Informatica

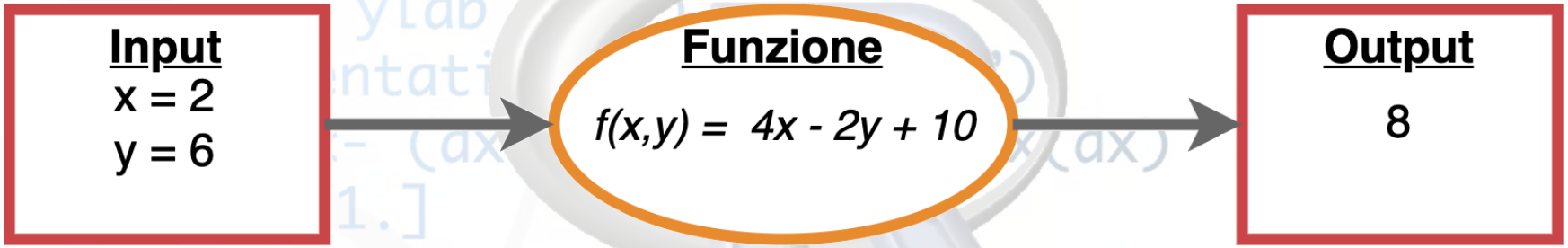
Principi di programmazione ed analisi dati in linguaggio

Scienze Biologiche

Dr. Bruno Bellisario, PhD

Le funzioni in R

Possiamo pensare alle funzioni in R in modo analogo alle classiche funzioni matematiche. Dati dei valori in input, le funzioni eseguono specifici calcoli e restituiscono in output il risultato ottenuto.



Le funzioni in R

Creazione di una funzione

Il comando usato per creare una funzione in R è `function()` seguito da una coppia di parentesi graffe `{ }` al cui interno deve essere specificato il corpo della funzione:

```
nome_funzione <- function( ){
  <corpo-funzione>
}
```

Nota come sia necessario assegnare la funzione ad un oggetto, ad esempio `my_function`, che diventerà il nome della nostra funzione. Per eseguire la funzione sarà sufficiente, come per ogni altra funzione, indicare il nome della funzione seguito dalle parentesi tonde, nel nostro caso `my_function()`.

Le funzioni in R

Creazione di una funzione

Vediamo alcuni esempi di semplici funzioni:

```
# Definisco la funzione
my_greetings <- function(){
  print("Hello World!")
}

my_greetings()
## [1] "Hello World!"

# Definisco un'altra funzione
my_sum <- function(){
  x <- 7
  y <- 3
  x + y
}

my_sum()
## [1] 10
```

Quando chiamiamo la nostra funzione, R eseguirà il corpo della funzione e ci restituirà il risultato dell'ultimo comando eseguito. Le funzioni del precedente esempio, tuttavia, si rivelano poco utili poiché eseguono sempre le stesse operazioni senza che ci sia permesso di specificare gli input delle funzioni. Inoltre, sebbene siano funzioni molto semplici, potrebbe non risultare chiaro quale sia effettivamente l'output restituito dalla funzione.

Le funzioni in R

Creazione di una funzione

Definire input ed output

Ricordiamo che in generale le funzioni, ricevuti degli oggetti in input, compiono determinate azioni e restituiscono dei nuovi oggetti in output.

Input e output sono quindi due aspetti fondamentali di ogni funzione che richiedono particolare attenzione.

Input – vengono specificati attraverso gli argomenti di una funzione. Per definire gli argomenti di una funzione, questi devono essere indicati all'interno delle parentesi tonde al momento della creazione della funzione
Output – Per specificare l'output di una funzione si utilizza la funzione return(), indicando tra le parentesi il nome dell'oggetto che si desidera restituire come risultato della funzione.

La definizione di una funzione con tutti i suoi elementi seguirà quindi il seguente schema:

```
nome_funzione <- function(argument_1, argument_2, ...){
  <corpo-funzione>
  return(<nome-output>)
}
```

Le funzioni in R

Creazione di una funzione

Definire input ed output

Possiamo ora riscrivere le precedenti funzioni permettendo di personalizzare gli input e evidenziando quale sia output che viene restituito.

```
# Ridefinisco my_greetings()
my_greetings <- function(name){
  greetings <- paste0("Hello ", name, "!")
  return(greetings)
}
my_greetings(name = "Bruno")
## [1] "Hello Bruno!"

# Ridefinisco my_sum()
my_sum <- function(x, y){
  result <- x + y
  return(result)
}
my_sum(x = 8, y = 6)
## [1] 14
```

Le funzioni in R

Lavorare con le funzioni

Le funzioni sono sicuramente l'aspetto più utile e avanzato del linguaggio R e in generale dei linguaggi di programmazione.

I pacchetti che sono sviluppati in R non sono altro che insieme di funzioni che lavorano assieme per uno scopo preciso.

Oltre allo scopo della funzione è importante capire come gestire gli errori e gli imprevisti. Se la funzione infatti accetta degli argomenti, l'utente finale o noi stessi che la utilizziamo possiamo erroneamente usarla nel modo sbagliato.

E' importante quindi capire come leggere gli errori ma sopratutto creare messaggi di errore o di avvertimento utili per l'utilizzo della funzione.

Le funzioni in R

Lavorare con le funzioni

Prendiamo ad esempio la funzione somma +, anche se non sembra infatti l'operatore + è in realtà una funzione. Se volessimo scriverlo come una funzione simile a quelle viste in precedenza possiamo:

```
my_sum <- function(x, y){
  res <- x + y
  return(res)
}
my_sum(1, 5)
## [1] 6
```

Abbiamo definito una (abbastanza inutile) funzione per calcolare la somma tra due numeri. Cosa succede se proviamo a sommare un numero con una stringa? Ovviamente è un'operazione che non ha senso e ci aspettiamo un qualche tipo di errore:

```
my_sum("pino", 5)
## Error in x + y: non-numeric argument to binary operator
```

Le funzioni in R

Ambiente delle funzioni

Il concetto di ambiente in R è abbastanza complesso.

