3.1_condizionale

Programmazione in R

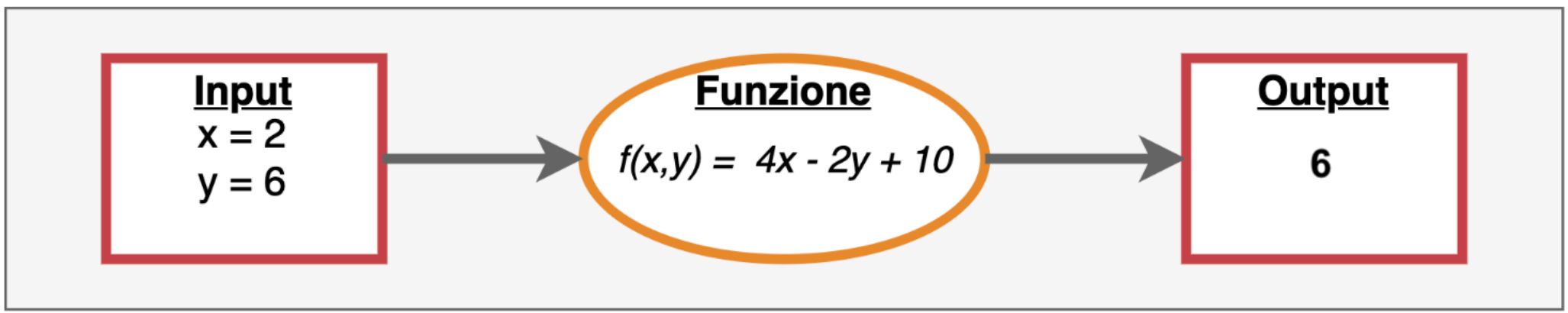
Vedremo i principali costrutti della programmazione e la loro applicazione in R. Molti dei concetti presentati sono trasversali, quindi applicabili anche ad altri linguaggi di programmazione. Qui affronteremo gli aspetti più basici, per applicazioni più avanzate vi suggeriamo il libro: Advanced R

Argomenti

- Costrutti della programmazione
- 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 - Creazione

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:

```
1 myfun_sum = function(x,y){ # argomenti
2
3   x + y  # corpo
4
5   }
6
7 myfun_sum(2,3)
```

```
[1] 5
```

La funzione che ho creato prende in **input** x e y, li **somma**, e fornisce in **output** il risulato.

Funzioni - Creazione

E’ possibile svolgere svariate operazioni dentro una sola funzione. E’ preferibile usare il comando **`return()`** per definire esplicitamente l’ouput che desideriamo, per esempio…

```
myfun = function(x,y, name){

  # corpo
  z = x + y  #sommo x e y
  id = paste(z, name, sep = "_") # creo un codice

  #output
  return(id)
}

myfun(2,3, "carlo")
```

```
[1] "5_carlo"
```

Funzioni - Creazione

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

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

Funzioni - Creazione

Per creare questa funzione dobbiamo quindi definire:

- **argomenti**: variabili da definire (se non già definite)
- **corpo**: le operazioni che la funzione deve eseguire usando gli argomenti
- **output**: cosa la funzione deve restituire come risultato

Funzioni - Creazione

Argomenti

Gli **argomenti** sono quelle parti **variabili** della funzione che vengono definiti e poi sono necessari ad eseguire la funzione stessa. Nel caso della nostra funzione l'unico argomento è il **vettore** in **input**. Possiamo analogalmente a **`mean`** e **`sd`** impostare un secondo argomento che indichi se eliminare gli NA:

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

Funzioni - Creazione

Body

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

```r
z_score = function(x, na.rm = FALSE){ # argomenti

  (x - mean(x, na.rm = na.rm)) / sd(x, na.rm = na.rm)
  # output
}
```

Funzioni - Creazione

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 ma trasformato in punti z. Come abbiamo visto in precedenza, possiamo utilizzare la funzione **`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)

  return(xcen)

}
```

Funzioni - Creazione

Abbiamo quindi creato questa funzione, che diventa un oggetto nel nostro ***enviroment*** e possiamo utilizzarla

```
# creo un vettore x
vect = runif(100, min = 1, max = 10)

mean(vect) # media
```

```
[1] 5.327916
```

```
sd(vect)   # deviazione standard
```

```
[1] 2.751754
```

```
# centro
vec0 = z_score(vect) # di default na.rm =  è impostato TRUE

round(mean(vec0),2) # media 0
```

```
[1] 0
```

```
sd(vec0) # deviazione standard 1
```

```
[1] 1
```

Funzioni - suggerimenti

- Le parentesi grafe **{}** possono essere omesse nel caso in cui il codice sia tutto in una stessa riga
- **`return()`** può essere omesso se l'ultima riga rappresenta l'output desiderato

