i in 1:n){
    # quante operazioni voglio
}
```

Se voglio stampare una cosa 5 volte, posso tranquillamente usare un ciclo `for`:

```
for(i in 1:5){
    print(paste("Ciclo for giro", i))
}
```

```
## [1] "Ciclo for giro 1"
## [1] "Ciclo for giro 2"
## [1] "Ciclo for giro 3"
## [1] "Ciclo for giro 4"
## [1] "Ciclo for giro 5"
```

Scomponiamo il ciclo `for`

Ci sono diversi elementi:

- `for(){}`: è l'implementazione in R (in modo simile all'`if statement`)
- `i`: questo viene chiamato *iteratore* o *indice*. E' un indice generico che può assumere qualsiasi valore e nome. Per convenzione viene chiamato `i`, `j` etc. Questo tiene conto del numero di iterazioni che il nostro ciclo deve fare
- `in <valori>`: questo indica i valori che assumerà l'*iteratore* all'interno del ciclo
- `{ # operazioni }`: sono le operazioni che i ciclo deve eseguire

Ma l'iteratore?

La potenza del ciclo `for` sta nel fatto che l'iteratore `i` assume i valori del vettore specificato dopo `in`, uno alla volta:

```
for(i in 1:10){
    print(i)
}
```

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

For con iteratore vs senza

Questa è una distinzione importante quanto sottile, notate la differenza tra questi due cicli:

```
vec <- 1:5

for(i in 1:length(vec)){
    print(vec[i])
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

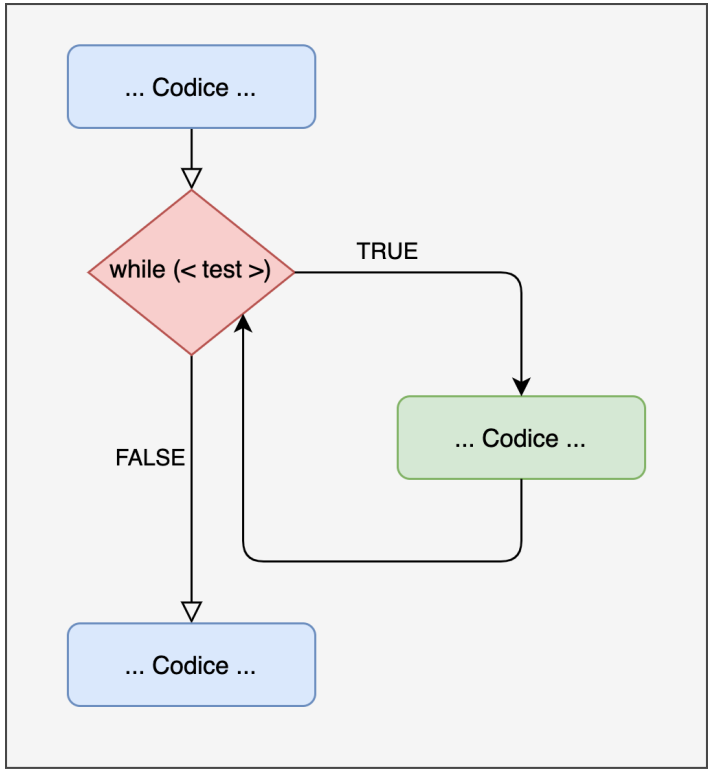
```
vec <- 1:5

for(i in vec){
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

While

Il ciclo `while` è una versione più generale del ciclo for. Per funzionare utilizza una *condizione logica* e non un iteratore e un range di valori come nel `for`. Il ciclo continuerà fino a che la `condizione` è vera:

```
while(condizione){
    # operazioni
}
```



While - (Fun 🤭)

Provate a scrivere questo ciclo `while` e vedere cosa succede e capire perchè accade.

```
x <- 10

while (x < 15) {
    print(x)
}
```

While

Questo esercizio è utile per capire che il `while` è un ciclo non pre-determinato e quindi necessita sempre di un modo per essere interrotto, facendo diventare la condizione falsa.

```
x <- 5

while (x < 15) {
    print(x)
    x <- x + 1
}
```

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
```

Applicazioni dei cicli

Gli esempi finora sono semplici ma poco utili. Quando il queste strutture iterative sono veramente utili?