In parole semplici, tutte le operazioni che normalmente eseguiamo nella console o in uno script, avvengono in quello che si chiama global environment.

Quando scriviamo ed eseguiamo una funzione, stiamo creando un oggetto funzione (nel global environment) che a sua volta crea un ambiente interno per eseguire le operazioni previste.

Immaginiamo di avere questa funzione my_fun() che riceve un valore x e lo somma ad un valore y che non è un argomento:

```
my_fun <- function(x){
  return(x + y)
}
my_fun(10)
## Error in my_fun(10): object 'y' not found
```

Le funzioni in R

Ambiente delle funzioni

```
my_fun <- function(x){
  return(x + y)
}
my_fun(10)
## Error in my_fun(10): object 'y' not found
```

Chiaramente otteniamo un errore perché l'oggetto y non è stato creato. Se però creiamo l'oggetto y all'interno della funzione, questa esegue regolarmente la somma MA non crea l'oggetto y nell'ambiente globale.

```
my_fun <- function(x){
  y <- 1
  return(x + y)
}
my_fun(10)
```

Le funzioni in R

Ambiente delle funzioni

Altra cosa importante, sopratutto per gestire effetti collaterali riguarda il fatto che la funzione NON modifica gli oggetti presenti nell'ambiente globale:

```
y <- 10 # ambiente globale
my_fun <- function(x){
  y <- 1 # ambiente funzione
  return(x + y) # questo si basa su y funzione
}
my_fun(1)
## [1] 2
y
## [1] 10
```

Come vedete, abbiamo creato un oggetto y dentro la funzione.

Se eseguito nello stesso ambiente questo avrebbe sovrascritto il precedente valore.

Il risultato si basa sul valore di y creato nell'ambiente funzione e l'y globale non è stato modificato.

Le funzioni in R

Ambiente delle funzioni

Un ultimo punto importante riguarda invece il legame tra ambiente funzione e quello globale.
Abbiamo visto la loro indipendenza che però non è totale.
Se infatti all'interno della funzione utilizziamo una variabile definita solamente nell'ambiente globale, la funzione in automatico userà quel valore (se non specificato internamente).
Questo è utile per far lavorare funzioni e variabili globali MA è sempre preferibile creare un ambiente funzione indipendente e fornire come **argomenti** tutte gli oggetti necessari.

```
y <- 10
my_fun <- function(x){
  return(x + y) # viene utilizzato y globale
}
```

Le cose importanti da ricordare quando si definiscono e utilizzano funzioni sono:
1. Ogni volta che una funzione viene eseguita, l'ambiente interno viene ricreato e quindi è come ripartire da zero
2. Gli oggetti creati all'interno della funzione hanno priorità rispetto a quelli nell'ambiente esterno
3. Se la funzione utilizza un oggetto non definito internamente, automaticamente cercherà nell'ambiente principale

Le funzioni in R

Best practice

Scrivere funzioni è sicuramente l'aspetto più importante quando si scrive del codice. Permette di automatizzare operazioni, ridurre la quantità di codice, rendere più chiaro il nostro script e riutilizzare una certa porzione di codice in altri contesti.
Ci sono tuttavia delle convenzioni e degli accorgimenti per scrivere delle ottime e versatili funzioni:

1. Quando serve una funzione?

2. Scegliere il nome

3. Semplificare la quantità di operazioni e output

4. Commentare e documentare

Le funzioni in R

Best practice

Quando serve una funzione?

Se ripetiamo una serie di operazioni più di 2 volte, forse è meglio scrivere una funzione. Immaginiamo di avere una serie di oggetti e voler eseguire la stessa operazione in tutti. Ad esempio vogliamo centrare (ovvero sottrarre a tutti i valori di un vettore la loro media) un vettore numerico:

```
vettore1 - mean(vettore1)
Vettore2 - mean(vettore2)
Vettore3 - mean(vettore3)
…
vettore_n - mean(vettore_n)
```

L'operazione viene eseguita correttamente ed è anche di facile comprensione. Tuttavia stiamo eseguendo sempre la stessa cosa, semplicemente cambiando un input (proprio la definizione di funzione) quindi possiamo scrivere la seguente funzione:

```r
my_fun <- function(x){
  return(x - mean(x))
}
```

Le funzioni in R

Best practice

Scegliere il nome

Questo potrebbe sembrare un argomento marginale tuttavia la scelta dei nomi sia per le variabili ma sopratutto per le funzioni è estremamente importante. Permette di:

1. leggere chiaramente il nostro codice e renderlo comprensibile ad altri

2. organizzare facilmente un gruppo di funzioni. Quando avete più funzioni, usare una giusta denominazione permette di sfruttare i suggerimenti di RStudio in modo più efficace. Il pacchetto stringr ad esempio che fornisce strumenti per lavorare con stringhe, utilizza tutte le funzioni denominate come str_ permettendo di cercare facilmente quella desiderata.

E' utile utilizzare verbi per nominare le funzioni mentre nomi per nominare argomenti.

Ad esempio un nome adatto alla nostra ultima funzione potrebbe essere center_var() mentre il nome del nuovo vettore centered_vec o c_vec.

Se troviamo center_var all'interno di uno script è subito chiaro il compito di quella funzione, anche senza guardare il codice al suo interno.

Le funzioni in R

Best practice

Semplificare la quantità di operazioni e output

Questo è un punto molto importante ma allo stesso tempo variegato. Ci sono diversi stili di programmazione e quindi non ci sono regole fisse oppure delle pratiche migliori di altre.

Abbiamo detto che una funzione è un modo per astrarre, riutilizzare e semplificare una serie di operazioni. Possiamo quindi scrivere funzioni molto complesse che ricevono diversi input, eseguono diverse operazioni e restituiscono diversi output.