```
1  z_score1 = function(x){
2    xcen = (x - mean(x))/sd(x
3    return(xcen)
4  }
5  z_score1(vect[1:5])
```

```
[1] -0.5851565  0.8728723
0.8006864 -1.4558692  0.3674670
```

```
1  z_score2 = function(x) (x - mean(x)) / s
2  z_score2(vect[1:5])
```

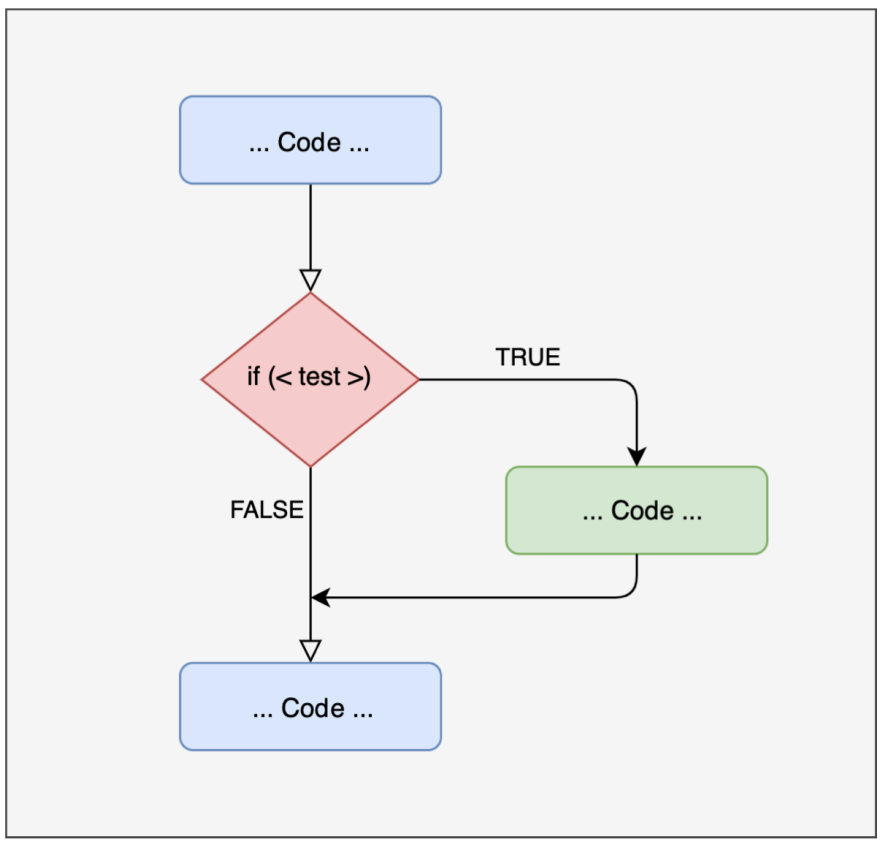
```
[1] -0.5851565  0.8728723  0.8006864
-1.4558692  0.3674670
```

Programmazione condizionale - `if`

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

Programmazione condizionale - `if`

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



`if` statement

Vediamo un'esempio

```r
myfun_if = function(x){ # argomento

  if (x > 0){
    cat("Il valore è maggiore di 0\n")
  }
  cat("Fine funzione")
}

myfun_if(10)
```

```
Il valore è maggiore di 0
Fine funzione
```

```r
myfun_if(-10)
```

```
Fine funzione
```

`if` statement - `STOP`

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.

```
1  myfun_stop = function(x){ # argomento
2
3    if (!is.numeric(x)) { # utile quando vogliamo evitare che la funzione ven
4      stop("il vettore deve essere numerico")
5    }
6    mean(x, na.rm = TRUE)
7  }
8
9  x = 1:10  # vettore num
10 y = letters[1:10] # vettore chr
11
12 myfun_stop(x)
```