Molte delle funzioni che utilizziamo come ad esempio `sum()`, `mean()`, etc. hanno al loro interno una sturttura iterativa

Immaginiamo di non avere la funzione `sum()` e di volerla ricreare, come facciamo? Idee?

Somma come iterazione

Scomponiamo concettualmente la somma, sommiamo i numeri da 1 a 10:

- prendo il primo e lo sommo al secondo (`somma = 1 + 2`)
- prendo la `somma` e la sommo al 3 elemento `somma = somma + 3`
- ...

In pratica abbiamo:

- il nostro vettore da sommare
- un oggetto `somma` che accumula progressivamente le somme precedenti

Somma come iterazione

```
somma <- 0 # inizializziamo la somma a 0
x <- 1:10

for(i in seq_along(x)){
    somma <- somma + x[i]
}
```

Somma come iterazione

Mettiamo tutto dentro una funzione

```
my_sum <- function(x){
    somma <- 0 # inizializziamo la somma a 0

    for(i in seq_along(x)){
        somma <- somma + x[i]
    }

    return(somma)
}

x <- rnorm(100)

my_sum(x)
```

```
## [1] -4.10236
```

```
sum(x)
```

```
## [1] -4.10236
```

Iterazione e funzioni

Per quanto sia un esercizio utile e divertente ricreare le funzioni base di R capendo la struttura iterativa (🤭) questo nella pratica non è quasi mai necessario.

Però è assolutamente fondamentale capire il **concetto** di iterazione perchè praticamente ogni operazione consiste nell'iterare tra:

- colonne/righe di un dataframe
- elementi di un vettore
- lettere in una parole
- ...

Ma in R c'è qualcosa di meglio...

Ma in R c'è qualcosa di meglio...

In R, l'utilizzo **esplicito** dei cicli `for` non è molto diffuso, per 2 motivi:

- R è un linguaggio fortemente **funzionale**
- R è un linguaggio spesso **vettorizzato**
- I cicli `for` sono molto verbosi e non sempre leggibili
- I cicli `for` in R, se non scritti bene, possono essere *estremamente lenti*

`*apply` family

`*apply` family

Immaginate di avere una `lista` di vettori, e di voler applicare la stessa funzione/i ad ogni elemento della lista. Come fare? ^[1]

- applico manualmente la funzione selezionando gli elementi
- ciclo `for` che itera sugli elementi della lista e applica la funzione/i
- ...

```
my_list <- list(
    vec1 <- rnorm(100),
    vec2 <- runif(100),
    vec3 <- rnorm(100),
    vec4 <- rnorm(100)
)
```

> Hadley Wickam - The joy of functional programming - link

`*apply` family

Applichiamo `media`, `mediana` e `deviazione standard`:

```
means <- vector(mode = "numeric", length = length(my_list))
medians <- vector(mode = "numeric", length = length(my_list))
stds <- vector(mode = "numeric", length = length(my_list))

for(i in 1:length(my_list)){
    means[i] <- mean(my_list[[i]])
    medians[i] <- median(my_list[[i]])
    stds[i] <- sd(my_list[[i]])
}
```

```
means
```

```
## [1] 0.08567786 0.50740656 0.07169040 0.05138925
```

```
medians
```

```
## [1] 0.05458076 0.52402518 0.03049500 0.06753916
```

```
stds
```

```
## [1] 1.077466 0.263920 1.096556 0.967408
```

`*apply` family

Funziona tutto! ma:

- il `for` è molto laborioso da scrivere gli indici sia per la lista che per il vettore che stiamo popolando
- dobbiamo *pre-allocare delle variabili* (per il motivo della velocità che dicevo)
- 8 righe di codice (per questo esempio semplice)

`*apply` family

In R è presente una famiglia di funzioni `*apply` come `lapply`, `sapply`, etc. che permettono di ottenere lo stesso risultato in modo più conciso, rapido e semplice:

```
means <- sapply(my_list, mean)
medians <- sapply(my_list, median)
stds <- sapply(my_list, sd)

means
```

```
## [1] 0.08567786 0.50740656 0.07169040 0.05138925
```

```
medians
```

```
## [1] 0.05458076 0.52402518 0.03049500 0.06753916
```

```
stds
```

```
## [1] 1.077466 0.263920 1.096556 0.967408
```