E' buona pratica però scrivere funzioni che:

1. riducono il numero di operazioni interne
2. forniscono un singolo (o limitati) output
3. hanno un numero di input limitato

Se quindi abbiamo pensato ad una funzione che ha troppi output, è troppo complessa oppure ha troppi input magari possiamo valutare di scomporre la funzione in sotto-funzioni.

Le funzioni in R

Best practice

Commentare e documentare

La documentazione è forse la parte di più importante della scrittura del codice.

Possiamo classificarla in documentazione formale e informale in base allo scopo. La documentazione formale è quella che troviamo facendo help(funzione) oppure ?funzione.

E' una documentazione standardizzata e necessaria quando si creano delle funzioni in un pacchetto che altri utenti devono utilizzare.

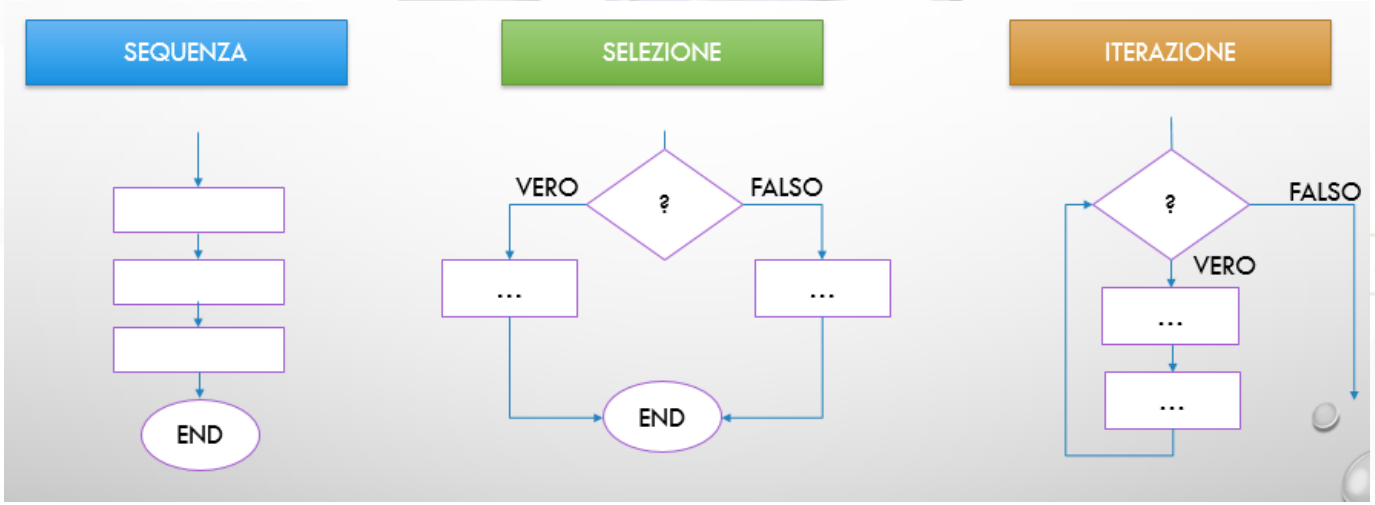
La documentazione informale è quella che mettiamo nei nostri script e all'interno delle funzioni come # commento.

Entrambe sono molto importati e permettono di descrivere lo scopo generale della funzione, i singoli argomenti e i passaggi eseguiti.

Programmazione iterativa

L'essenza della maggior parte delle operazioni nei vari linguaggi di programmazione è quella del concetto di iterazione.
Iterazione significa **ripetere una porzione di codice un certo numero di volte o fino anche una condizione viene soddisfatta.**

Molte delle funzioni che abbiamo usato finora si basano su operazioni iterative.
In R, purtroppo o per fortuna, userete abbastanza raramente delle iterazioni tramite loop anche se nella maggior parte delle funzioni sono presenti. Infatti molte delle funzioni implementate in R sono disponibili solo con pacchetti esterni oppure devono essere scritte manualmente implementando strutture iterative.



Programmazione iterativa

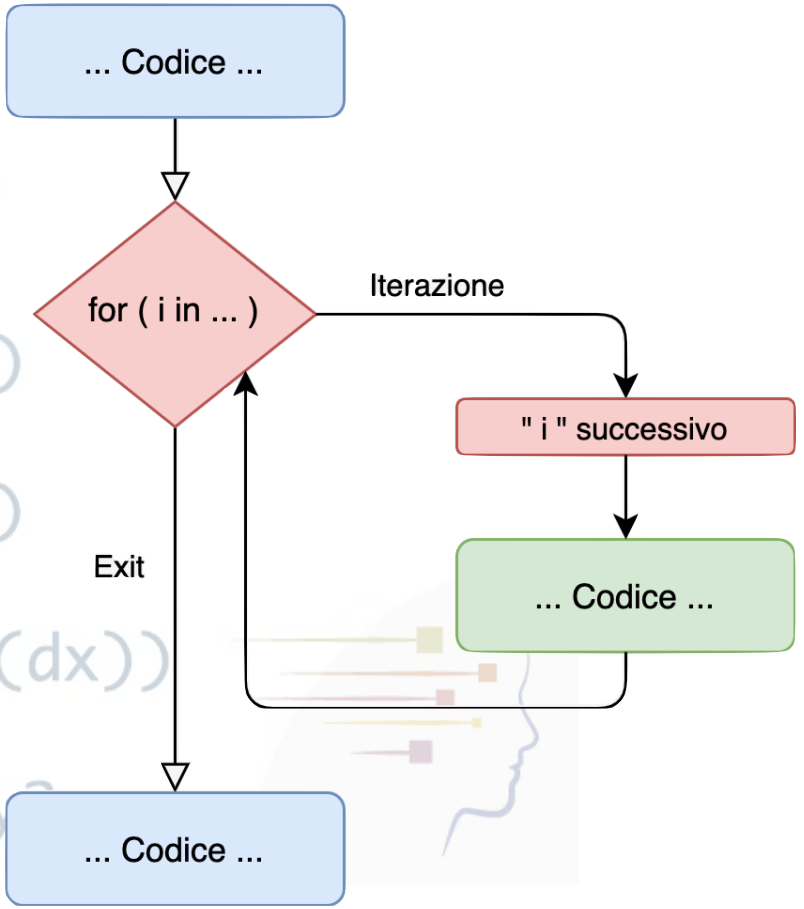
Looping

Il primo tipo di struttura iterativa viene denominata ciclo `for`. L'idea è quella di ripetere una serie di istruzioni un numero **predefinito** di volte. La figura rappresenta l' idea di un ciclo `for`.