```
[1] 5.5
```

```
1  myfun_stop(y)
```

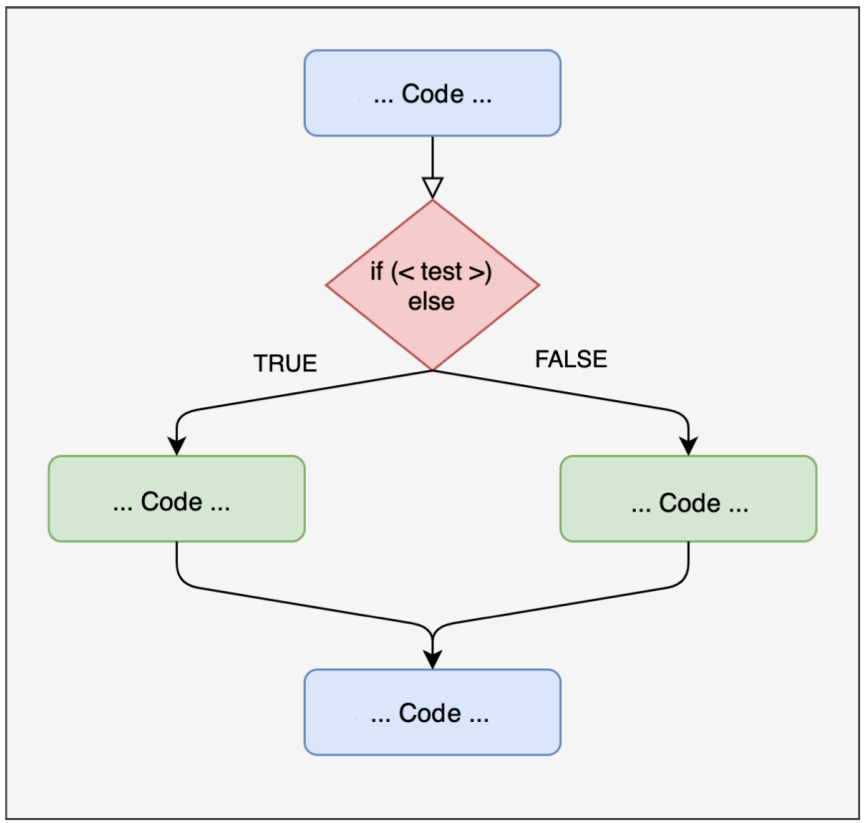
```
Error in myfun_stop(y): il vettore deve essere numerico
```

`if...else`

Il semplice utilizzo di un singolo `if` potrebbe non essere sufficiente in alcune situazioni. Sopratutto perchè possiamo vedere l'`if` come una **deviazione temporanea** dallo script principale che viene imboccata solo se è vera una condizione, altrimenti lo script continua.

`if...else`

Se vogliamo una struttura più “simmetrica” possiamo eseguire delle operazioni se la condizone è vera **`if`** e altre per tutti gli altri scenari (**`else`**).



if...else

Vediamo un esempio

```
myfun_ifelse = function(x){ # argomento

  if (x > 0){
    cat("Il valore è maggiore di 0\n")
  }
  else{
    cat("Il valore è minore di 0\n")
  }
  cat("Fine funzione")
}

myfun_ifelse(10)
```

```
Il valore è maggiore di 0
Fine funzione
```

```
myfun_ifelse(-10)
```

```
Il valore è minore di 0
Fine funzione
```

if...else - nested

```r
myfun_ifelse = function(x){ # argomento
  if (x > 2){
    cat("Il valore è maggiore di 0\n")
  }
  else if (x <= 2 & x >= 0){
    cat("Il valore è compreso tra 2 e 0\n")
  }
  else{
     cat("Il valore è minore di 0\n")
  }
}

myfun_ifelse(2.3)
```

```
Il valore è maggiore di 0
```

```r
myfun_ifelse(-1)
```

```
Il valore è minore di 0
```

```r
myfun_ifelse(1.4)
```

```
Il valore è compreso tra 2 e 0
```

Programmazione condizionale

Per poter capire quale struttura condizionale utilizzare è importante capire bene il problema che dobbiamo risolvere. Facciamo un esempio, 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 fa una determinata operazione.

Programmazione condizionale

Scriviamo una funzione che restituisca la media quando il vettore è numerico e la tabella di frequenza.

```
my_summary =  function(x){

    if(is.numeric(x)){
      return(mean(x))

    }else{
      return(table(x))
      }
  }
```

Programmazione condizionale

Testiamo la funzione

```r
x = 1:7
my_summary(x)
```

```
[1] 4
```

```r
x = c(rep(c("A","B"), 3), "C")
my_summary(x)
```

```
x
A B C
3 3 1
```

Programmazione condizionale

Un limite nell'utilizzare gli `if statement` riguarda il fatto che funzionano solo su un **singolo valore** (i.e. non sono vettorizzati). Quindi mentre la funzione `my_summary` funziona perchè valuta l'intero vettore come numerico (`is.numeric()`)...

Programmazione condizionale

Se volessiamo utilizzare la funzione `myfun_if` ...

```
1 cat(deparse(myfun_if), sep = "\n")
```

```
function (x)
{
    if (x > 0) {
        cat("Il valore è maggiore di 0\n")
    }
    cat("Fine funzione")
}
```

questa non funzionerebbe...