`*apply` family - Bonus

Prima di introdurre l'`*apply` family un piccolo bonus. Sfruttando il fatto che in R **tutto è un oggetto** possiamo scrivere in modo ancora più conciso:

```
my_funs <- list(median = median, mean = mean, sd = sd)

lapply(my_list, function(vec) sapply(my_funs, function(fun) fun(vec)))
```

```
## [[1]]
##     median       mean         sd
## 0.05458076 0.08567786 1.07746584
##
## [[2]]
##    median      mean        sd
## 0.5240252 0.5074066 0.2639200
##
## [[3]]
##    median      mean        sd
## 0.0304950 0.0716904 1.0965559
##
## [[4]]
##     median       mean         sd
## 0.06753916 0.05138925 0.96740805
```

Amazing! ora cerchiamo di dare un senso a queste righe di codice!

`*apply` family

```
apply(<lista>, <funzione>)
```

- cosa può essere la `lista`?
  - lista
  - dataframe
  - vettore
- cosa può essere la `funzione`?
  - funzione *base* o importata *pacchetto*
  - funzione *custom*
  - funzione *anonima*

`*apply` family - intuizione

Prima di analizzare l'`*apply` family, credo sia utile un ulteriore parallelismo con il ciclo `for` che abbiamo visto. `*apply` non è altro che un ciclo `for`, leggermente semplificato:

```
vec <- 1:5
for(i in vec){
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

```
vec <- 1:5
res <- sapply(vec, print)
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

`*apply` family - spoiler funzione anonima

Quindi come il ciclo `for` scritto come `i in vec` assegna al valore `i` un elemento per volta dell'oggetto `vec`, internamente le funzioni `*apply` prendono il primo elemento dell'oggetto in input (`lista`) e applicano direttamente la funzione che abbiamo scelto.

C'è un modo per rendere esplicito questo, anche nelle funzioni `*apply`:

```
vec <- 1:5
res <- sapply(vec, print)
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

```
vec <- 1:5
res <- sapply(vec, function(i) print(i))
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

`*apply` e funzioni custom

```
center_var <- function(x){
    x - mean(x)
}

my_list <- list(
    vec1 = runif(10),
    vec2 = runif(10),
    vec3 = runif(10)
)

lapply(my_list, center_var)
```

```
## $vec1
##  [1]  0.28977080  0.19674640 -0.07957851 -0.34161444  0.01743640 -0.28611180 -0.17932146
##  [8]  0.35095297  0.40233935 -0.37061970
##
## $vec2
##  [1]  0.3850815332 -0.2409087854  0.0168611695  0.2452769681 -0.0006029097 -0.3669795744
##  [7] -0.1183059149 -0.3885490469  0.3651035643  0.1030229962
##
## $vec3
##  [1] -0.1545120  0.1068950 -0.2107561  0.3474578 -0.3787741  0.1887162 -0.3333551
##  [8]  0.2352372 -0.1189795  0.3180705
```

`*apply` e funzioni anonime

Una funzione anonima è una funzione non salvata in un oggetto ma scritta per essere **eseguita direttamente**, all'interno di altre funzioni che lo permettono:

```
lapply(my_list, function(x) x - mean(x))
```

```
## $vec1
##  [1]  0.28977080  0.19674640 -0.07957851 -0.34161444  0.01743640 -0.28611180 -0.17932146
##  [8]  0.35095297  0.40233935 -0.37061970
##
## $vec2
##  [1]  0.3850815332 -0.2409087854  0.0168611695  0.2452769681 -0.0006029097 -0.3669795744
##  [7] -0.1183059149 -0.3885490469  0.3651035643  0.1030229962
##
## $vec3
##  [1] -0.1545120  0.1068950 -0.2107561  0.3474578 -0.3787741  0.1887162 -0.3333551
##  [8]  0.2352372 -0.1189795  0.3180705
```

Come per i cicli `for` (ricordo che `*apply` e `for` sono identici), `x` è solo un placeholder (analogo di `i`) e può essere qualsiasi lettera o nome

Tutte le tipologie di `*apply`

Vediamo tutti i tipi di `*apply` che ci sono. Alcuni sono più *utili* altri più *robusti* e altri ancora poco utilizzati:

- `lapply()`: la funzione di base
- `sapply()`: `simplified-apply`
- `tapply()`: poco utilizzata, utile con i *fattori*
- `apply()`: utile per i *dataframe/matrici*
- `mapply()`: versione multivariata, utilizza *più liste contemporaneamente*
- `vapply()`: utilizzata dentro le funzioni e pacchetti

`lapply`

`lapply` sta per list-apply e restituisce sempre una lista, applicando la funzione ad ogni elemento della lista in input:

```
res <- lapply(my_list, mean)
res
```

```
## $vec1
## [1] 0.5143858
##
## $vec2
## [1] 0.4212777
##
## $vec3
## [1] 0.580982
```

```
class(res)
```

```
## [1] "list"
```

`sapply`

`sapply` sta per simplified-apply e (cerca) di restituire una versione più semplice di una lista, applicando la funzione ad ogni elemento della lista in input:

```
res <- sapply(my_list, mean)
res
```

```
##      vec1      vec2      vec3
## 0.5143858 0.4212777 0.5809820
```

```
class(res)
```

```
## [1] "numeric"
```

`apply`

`apply` funziona in modo specifico per dataframe o matrici, applicando una funzione alle righe o alle colonne:

- `apply(dataframe, index, fun)`

```
# index 1 = riga, 2 = colonna
my_dataframe <- data.frame(my_list)
head(my_dataframe)
```

```
##        vec1       vec2      vec3
## 1 0.8041566 0.80635922 0.4264700
## 2 0.7111322 0.18036890 0.6878771
## 3 0.4348073 0.43813886 0.3702259
## 4 0.1727713 0.66655465 0.9284399
## 5 0.5318222 0.42067478 0.2022079
## 6 0.2282740 0.05429811 0.7696983
```

```
apply(my_dataframe, 1, mean)
```

```
##  [1] 0.6789953 0.5264594 0.4143907 0.5892553 0.3849016 0.3507568 0.2952210 0.571
##  [9] 0.7217030 0.5223731
```

```
apply(my_dataframe, 2, mean)
```

```
##      vec1      vec2      vec3
## 0.5143858 0.4212777 0.5809820
```

```
apply(my_dataframe, 2, center_var)
```

```
##              vec1          vec2       vec3
##  [1,]  0.28977080  0.3850815332 -0.1545120
##  [2,]  0.19674640 -0.2409087854  0.1068950
##  [3,] -0.07957851  0.0168611695 -0.2107561
##  [4,] -0.34161444  0.2452769681  0.3474578
##  [5,]  0.01743640 -0.0006029097 -0.3787741
##  [6,] -0.28611180 -0.3669795744  0.1887162
##  [7,] -0.17932146 -0.1183059149 -0.3333551
```

`tapply`

`tapply` permette di applicare una funzione ad un *vettore*, dividendo questo vettore in base ad una variabile categoriale:

- `tapply(dataframe, index, fun)`: dove `index` è un vettore di stringa o un fattore

```
vec <- rnorm(75)
index <- rep(c("a", "b", "c"), each = 25)

tapply(vec, index, mean)
```

```
##          a          b          c
## 0.10181643 0.02448205 0.20929689
```

`vapply`

`vapply` è una versione più *solida* delle precedenti dal punto di vista di programmazione. In pratica permette (e richiede) di specificare in anticipo la tipologia di dato che ci aspettiamo come risultato

`vapply(X = , FUN = , FUN.VALUE = ,... )`

```
vapply(my_list, FUN = mean, FUN.VALUE = numeric(length = 1))
```

```
##      vec1      vec2      vec3
## 0.5143858 0.4212777 0.5809820
```

- `my_list, FUN = mean`: è esattamente uguale a `sapply/lapply`
- `FUN.VALUE = numeric(length = 1)`: indica che ogni risultato è un singolo valore numerico