In modo simile alle strutture condizionali, quando scriviamo un ciclo entriamo in una parte di codice temporaneamente, eseguiamo le operazioni richieste e poi continuiamo con il resto del codice.

Quello che nell'immagine è chiamato **i** è un modo convenzionale di indicare il conteggio delle operazioni.

Se vogliamo ripetere un'operazione 1000 volte, **i** parte da 1 e arriva fino a 1000.



Programmazione iterativa

Looping

Struttura For Loop

In R la scrittura del ciclo for è la seguente:

```
for (i in c(...)) {
    <codice-da-eseguire>
}
```

1. **i** è un nome generico per indicare la variabile conteggio che abbiamo introdotto prima. Può essere qualsiasi carattere, ma solitamente per un ciclo generico si utilizzano singole lettere come i o j probabilmente per una similarità con la notazione matematica che spesso utilizza queste lettere per indicare una serie di elementi
2. **in** è l'operatore per indicare che i varia rispetto ai valori specificati di seguito
3. **c**(…) è il range di valori che assumerà i in per ogni iterazione

Possiamo riformulare il codice in:

Ripeti le operazioni incluse tra {} un numero di volte uguale alla lunghezza di **c**(…) *e in questo ciclo,* **i** *assumerà, uno alla volta i valori contenuti in* **c**(…).

Programmazione iterativa

Looping

Struttura For Loop

Informalmente ci sono due tipi di ciclo, quello che utilizza un counter generico da assegnare a `i` e un'altro che utilizza direttamente dei valori di interesse.

Loop con direttamente i valori di interesse

```
# numerico
# caratteri
for (name in c("Alessio", "Beatrice", "Carlo")){
  print(paste0("Ciao ", name))
}
## [1] "Ciao Alessio"
## [1] "Ciao Beatrice"
## [1] "Ciao Carlo"
```

Programmazione iterativa

Looping

Struttura For Loop

Loop che utilizza un counter generico per indicizzare gli elementi:

```
my_vector <- c(93, 27, 46, 99)
# i in 1:length(my_vector)
for (i in seq_along(my_vector)){
  print(my_vector[i])
}
## [1] 93
## [1] 27
## [1] 46
## [1] 99
```

Programmazione iterativa

Looping

Struttura For Loop

Questa distinzione è molto utile e spesso fonte di errori. Se si utilizza direttamente il vettore e il nostro counter assume i valori del vettore, “perdiamo” un indice di posizione. Nell’esempio del ciclo con i nomi infatti, se volessimo sapere e stampare quale posizione occupa Alessio dobbiamo modificare l’approccio, puntando ad utilizzare un counter generico. Possiamo crearlo fuori dal ciclo e aggiornarlo manualmente:

```
i <- 1
for (name in c("Alessio", "Beatrice", "Carlo")){
  print(paste0(name, " è il numero ", i))
  i <- i + 1
}
## [1] "Alessio è il numero 1"
## [1] "Beatrice è il numero 2"
## [1] "Carlo è il numero 3"
```

Programmazione iterativa

Looping

Struttura For Loop

In generale, il modo migliore è sempre quello di utilizzare un ciclo utilizzando gli indici e non i valori effettivi, in modo da poter accedere comunque entrambe le informazioni.

```
nomi <- c("Alessio", "Beatrice", "Carlo")
for (i in seq_along(nomi)){
  print(paste0(nomi[i], " è il numero ", i))
}
## [1] "Alessio è il numero 1"
## [1] "Beatrice è il numero 2"
## [1] "Carlo è il numero 3"
```

Programmazione iterativa

Looping

Struttura For Loop

```
my_vector <- c(93, 27, 46, 99)
my_NULL <- NULL

1:length(my_vector)
## [1] 1 2 3 4
1:length(my_NULL)
## [1] 1 0

seq_along(my_vector)
## [1] 1 2 3 4
seq_along(my_NULL)
## integer(0)

seq_len(length(my_vector))
## [1] 1 2 3 4
seq_len(length(my_NULL))
## integer(0)
```

Programmazione iterativa

Looping

Struttura For Loop – Esempio: la funzione somma

Molte delle funzioni disponibili in R derivano da strutture iterative. Se pensiamo alla funzione sum() sappiamo che possiamo calcolare la somma di un vettore semplicemente con sum(x).
Per capire appieno i cicli, è interessante pensare e implementare le funzioni comuni.
Se dovessimo sommare **n** numeri a mano la struttura sarebbe questa:

1. prendo il primo numero x_1 e lo sommo con il secondo x_2
2. ottengo un nuovo numero x_{1+2}
3. prendo il terzo numero x_3 e lo sommo a x_{1+2}
4. ottengo x_{1+2+3}
5. ripeto questa operazione fino all'ultimo elemento di x_n

Programmazione iterativa

Looping

Struttura For Loop – Esempio: la funzione somma

In R:

```
my_values <- c(2,4,6,8)

# Calcolare somma valori
my_sum <- 0      # inizializzo valore
for (i in seq_along(my_values)){
  my_sum <- my_sum + my_values[i]
}

my_sum
## [1] 20
```

La struttura è la stessa del nostro ragionamento in precedenza. Creo una variabile “partenza” che assume valore 0 ed ogni iterazione sommo indicizzando il rispettivo elemento.

