2. Data Science

27/03/18 Reg 272

Sommario

In questa parte useremo fondamentalmente due software: R e RStudio. Inizieremo con un breve tour di R.

2.1 Tour in R

2.1.5 Dentro R

Il risultato di 1+1 su RStudio è:

1+1

[1] 2

Quel uno iniziale ci indica che anche il risultato è un vettore. R lavora in termini vettoriali. Vediamo ora operatori basilari, ricordiamo che la divisione intera si fa con il doppio percentuale %%:

```
# integer division
31 %/% 3
## [1] 10

# modulus
31 %% 3
## [1] 1

# exponents
2^10
## [1] 1024

# comparison
1 == 1
## [1] TRUE

1 != 1
## [1] FALSE

1 < 1
## [1] FALSE

1 <= 1
## [1] TRUE

# logic operators
# conjunction
TRUE  & TRUE
## [1] TRUE

FALSE & TRUE
## [1] FALSE

TRUE  & FALSE
## [1] FALSE

FALSE & FALSE
## [1] FALSE

# disjunction
TRUE  | TRUE
## [1] TRUE

FALSE | TRUE
## [1] TRUE

TRUE  | FALSE
```

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

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

```
# negation
!TRUE
## [1] FALSE
```

```
!FALSE
## [1] TRUE
```

```
# exclusive disjunction
xor(TRUE, TRUE)
## [1] FALSE
```

```
xor(TRUE, FALSE)
## [1] TRUE
```

```
xor(FALSE, TRUE)
## [1] TRUE
```

```
xor(FALSE, FALSE)
## [1] FALSE
```

Possiamo usare gli operatori appena visti o la versione più lunga, la quale è più efficiente, perché valuta da sinistra a destra e appena il risultato è chiaro dà il risultato, la forma corta valuta tutto. L'and lungo sarebbe **&&** e l'or lungo è **||**. Questo vuol dire che se scriviamo: `TRUE || ESP`, quindi un valore con valore logico vero o una espressione, questo comando da TRUE senza analizzare l'espressione, cosa che avrebbe fatto con **|**.

Ci sono alcuni valori speciali:

- il valore *NA* (non disponibile) viene utilizzato per rappresentare i valori mancanti;
- il valore *NULL* è l'oggetto nullo (da non confondere con NULL nei database);
- il valore *Inf* sta per infinito positivo;
- il valore *NaN* (non un numero) è il risultato di un calcolo che non ha senso.

```
NA & TRUE
## [1] NA
```

```
NA & FALSE
## [1] FALSE
```

```
NA | TRUE
## [1] TRUE
```

```
NA | FALSE
## [1] NA
```

```
!NA
## [1] NA
```

```
2^1024
## [1] Inf
```

```
1/0
## [1] Inf

0 / 0
## [1] NaN

Inf - Inf
## [1] NaN
```

Ci sono tre modi per fare **assegnazioni** alle variabili:

```
x = 45
x <- 45
45 -> x
```

Per vedere il valore della variabile, basta:

```
## [1] 45
# or
print(x)
## [1] 45

# or print the structure of the object
str(x)
##  num 45
```

Non ci sono tanti algoritmi, ma ci sono tante strutture di dati e ne abbiamo sostanzialmente 5:

- Il vettore
- La matrice
- La lista
- L'array
- Data frame

Dimensione	Omogeneo	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Eterogenea vuol dire che può contenere tipi diversi. Se abbiamo un oggetto x per sapere che tipo sia:

```
typeof(x)
```

Vettori:

Un **vettore** è una sequenza di elementi che hanno tuti lo stesso tipo, e si usa le funzione *c()* o la funzione *seq()*. Se volessi un vettore di interi:

```
c(0, 1, 1, 2, 3, 5, 8)
## [1] 0 1 1 2 3 5 8
seq(1, 10, 1)
##  [1]  1  2  3  4  5  6  7  8  9 10
seq(2, 10, 2)
## [1]  2  4  6  8 10
```

Per sapere che tipo di vettore è

```
is.integer(y)
is.numeric(x)
```

Esiste la somma tra vettori. La somma ed altri operatori come il prodotto e la divisione è elemento per elemento del vettore e non nel complesso.

```
# element-wise sum
c(1, 2, 3, 4) + c(10, 20, 30, 40)
## [1] 11 22 33 44

# element-wise product
c(1, 2, 3, 4) * c(10, 20, 30, 40)
## [1]  10  40  90 160

# scalar product
c(1, 2, 3, 4) %*% c(10, 20, 30, 40)
##      [,1]
## [1,]  300
```

Bisogna stare attenti al **recicling:**

```
c(1, 2, 3, 4) + 10
## [1] 11 12 13 14
c(1, 2, 3, 4) + c(10, 20)
## [1] 11 22 13 24
```