`mapply`

Questa è quella più complicata ma anche molto utile. Praticamente permette di gestire più liste contemporaneamente per scenari più complessi. Ad esempio vogliamo usare la funzione `rnorm()` e generare vettori con diverse **medie** e **deviazioni stardard** in combinazione.

```
medie <- list(10, 20, 30, 40)
stds <- list(1,2,3,4)
mapply(function(x, y) rnorm(n = 10, mean = x, sd = y), medie, stds, SIMPLIFY = FALSE)
```

```
## [[1]]
##  [1] 11.876308  9.788507  9.546771 10.372822  9.416643  9.511142 10.224284 10.449884
##  [9] 10.229523 10.236444
##
## [[2]]
##  [1] 16.45552 19.41010 20.96536 19.52329 19.19807 22.06641 20.57918 21.02376 20.46342
## [10] 21.41231
##
## [[3]]
##  [1] 27.41879 32.57708 28.13387 33.63300 33.01750 33.70376 34.75323 27.86814 30.90583
## [10] 34.07511
##
## [[4]]
##  [1] 35.66711 35.33272 44.02816 36.95099 39.30467 40.31875 43.22334 35.36285 45.33968
## [10] 47.26893
```

IMPORTANTE, tutte le liste incluse devono avere la stessa dimensione!

`mapply`

```
mapply(function(x, y) rnorm(n = 10, mean = x, sd = y), medie, stds, SIMPLIFY = FALSE)
```

- `function(...)`: è una funzione anonima come abbiamo visto prima che può avere *n* elementi
- `rnorm(n = 10, mean = x, sd = y)`: è l'effettiva funzione anonima dove abbiamo i placeholders `x` and `y`
- `medie, stds`: sono **in ordine** le liste corrispondenti ai placeholders indicati, quindi `x = medie` e `y = stds`.
- `SIMPLIFY = FALSE`: semplicemente dice di restituire una lista e non cercare (come `sapply`) di semplificare il risultato

`mapply` come `for`

Lo stesso risultato (in modo più verboso e credo meno intuitivo) si ottiene con un `for` usando più volte l'iteratore `i`:

```
medie <- list(10, 20, 30, 40)
stds <- list(1,2,3,4)

res <- vector(mode = "list", length = length(medie))

for(i in 1:length(medie)){
    res[[i]] <- rnorm(10, mean = medie[[i]], sd = stds[[i]])
}

res
```

```
## [[1]]
##  [1] 10.078558 10.651960 10.682713 11.024984 10.784553 10.456683  8.147545  9.373883
##  [9]  8.077296  9.487587
##
## [[2]]
##  [1] 20.70090 22.09031 20.45758 17.85203 14.85874 17.80830 21.15399 19.79748 19.72561
## [10] 19.44003
##
## [[3]]
##  [1] 28.46641 30.16772 28.49762 31.65286 26.27925 30.94106 29.72999 30.58860 29.53732
## [10] 28.60502
##
## [[4]]
##  [1] 43.04116 40.67211 36.36465 34.57080 37.83380 35.86571 41.26113 44.69153 44.47569
## [10] 35.62735
```

`*apply` alcune precisazioni

`*apply` vettore vs lista

Abbiamo sempre usato esplicitamente `liste` fino ad ora, ma le funzioni `*apply` sono direttamente applicabili anche a **vettori**

- se usiamo un vettore di *n* elementi, allora itereremo da `1:n`
- se usiamo una lista di *n* elementi, allora iteriamo da `1:n` dove il singolo elemento può essere qualsiasi cosa

```
my_vec <- 1:5
my_list <- list(a = 1:2, b = 3:4, c = 5:6)
res <- sapply(my_vec, print)
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

```
res <- sapply(my_list, print)
```

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

`*apply` come un `for`

Nulla ci vieta (ma perdiamo l'aspetto intuitivo e conciso) di usare le funzioni `*apply` esattamente come un ciclo `for`, usando un **iteratore**:

```
medie <- c(10, 20, 30, 40)
stds <- c(1,2,3,4)

res <- lapply(1:length(medie), function(i){
    rnorm(n = 10, mean = medie[i], sd = stds[i])
})
```

Trovo tuttavia più chiara l'alternativa usando `mapply`:

```
mapply(function(x, y) rnorm(n = 10, mean = x, sd = y), medie, stds, SIMPLIFY = FALSE)
```

Extra: `purrr::map*`

Extra: `purrr::map*`

Senza addentrarci troppo in questo modo, c'è una famiglia di funzioni che una volta imparato `*apply` vi consiglio di usare perchè più consistenti e intuitive, la `map*` family.

Extra: `purrr::map*`

Per usare `purrr::map*` è sufficiente installare il pacchetto `purrr` con `install.packages("purrr")` ed iniziare ad usare le nuove funzioni. La sintassi è esattamente la stessa di `*apply` (qualche modifica ma potete usare la stessa) ma invece che usare una funzione per tutto, abbiamo molte funzioni per ogni casistica:

- `map(lista, funzione)` è l'analogo di `lapply()` e fornisce sempre una lista
- `map_dbl(lista, funzione)` applica la funzione ad ogni elemento e **si aspetta che** il risultato sia un vettore di *double*
- `map_lgl(lista, funzione)` applica la funzione ad ogni elemento e **si aspetta che** il risultato sia un vettore *logico*
- `map2/pmap_*` sono rispettivamente applicare la funzione a 2/n liste (analogo di `mapply()`)

Extra: `replicate()` and `repeat()`

Extra: `replicate()` and `repeat()`

Ci sono altre due funzioni in R che permettono di *iterare*. Sono meno utilizzate perchè si ottengono gli stessi risultati usando un semplice `for` o `*apply`.

- `replicate()` permette di ripetere un operazione *n* volte, senza però utilizzare un `iteratore` o un `placeholder`.

- `repeat()` anche repeat permette di ripetere ma fino a che non si verifica un certa condizione (**logica**). Ha una struttura simile al ciclo `while`

Extra: Formula syntax

Formula syntax

In R molte operazioni vengono eseguite usando la **formula syntax** `something ~ something else` ad esempio:

- modelli statistici: `lm(y ~ x, data = data)`, `t.test(y ~ factor, data = data)`
- plot: `boxplot(y ~ x, data = data)`
- ...

In cosa consiste?

Formula syntax

Senza andare nei dettagli tecnici, R usa una cosa che si chiama *lazy evaluation*. In altri termini "salva" delle operazioni per essere eseguite in un secondo momento. Tutti sappiamo che se scriviamo un nome (senza virgolette) e questo non è associato ad un oggetto otteniamo un errore. Tuttavia alcune funzioni come `library()` non forniscono errore. Perchè?

```
stats # errore
```

```
## Error in eval(expr, envir, enclos): object 'stats' not found
```

```
library(stats) # no errore
```

Formula syntax

La ragione è che R è in grado di salvare un'espressione per usarla poi in uno specifico contesto (ad esempio dentro una funzione). La `formula syntax` è un esempio. Usando la tilde `~` possiamo creare delle `formule` che R può utilizzare in specifici contesti:

```
head(y)
```

```
## [1] a a a a a a
## Levels: a b c
```

```
head(x)
```

```
## [1] -0.4859233 -0.4455036 -2.5433677 -1.5042388 -0.8073159 -0.5685927
```

```
y ~ x
```

```
## y ~ x
## <environment: 0x55a9fbeda540>
```

```
my_formula <- y ~ x
class(my_formula)
```

```
## [1] "formula"
```

Formula syntax e `aggregate()`

Un esempio utile è la funzione `aggregate()` molto interessante per applicare funzioni a dataframe. Immaginate di avere il dataset `iris` e calcolare la media per ogni livello del fattore `Species`:

```
tapply(iris$Sepal.Length, iris$Species)
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [42] 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [83] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
## [124] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

```
aggregate(Sepal.Length ~ Species, FUN = mean, data = iris)
```

```
##      Species Sepal.Length
## 1     setosa        5.006
## 2 versicolor        5.936
## 3  virginica        6.588
```

```
# Anche creando un oggetto, ma solo come formula

my_formula <- Sepal.Length ~ Species
my_char <- "Sepal.Length ~ Species"
aggregate(my_char, FUN = mean, data = iris)
```

```
## Error in inherits(by, "formula"): argument "by" is missing, with no default
```

```
# Viene lo stesso usando $ e senza specificare data =
aggregate(iris$Sepal.Length, iris$Species, FUN = mean)
```

```
## Error in aggregate.data.frame(as.data.frame(x), ...): 'by' must be a list
```

Formula syntax e `aggregate()`

Ma anche operazioni più complesse:

```
my_iris <- iris
my_iris$fac <- rep(c("a", "b", "c"), 50)
aggregate(Sepal.Length ~ Species + fac, mean, data = my_iris)
```