Programmazione iterativa

Looping

Struttura For Loop – Esempio: creazione di un vettore

Poiché utilizziamo un indice che assume un range di valori, possiamo non solo accedere ad un vettore utilizzando il nostro indice ma anche creare o sostituire un vettore progressivamente.

```
# Calcola la somma di colonna
my_matrix <- matrix(1:24, nrow = 4, ncol = 6)

# Metodo non efficiente (aggiungo valori)
sum_cols <- c()
for( i in seq_len(ncol(my_matrix))){
  sum_col <- sum(my_matrix[, i])   # calcolo i esima colonna
  sum_cols <- c(sum_cols, sum_col) # aggiungo il risultato
}

sum_cols
## [1] 10 26 42 58 74 90
```

Programmazione iterativa

Looping

Struttura For Loop – Esempio: creazione di un vettore

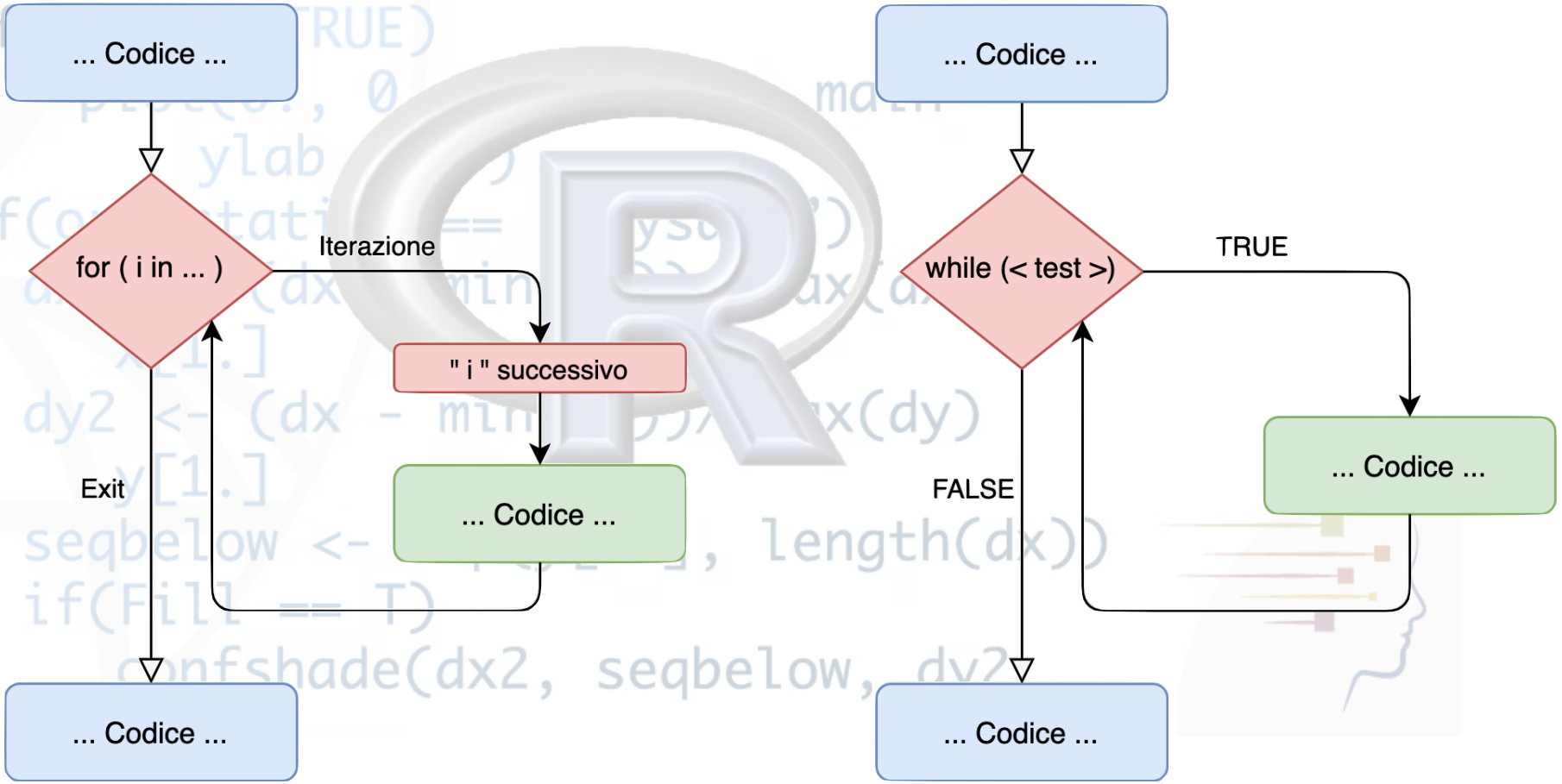
Poiché utilizziamo un indice che assume un range di valori, possiamo non solo accedere ad un vettore utilizzando il nostro indice ma anche creare o sostituire un vettore progressivamente.

```
# Calcola la somma di colonna
my_matrix <- matrix(1:24, nrow = 4, ncol = 6)

# Metodo efficiente (aggiorno valori)
sum_cols <- vector(mode = "double", length = ncol(my_matrix))
for( i in seq_along(sum_cols)){
  sum_col <- sum(my_matrix[, i])   # calcolo i esima colonna
  sum_cols[i] <-  sum_col # aggiorno il risultato
}

sum_cols
## [1] 10 26 42 58 74 90
```

Programmazione iterativa

Il ciclo **while** può essere considerato come una generalizzazione del ciclo **for.** In altri termini il ciclo **for** è un tipo particolare di ciclo **while.**

While



Programmazione iterativa

While

Struttura While Loop

La scrittura è più concisa del ciclo **for** perché non definiamo nessun *counter* o *placeholder* e nemmeno un vettore di valori. L'unica cosa che muove un ciclo **while** è una condizione logica (quindi con valori booleani **TRUE** e **FALSE**).