```
1 x = 1:7
2
3 myfun_if(x)
```

```
Error in if (x > 0) {: the condition has length > 1
```

Programmazione condizionale

La versione vettorizzata si ottiene tramite la funzione **`ifelse()`**, i cui argomenti sono la condizione da testare, l'output in caso la condizione risulti **TRUE**, nel caso sia **FALSE**

```
1 x
```

```
[1] 1 2 3 4 5 6 7
```

```
1 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"
[4] "x è minore di 5"   "x è maggiore di 5" "x è maggiore di 5"
[7] "x è maggiore di 5"
```

Programmazione condizionale

Quando abbiamo bisogno di testare più alternative possiamo creare degli **`ifelse()`** nested. Immaginiamo di avere un vettore **age** e voler creare un altro vettore dove l'età è divisa in **3 fasce**, bambino, adulto, anziano:

```r
age = round(runif(10, 3, 80)) # campiono da una distribuzione uniforme da 3
age
```

```
[1] 71 54 77 30 47 29 46 66  5 34
```

```r
age_ifelse = ifelse(age < 18,
                    yes = "bambino",
                    no = ifelse( age >= 18 & age < 60, "adulto", "anziano"))
age_ifelse
```

```
[1] "anziano" "adulto"  "anziano" "adulto"  "adulto"  "adulto"  "adulto"
[8] "anziano" "bambino" "adulto"
```

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 = dplyr::case_when(age < 18 ~ "bambino",
                                 age >= 18 & age < 60 ~ "adulto",
                                 TRUE ~ "anziano")
# con TRUE si identifica "tutto il resto"
```

i risultati ottenuti sono identici...

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

```
[1] TRUE
```

`dplyr::case_when`

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

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

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

```r
library(dplyr) #carico il pacchetto per non dover sempre scrivere il nome

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

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

dplyr::case_when

Queste funzioni si possono applicare anche alle variabili presenti in un dataframe:

```r
# creo un dataframe con variabili id e età
mydf = data.frame(id = factor(1:30), age = sample(14:50, 30))
# lo salvo per dopo
readr::write_csv(mydf, file = "data/mydf.csv")
# creo una terza variabile con "adolescelte","giovane", "adulto
mydf$age_cat = with(mydf,
                    factor(
                      case_when( age > 30 ~ "adulto",
                                 age <= 30 & age >= 20 ~ "giovane",
                                 age < 20 ~ "adolescente",
                                 TRUE ~ "errore" #  check  errori di codifi
                                    )))
str(mydf)
```

```
'data.frame':   30 obs. of  3 variables:
 $ id     : Factor w/ 30 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ age    : int  44 18 32 25 45 33 15 24 30 35 ...
 $ age_cat: Factor w/ 3 levels "adolescente",..: 2 1 2 3 2 2 1 3 3 2 ...
```

Importare una funzione

Abbiamo già visto che il comando `library()` carica un certo pacchetto, rendendo le funzioni contenute disponibili all'utilizzo. Senza la necessità di creare un pacchetto, possiamo comunque organizzare le nostre funzioni in modo efficace.

Importare una funzione

Abbiamo due opzioni

- scrivere le funzioni nello stesso script dove esse vengono utilizzate
- scrivere uno script separato e importare tutte le funzioni contenute

Importare una funzione

Anche in questo caso è una questione di stile e comodità, in generale:

- se abbiamo tante funzioni, è meglio scriverle in uno o più file separati e poi importarle all'inizio dello script principale
- se abbiamo poche funzioni possiamo tenerle nello script principale, magari in una sezione apposita nella parte iniziale

Importare una funzione

Nel secondo caso è sufficiente quindi scrivere la funzione e questa sarà salvata come oggetto nell'ambiente principale. Mentre per il primo scenario è possibile utilizzare la funzione **`source("utl/script.R")`**:

```
rm(list = ls()) # pulisco l'enviorment

# carico le funzioni
source("utl/myfun.R")
# ora avrò le mie funzioni disponibili e potrò utilizzarle
ls()
```

```
[1] "my_summary"    "mydf_fun"      "myfun_ifelse" "myfun_stop"    "z_score"
```

Importare e utilizzare una funzione

Ora avrò le mie funzioni disponibili come oggetti nel mio enviroment e potrò utilizzarle:

```r
# carico il dataframe salvato in precedenza
mydf_1 = data.frame(readr::read_csv("data/mydf.csv"))
str(mydf_1) #check
```

```
'data.frame':   30 obs. of  2 variables:
 $ id : num  1 2 3 4 5 6 7 8 9 10 ...
 $ age: num  44 18 32 25 45 33 15 24 30 35 ...
```

```r
# applico la funzione mydf_fun
mydf_2 = mydf_fun(mydf_1)
head(mydf_2)
```

```
  id age     age_cat       age_z
1  1  44      adulto  1.15416949
2  2  18 adolescente -1.19023729
3  3  32      adulto  0.07213559
4  4  25     giovane -0.55905085
5  5  45      adulto  1.24433898
6  6  33      adulto  0.16230508
```

```r
readr::write_csv(mydf_2, file = "data/mydf_2.csv") # lo salvo per dopo
```

Ora facciamo un po’ di pratica!

Aprite e tenete aperto questo link:

https://etherpad.wikimedia.org/p/arca-corsoR