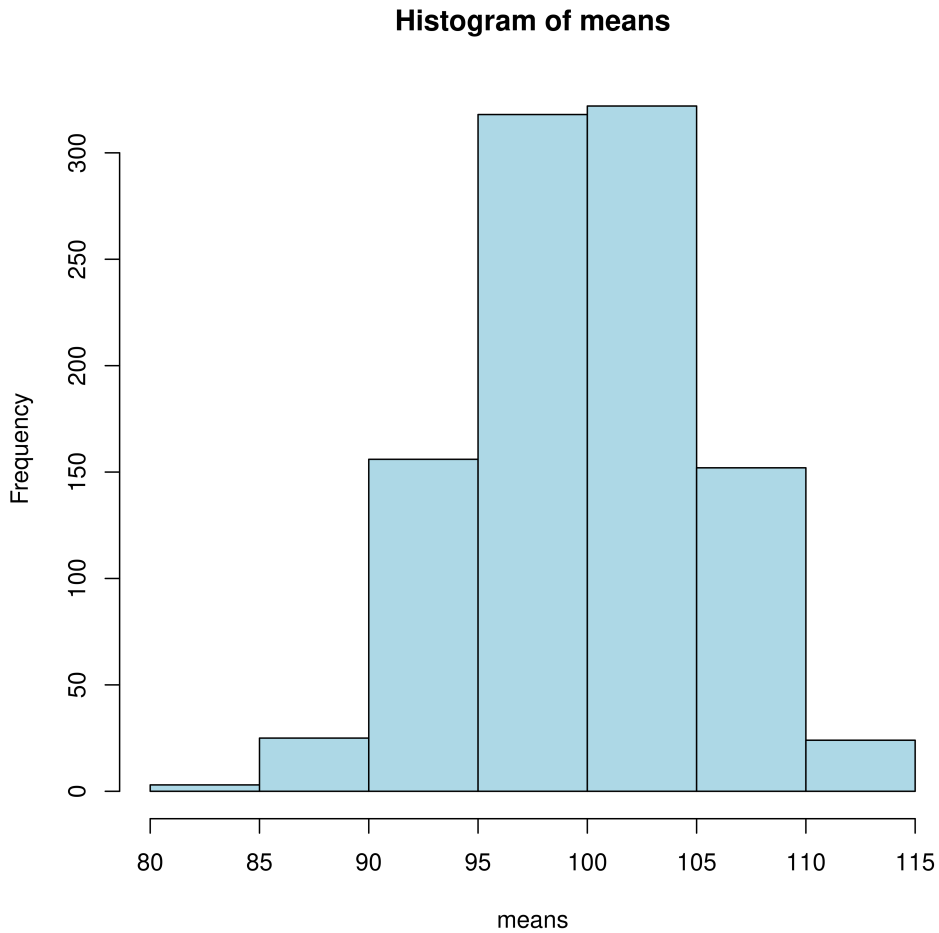
```
##      Species fac Sepal.Length
## 1     setosa   a     5.052941
## 2 versicolor   a     5.770588
## 3  virginica   a     6.756250
## 4     setosa   b     5.011765
## 5 versicolor   b     6.018750
## 6  virginica   b     6.447059
## 7     setosa   c     4.950000
## 8 versicolor   c     6.023529
## 9  virginica   c     6.570588
```

Replicate

`replicate(n, expr)`

- `n` è il numero di ripetizioni
- `expr` è la porzione di codice da ripetere

```r
# Campioniamo 1000 volte da una normale e facciamo la media AKA distribuzione camp

nrep <- 1000
nsample <- 30
media <- 100
ds <- 30

means <- replicate(n = nrep, expr = {
    mean(rnorm(nsample, media, ds))
})
```



repeat()

```
repeat {
    # cose da ripetere

    if(...){ # condizione da valutare

        break # ferma il loop
    }
}
```

```
i <- 1

repeat {
    print(i)
    i = i + 1
    if(i > 3){
        break
    }
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

`repeat()` vs `while`

```
i <- 1

repeat {
    print(i)
    i = i + 1
    if(i > 3){
        break
    }
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

```
i <- 1

while(i < 4){
    print(i)
    i <- i + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

- `repeat` valuta la condizione una volta finita l'iterazione, mentre `while` all'inizio. Se la condizione non è `TRUE` all'inizio, il `while` non parte mentre `repeat` si.

Dataframe come Liste

Dataframe come Liste

Essendo il dataframe tecnicamente una lista, è possibile eseguire delle operazioni iterative. Ad esempio:

```
sapply(mtcars, mean)
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec         vs
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750   0.437500
##         am       gear       carb
##   0.406250   3.687500   2.812500
```

Applica a tutti gli elementi della lista i.e. colonne la funzione mean

Dataframe come Liste

Possiamo però anche dividere un dataframe in liste di dataframes in base alle righe. Ad esempio possiamo voler fittare un modello statistico su ogni soggetto separatamente. Prendiamo questo dataframe di esempio con 2 condizioni, 30 trial in ogni condizione e 10 soggetti:

```
dat <- expand.grid(
    id = 1:10,
    cond = c("a", "b"),
    ntrial = 1:30
)
dat$y <- rnorm(nrow(dat))
head(dat)
```

```
##   id cond ntrial          y
## 1  1    a      1 -0.05539815
## 2  2    a      1 -1.14623219
## 3  3    a      1  0.91120638
## 4  4    a      1  0.44500075
## 5  5    a      1 -0.87490481
## 6  6    a      1  0.12107677
```

Dataframe come Liste

L'idea è quindi di calcolare un `t.test()` tra le condizioni separatamente per ogni soggetto. Possiamo splittare il dataframe per soggetto ottenendo una lista con 10 dataframes e poi applicare la funzione `t.test()` ad ogni elemento.

```
# definisco la funzione con tutti gli argomenti
ttest <- function(data){
    t.test(y ~ cond, data = data, paired = TRUE)
}

dat_list <- split(dat, dat$id) # splittiamo per id
length(dat_list)
```

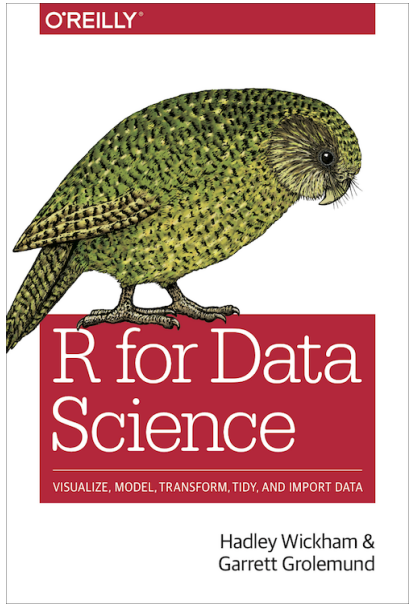
```
## [1] 10
```

```
t_list <- lapply(dat_list, ttest)
t_list[[1]]
```

```
##
##     Paired t-test
##
## data:  y by cond
## t = 1.8637, df = 29, p-value = 0.07252
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  -0.04738855  1.02015269
## sample estimates:
## mean difference
##       0.4863821
```

Dataframe come Liste (extra)

Questo approccio è la base per lavorare in modo molto compatto anche per fare cose complesse con più dataframe insieme. Basta avere chiaro il concetto di `funzione` e di `iterazione`. Il capitolo Many models di R4DS illustra molto chiaramente questa idea introducendo il concetto di nested dataframe.



.pull-right[

```
nestdat <- tibble::tibble(
    id = 1:10,
    data = dat_list
)

nestdat
```

```
## [90m# A tibble: 10 × 2[39m
##       id data
##    [3m[90m<int>[39m[23m [3m[90m<named list>[39m[23m
## [90m 1[39m     1 [90m<df [60 × 4]>[39m
## [90m 2[39m     2 [90m<df [60 × 4]>[39m
## [90m 3[39m     3 [90m<df [60 × 4]>[39m
## [90m 4[39m     4 [90m<df [60 × 4]>[39m
## [90m 5[39m     5 [90m<df [60 × 4]>[39m
## [90m 6[39m     6 [90m<df [60 × 4]>[39m
## [90m 7[39m     7 [90m<df [60 × 4]>[39m
## [90m 8[39m     8 [90m<df [60 × 4]>[39m
## [90m 9[39m     9 [90m<df [60 × 4]>[39m
## [90m10[39m    10 [90m<df [60 × 4]>[39m
```


1