Ripeti le operazioni incluse tra {} fino a che la condizione <test> *è VERA.*
In altri termini, ad ogni iterazione la condizione <test> *viene valutata. Se questa è vera, viene eseguita l'operazione altrimenti il ciclo si ferma*.

```
while (<test>) {
    <codice-da-eseguire>
}
```

Programmazione iterativa

While

Struttura While Loop – Esempio: un semplice conto alla rovescia

Se vogliamo creare un semplice ciclo while per effettuare un conto alla rovescia:

```
count <- 5

while(count >= 0){
  print(count)
  count <- count - 1 # aggiorno variabile
}
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
## [1] 0
```

Programmazione iterativa

While

Struttura While Loop – Esempio: un semplice conto alla rovescia

Quando si scrive un ciclo while è importante assicurarsi di due cose:

1. Che la condizione sia TRUE inizialmente, altrimenti il ciclo non comincerà nemmeno
2. Che ad un certo punto la condizione diventi FALSE (perché abbiamo ottenuto il risultato o perché è passato troppo tempo o iterazioni)

Se la seconda condizione non è rispettata, otteniamo quelli che si chiamano **endless** loop come ad esempio:

```
count <- 5

# Attenzione loop infinito
while(count >= 0){
  print(count)
  # count <- count - 1
}
```

Programmazione iterativa

While & For

Abbiamo introdotto in precedenza che il **for** è un tipo particolare di **while.** Concettualmente infatti possiamo pensare ad un **for** come un **while** dove il nostro counter **i** incrementa fino alla lunghezza del vettore su cui iterare. In altri termini possiamo scrivere un **for** anche in questo modo:

```
nomi <- c("Alessio", "Beatrice", "Carlo")
i <- 1 # counter

while(i <= length(nomi)){ # condizione
  print(paste0(nomi[i], " è il numero ", i))
  i <- i + 1
}
## [1] "Alessio è il numero 1"
## [1] "Beatrice è il numero 2"
## [1] "Carlo è il numero 3”
```

Programmazione iterativa

Next & Break

All'interno di una struttura iterativa, possiamo eseguire qualsiasi tipo di operazione, ed anche includere strutture condizionali. Alcune volte può essere utile saltare una particolare iterazione oppure interrompere il ciclo iterativo. In R tali operazioni posso essere eseguite rispettivamente con i comandi **next** e **break.**

1. **next** - passa all'iterazione successiva
2. **break** - interrompe l'esecuzione del ciclo

con `for` loop

```
my_vector <- 1:6
for (i in seq_along(my_vector)){
  if (my_vector[i] == 3) next
  if (my_vector[i] == 5) break
  print(my_vector[i])
}
## [1] 1
## [1] 2
## [1] 4
```

con `while` loop

```
count <- 7
while(count >= 0){
  count <- count - 1
  if (count == 5) next
  if (count == 2)
break
  print(count)
}
## [1] 6
## [1] 4
## [1] 3
```

Programmazione iterativa

Nested loop

Una volta compresa la struttura iterativa, è facile espanderne le potenzialità inserendo un ciclo all'interno di un altro. Possiamo avere quanti cicli nested necessari, chiaramente aumenta non solo la complessità ma anche il tempo di esecuzione. Per capire al meglio cosa succede all'interno di un ciclo nested è utile visualizzare gli indici:

```
for(i in 1:3){ # livello 1
  for(j in 1:3){ # livello 2
    for(l in 1:3){ # livello 3
      print(paste(i, j, l))
    }
  }
}
```

Programmazione iterativa

Nested loop

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

Guardando gli indici, è chiaro che il ciclo più interno viene ultimato per primo fino ad arrivare a quello più esterno. La logica è la seguente:

1. Con la prima iterazione entriamo nel ciclo più esterno **i** = 1, poi in quello interno **j** = 1 e in quello più interno **l** = 1.

2. Con la seconda iterazione siamo bloccati nel ciclo interno e quindi sia **i** che **j** saranno 1 mentre **l** sarà uguale a 2.

3. quando il ciclo **l** sarà finito, **i** sarà sempre 1 mentre **j** passerà a 2 e così via

Un aspetto importante è l'utilizzo di indici diversi, infatti i valori **i**, **j** e **l** assumono valori diversi ad ogni iterazione e se usassimo lo stesso indice, non otterremo il risultato voluto.

Programmazione iterativa

Apply family

Esiste una famiglia di funzioni in R estremamente potenti e versatili chiamate *apply. L'asterisco suggerisce una serie di varianti presenti in R che nonostante la struttura e funzione comune hanno degli obiettivi diversi:

1. apply: dato un dataframe (o matrice) esegue la stessa funzione su ogni riga o colonna
2. tapply: dato un vettore di valori esegue la stessa funzione su ogni gruppo che è stato definito
3. lapply: esegue la stessa funzione per ogni elemento di una lista. Restituisce ancora una lista
4. sapply: esegue la stessa funzione per ogni elemento di una lista. Restituisce se possibile un oggetto semplificato (un vettore, una matrice o un array)
5. vapply: analogo a sapply ma richiede di definire il tipo di dati restituiti
6. mapply: è la versione multivariata. Permette di applicare una funzione a più liste di elementi

Programmazione iterativa

Apply family

Prima di illustrare le varie funzioni è utile capire la struttura generale.

In generale queste funzioni accettano un oggetto lista quindi un insieme di elementi e una funzione.

L'idea infatti è quella di avere una funzione che accetta altre funzioni come argomenti e applichi la funzione-argomento ad ogni elemento in input.

Queste funzioni, sopratutto in R, sono spesso preferite rispetto ad utilizzare cicli **for** per **velocità**, **compattezza** e **versatilità.**

Programmazione iterativa

Apply family

Hadley Wickam riporta un bellissimo esempio per capire la differenza tra loop e *apply.
Immaginiamo di avere una serie di vettori e voler applicare alcune funzioni ad ogni vettore, possiamo impostare un semplice loop in questo modo:

```
list_vec <- list(
  vec1 <- rnorm(100),
  vec2 <- rnorm(100),
  vec3 <- rnorm(100),
  vec4 <- rnorm(100),
  vec5 <- rnorm(100)
)
means <- vector(mode = "numeric", length = length(list_vec))
medians <- vector(mode = "numeric", length = length(list_vec))
st_devs <- vector(mode = "numeric", length = length(list_vec))
for(i in seq_along(list_vec)){
  means[i] <- mean(list_vec[[i]])
  medians[i] <- median(list_vec[[i]])
  st_devs[i] <- sd(list_vec[[i]])
}
```

Programmazione iterativa

Apply family

Nonostante sia perfettamente corretto, questa scrittura ha diversi problemi:

E' molto ridondante. Tra calcolare media, mediana e deviazione standard l'unica cosa che cambia è la funzione applicata mentre dobbiamo per ognuno preallocare una variabile, impostare l'indicizzazione in base all'iterazione per selezionare l'elemento della lista e memorizzare il risultato.