I vettori possono essere anche vettori di stringhe o di booleani

```
c("This", "class", "is", "so", "boring!")
```

```
## [1] "This"    "class"   "is"      "so"      "boring!"
x = c(TRUE, FALSE, TRUE, FALSE)
x
## [1]  TRUE FALSE  TRUE FALSE
```

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

a[5]
## [1] 5

a[c(1, 5, 10)]
## [1]  1  5 10

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

a[-c(1, 5, 10)]

## [1] 2 3 4 6 7 8 9

#  Seleziona solo gli  elementi che sono maggiori di 5.
a > 5
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

#  Per prendere solo gli  elementi che del vettore  hanno valor
e  maggiore di  5
a[a > 5]
## [1]  6  7  8  9 10

a[a > 5] = Inf
a
##  [1]   1   2   3   4   5 Inf Inf Inf Inf Inf
```

Tutti gli elementi di un vettore atomico devono essere dello stesso tipo, allora quando cerchiamo di combinare differenti tipi saranno tradotti nel tipo più flessibile (coercizione). L'ordine di flessibilità (dal meno flessibile al più flessibile) è: *logical*, *integer*, *double*, and *character*.

```
as.integer(0.5)
## [1] 0

as.character(0.5)
## [1] "0.5"

as.double("0.5")
## [1] 0.5

as.numeric("a")
## Warning: si è prodotto un NA per coercizione
## [1] NA
```

Si usa il comando ***names()*** per assegnare un nome ai valori di un vettore.

```
x = c(a = 1, b = 2, c = 3)
# or
x = c(1, 2, 3)
names(x) = c("a", "b", "c")

x[c("a", "b")]
## a b
## 1 2
```

Factor

Il **fattore** è un vettore che rappresenta una variabile categoriale (colore degli occhi, sesso, ecc.), cioè un vettore di interi che ha un attributo ***levels*** che mi dice quali sono i valori ammessi. È bene creare factor solo quando so che tipo di dati sto importando

```
x = factor(c("male", "female", "female", "male", "male"))
x
## [1] male   female female male   male
## Levels: female male

levels(x)
## [1] "female" "male"

table(x)
## x
## female   male
##      2      3

# You can't use values that are not levels
x[1] = "unknown"
```

È interessante fare *table()* per notare le frequenze per livello. Se assegnamo il valore "*unknown*" genero un elemento di tipo NA, come sopra. Attenzione che storicamente proprio perché nato dagli statistici, tutti i vettori di stringhe vengono trasformati in factors, questo non è bene perché è bene creare factors solo quando conosciamo in anticipo quali sono i valori noti. Quindi se stiamo importando un DSet allora se abbiamo una colonna che è di stringhe viene convertita in factor, possiamo eliminare questa cosa con *stringeAsFactors=FALSE*

Liste

Arriviamo quindi ad un argomento spinoso: le **liste**. Sono molto importanti perché si usano parecchio, vedremo che un DFrame non è altro che una lista. Abbiamo una sola dimensione, può essere ricorsiva (contenere altre liste o in generale uno strumento strutturale come un vettore), può contenere valori di diverso tipo. Questo ci permette di creare liste che contengono liste, che contengono liste, ecc

```
l = list(thing = "hat", size = 8.25, female = "TRUE")
l

## $thing
## [1] "hat"
##
## $size
## [1] 8.25
```

```
##
## $female
## [1] "TRUE"
```

```
# an element
l$thing
## [1] "hat"
```

```
l[[1]]
## [1] "hat"
```

```
# [  ] prende il vaso di datteri, mentre [[ ]]  prende il dattero
l[["thing"]]
## [1] "hat"
```

Come vediamo per estrarre il primo elemento della lista abbiamo tre metodi, se vogliamo estrarre una sottolista dobbiamo usare il doppio indice[[ ]]. Nota: l[1] abbiamo la sottolista che contiene il primo elemento e l[[1]] è il primo elemento. Bisogna differenziare il contenitore dal contenuto.

```
typeof(l[1])
## [1] "list"
```

```
l[[1]]
## [1] "hat"
```

```
typeof(l[[1]])
## [1] "character"
```

Se x è un treno fatto da più vagoni allora x[[5]] è oggetto della carrozza, mentre x[4:6] sono le carrozze 4,5,6. Per aggiungere l'elemento basta che lo nomini. Se si vuole eliminare un elemento lo si deve assegnare a NULL, cioè all'oggetto vuoto.

You can add and remove elements from a list as follows:

```
l = list(a = 1, b = 2)
l$c = 3
```

```
l
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

```
l$c = NULL
l
$`a`
[1] 1

$b
[1] 2
```

Una lista può contenere un vettore.

```
l= list(thing = "hat", prices = c(8.25, 10.5), female = "TRUE")
l
## $thing
## [1] "hat"
##
## $prices
## [1]  8.25 10.50
##
## $female
## [1] "TRUE"

l$prices[1]
## [1] 8.25
```

Con l'ultimo comando prendiamo il primo elemento del vettore contenuto nella lista.

```
l = list(1, list(1, 2, 3), list("a", 1, list("TRUE", "FALSE")))
```

Una lista potrebbe essere ricorsiva, quindi contenere una lista al suo interno.