Per migliorare questa scrittura possiamo mettere in una funzione tutta questa struttura (preallocazione, indicizzazione e memorizzazione) e utilizzare questa funzione con argomenti la lista di input e la funzione da applicare. Utilizzando la funzione **lapply**:

```
means <- lapply(list_vec, mean)
medians <- lapply(list_vec, median)
st_devs <- lapply(list_vec, sd)
```

Come vedete il codice diventa estremamente compatto, pulito e facile da leggere.

Programmazione iterativa

Apply family

Quali funzioni applicare?

Prima di descrivere nel dettaglio ogni funzione *apply è importante capire quali tipi di funzioni possiamo usare all'interno di questa famiglia.

In generale, qualsiasi funzione può essere applicata ma per comodità possiamo distinguerle in:

1. funzioni già presenti in R
2. funzioni personalizzate (create e salvate nell'ambiente principale)
3. funzioni anonime

Nell'esempio precedente, abbiamo utilizzato la funzione mean semplicemente scrivendo `lapply(lista, mean)`. Questo è possibile perché mean necessita di un solo argomento.

Se tuttavia volessimo applicare funzioni più complesse o aggiungere argomenti possiamo usare la scrittura più generale:

```
means <- lapply(list_vec, function(x) mean(x))
```

Programmazione iterativa

Apply family

Quali funzioni applicare?

L'unica differenza è che abbiamo definito una funzione anonima con la scrittura `function(x) ....` Questa scrittura si interpreta come **ogni elemento di list_vec diventa x, quindi applica la funzione mean() per ogni elemento di list_vec.**

La funzione anonima permette di scrivere delle funzioni non salvate o presenti in R e applicare direttamente ad una serie di elementi.

Possiamo anche usare funzioni più complesse come centrare ogni elemento di list_vec:

```
centered_list <- lapply(list_vec, function(x) x - mean(x))
```

In questo caso, è chiaro che x è un *placeholder* (segnaposto), che assume il valore di ogni elemento della lista list_vec.

Programmazione iterativa

Apply family

Quali funzioni applicare?

L'uso di funzioni anonime è estremamente utile e chiaro una volta compresa la notazione. Tuttavia per funzioni più complesse è più conveniente scrivere salvare la funzione in un oggetto e poi applicarla come per mean.

Usando sempre l'esempio di centrare la variabile possiamo:

```
center_vec <- function(x){
  return(x - mean(x))
}

centered_list <- lapply(list_vec, center_vec)
```

Programmazione iterativa

Apply family

Quali funzioni applicare?

Un ultimo aspetto riguarda un parallelismo tra **x** nei nostri esempi e **i** nei cicli for che abbiamo visto in precedenza.

Proprio come **i**, **x** è una semplice convenzione e si può utilizzare qualsiasi nome per definire l'argomento generico.

Inoltre, è utile pensare a **x** proprio con lo stesso ruolo di **i**, infatti se pensiamo alla funzione in precedenza, **x** ad ogni iterazione prende il valore di un elemento di list_vec proprio come utilizzando il ciclo **for** non con gli indici ma con i valori del vettore su cui stiamo iterando.

Programmazione iterativa

Apply family

Quali funzioni applicare?

Qualche volta infatti può essere utile applicare un principio di indicizzazione anche con l' *apply family:

```
means <- lapply(seq_along(list_vec), function(i) mean(list_vec[[I]]))
```

In questo caso l'argomento non è più la lista ma un vettore di numeri da 1 alla lunghezza della lista (proprio come nel ciclo **for**).

La funzione anonima poi prende come argomento **i** (che ricordiamo può essere qualsiasi nome) e lo utilizza per indicizzare e applicare una funzione.

In questo caso non è estremamente utile, ma con questa scrittura abbiamo riprodotto esattamente la logica del ciclo **for** in un modo estremamente compatto.

Programmazione iterativa

Apply family

Apply

La funzione apply viene utilizzata su **matrici** e/o **dataframe** per applicare una funzione ad ogni dimensione (riga o colonna)**.**

La struttura della funzione è questa:

```
apply(X = , MARGIN = , FUN = , ...)
```

Dove:

- `X` è il dataframe o la matrice
- `MARGIN` è la dimensione su cui applicare la funzione: 1 = riga e 2 = colonna
- `FUN` è la funzione da applicare

Programmazione iterativa

Apply family

Apply - Esempi

Semplici funzioni

```r
my_matrix <- matrix(1:24, nrow = 4, ncol = 6)
# Per riga
apply(my_matrix, MARGIN = 1, FUN = sum)
## [1] 66 72 78 84
# Per colonna
apply(my_matrix, MARGIN = 2, FUN = sum)
## [1] 10 26 42 58 74 90
```

Funzioni complesse

```r
# Coefficiente di Variazione
apply(my_matrix, MARGIN = 2, FUN = function(x){
  mean <- mean(x)
  sd <- sd(x)
  return(round(sd/mean,2))
})
```

Programmazione iterativa

Apply family

tapply

tapply è utile quando vogliamo applicare una funzione ad un elemento che viene diviso in base ad un'altra variabile.

La scrittura è la seguente:

```
tapply(X = , INDEX = , FUN = , ...)
```

Dove:

- `X` è la variabile principale
- `INDEX` è la variabile in base a cui suddividere `X`
- `FUN` è la funzione da applicare

Programmazione iterativa

Apply family

tapply - Esempi

```r
my_data <- data.frame(
  y = sample(c(2,4,6,8,10), size = 32, replace = TRUE),
  gender = factor(rep(c("F", "M"), each = 16)),
  class = factor(rep(c("3", "5"), times = 16))
)
head(my_data, n = 4)

# Media y per classe
tapply(my_data$y, INDEX = my_data$class, FUN = mean)
##     3     5
## 5.875 6.625

# Media y per classe e genere
tapply(my_data$y, INDEX = list(my_data$class, my_data$gender), FUN = mean)
##      F    M
## 3 6.50 5.25
## 5 5.75 7.50
```

Programmazione iterativa

Apply family

sapply

sapply ha la stessa funzionalità di lapply ma ha anche la capacità di restituire una versione semplificata (se possibile) dell'output.

```
sapply(X = , FUN = , ... )
```

Esempi

```
# Media
sapply(my_list, FUN = mean)
## sample_norm sample_unif sample_pois
##   0.5029392   0.3935366   4.2500000
```

Programmazione iterativa

Apply family

sapply

Per capire la differenza, applichiamo sia lapply che sapply con gli esempi precedenti:

```
sapply(list_vec, mean)

lapply(list_vec, mean)

sapply(list_vec, mean, simplify = FALSE)
```

Il risultato di queste operazioni corrisponde ad un valore per ogni elemento della lista list_vec.

lapply restituisce una lista con i risultati mentre sapply restituisce un vettore. Nel caso di risultati singoli come in questa situazione, l'utilizzo di sapply è conveniente mentre mantenere la struttura a lista può essere meglio in altre condizioni.
Possiamo comunque evitare che sapply semplifichi l'output usando l'argomento `simplify = FALSE`.

Programmazione iterativa

Apply family

vapply

vapply è una ancora una volta simile sia a lapply che a sapply. Tuttavia richiede che il tipo di output sia specificato in anticipo. Per questo motivo è ritenuta una versione più *solida* delle precedenti perché permette più controllo su quello che accade.

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

```
# Media
vapply(my_list, FUN = mean, FUN.VALUE = numeric(length = 1L))
## sample_norm sample_unif sample_pois
##   0.5029392   0.3935366   4.2500000
```

Programmazione iterativa

Apply family

vapply

vapply è una ancora una volta simile sia a lapply che a sapply. Tuttavia richiede che il tipo di output sia specificato in anticipo. Per questo motivo è ritenuta una versione più *solida* delle precedenti perchè permette più controllo su quello che accade.

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

```
# Media
vapply(my_list, FUN = mean, FUN.VALUE = numeric(length = 1L))
## sample_norm sample_unif sample_pois
##   0.5029392   0.3935366   4.2500000
```

In questo caso come in precedenza definiamo la lista su cui applicare la funzione. Tuttavia l'argomento FUN.VALUE = numeric(length = 1L) specifica che ogni risultato dovrà essere un valore di tipo numeric di lunghezza 1. Infatti applicando la media otteniamo un singolo valore per iterazione e questo valore è necessariamente numerico.

`sapply()` non restituisce sempre la stessa tipologia di oggetto mentre `vapply()` richiede sia specificato il tipo di output di ogni iterazione.

Programmazione iterativa

Apply family

Lista di funzioni a lista di oggetti

L'*apply family permette anche di estendere la formula **applica una funzione ad una lista di argomenti** applicando diverse funzioni in modo estremamente compatto.
Le funzioni infatti sono oggetti come altri in R e possono essere contenute in liste:

```
list_funs <- list(
  "mean" = mean,
  "median" = median,
  "sd" = sd
)
lapply(list_funs, function(f) sapply(list_vec, function(x) f(x)))
```

Quello che abbiamo fatto è creare una lista di funzioni e poi scrivere due lapply e sapply in modo nested. Proprio come quando scriviamo due loop nested, la stessa funzione viene applicata a tutti gli elementi, per poi passare alla funzione successiva.

Il risultato infatti è una lista dove ogni elemento contiene i risultati applicando ogni funzione. Questo tipo di scrittura è più rara da trovare, tuttavia è utile per capire la logica e la potenza di questo approccio.

Programmazione iterativa

Apply family

mapply

`mapply` è la versione più complessa di quelle considerate perché estende a *n* il numero di liste che vogliamo utilizzare. La scrittura è la seguente:

```
mapply(FUN, ...)
```

Dove:

- `FUN` è la funzione da applicare
- `...` sono le liste di elementi su cui applicare la funzione. E' importante che tutti gli elementi siano della stessa lunghezza

Programmazione iterativa

Apply family

mapply

Proviamo a generare dei vettori da una distribuzione normale, usando la funzione `rnorm()` con diversi valori di numerosità (ns), media (means) e deviazione standard (sds).

```
ns <- c(10, 3, 5)
means <- c(10, 20, 30)
sds <- c(2, 5, 7)

mapply(function(x, y, z) rnorm(x, y, z), # funzione
       ns, means, sds) # argomenti
```

La scrittura è sicuramente meno chiara rispetto agli esempi precedenti ma l'idea è la seguente:

1. La funzione anonima non ha solo un argomento ma *n* argomenti
2. Gli argomenti sono specificati in ordine, quindi nel nostro esempio x = ns, y = means e z = sds
3. Per ogni iterazione, la funzione **rnorm** ottiene come argomenti una diversa numerosità, media e deviazione standard

Programmazione iterativa

Replicate

`replicate` è una funzione leggermente diversa ma estremamente utile. Permette di ripetere una serie di operazioni un numero prefissato di volte.

```
replicate(n = , expr = )
```

Dove:

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

Programmazione iterativa

Replicate – Esempi

```
sample_info <- replicate(n = 1000,{
  my_sample <- rnorm(n = 20, mean = 0, sd = 1)
  my_mean <- mean(my_sample)
  return(my_mean)
})
str(sample_info)
```