Data

```
l = list(1, list(1, 2, 3), list("a", 1, list("TRUE", "FALSE"))) # trovare:
```

- the list list(1, 2, 3):

```
list[[2]]
```

- the element 1 of list list(1, 2, 3)

```
list[[2]][[1]]
```

- the element TRUE of list list("TRUE", "FALSE")

```
l[[3]][[3]][[1]]
```

Si possono combinare le liste fra di loro con:

```
a = list(1, 2, 3)
b = list(3, 2, 1)
c(a, b)
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
```

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

Matrici

Non le useremo troppo, comunque è bene sapere che ci sono. È una struttura di dati omogenea però ha righe e colonne, cioè è bidimensionale:

```
M = matrix(data = 1:9, nrow = 3, ncol = 3, byrow = TRUE)
M
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
nrow(M)
## [1] 3
ncol(M)
## [1] 3
dim(M)
```

Per accedere agli elementi si possono usare gli indici (in questo caso sono due):

```
# element
M[1, 2]
## [1] 2

# first row
M[1, ]
## [1] 1 2 3

# first column
M[ ,1]
## [1] 1 4 7

# sub-matrix
M[1:2, 1:2]

##      [,1] [,2]
## [1,]    1    2
## [2,]    4    5

M[-3, -3]
##      [,1] [,2]
```

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

```
# diagonal
diag(M)
```

```
## [1] 1 5 9
```

```
diag(M) = 0
M
```

```
##      [,1] [,2] [,3]
## [1,]    0    2    3
## [2,]    4    0    6
## [3,]    7    8    0
```

Le operazioni su matrici avvengono elemento per elemento, non nel complesso come per i vettori:

```
# element-wise sum
M + N
```

```
##      [,1] [,2] [,3]
## [1,]    1    6   10
## [2,]    6    5   14
## [3,]   10   14    9
```

```
# element-wise product
M * N
```

```
##      [,1] [,2] [,3]
## [1,]    0    8   21
## [2,]    8    0   48
## [3,]   21   48    0
```

```
# matrix product
M %*% N
```

```
##      [,1] [,2] [,3]
## [1,]   13   28   43
## [2,]   22   52   82
## [3,]   23   68  113
```

```
# matrix transpose
t(M)
```

```
##      [,1] [,2] [,3]
## [1,]    0    4    7
## [2,]    2    0    8
## [3,]    3    6    0
```

```
# matrix inverse
C = matrix(c(1,0,1, 1,1,1, 1,1,0), nrow=3, ncol=3, byrow=TRUE)
D = solve(C)
D
```

```
##      [,1] [,2] [,3]
## [1,]    1   -1    1
## [2,]   -1    1    0
## [3,]    0    1   -1
```

```
D %*% C
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
# linear systems C x = b
C
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    1
## [2,]    1    1    1
## [3,]    1    1    0
```

```
b = c(2,1,3)

# the system is:
# x1      + x3 = 2
# x1 + x2 + x3 = 1
# x1 + x2      = 3
x = solve(C,b)
x
```

```
## [1]  4 -1 -2
```

```
C %*% x
```

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

```
# matrix spectrum
spectrum = eigen(C)
spectrum$vectors
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.4151581 -0.4743098 -0.6026918
## [2,] -0.7480890 -0.2110877  0.7515444
## [3,] -0.5176936  0.8546767  0.2682231
```

```
spectrum$values
```

```
## [1]  2.2469796 -0.8019377  0.5549581
```

```
spectrum2 = eigen(t(C))
spectrum2$vectors
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.7480890  0.2110877 -0.7515444
## [2,] -0.4151581  0.4743098  0.6026918
## [3,] -0.5176936 -0.8546767 -0.2682231
```

```
spectrum2$values
```

```
## [1]  2.2469796 -0.8019377  0.5549581
```

Possiamo aggiungere righe e colonne:

```
M
```

```
##      [,1] [,2] [,3]
## [1,]    0    2    3
## [2,]    4    0    6
```

```
## [3,]    7    8    0
```

```
rbind(M, 10:12)
##      [,1] [,2] [,3]
## [1,]    0    2    3
## [2,]    4    0    6
## [3,]    7    8    0
## [4,]   10   11   12
```

```
M = cbind(M, seq(4, 16, 4))
M
##      [,1] [,2] [,3] [,4]
## [1,]    0    2    3    4
## [2,]    4    0    6    8
## [3,]    7    8    0   12
## [4,]   10   11   12   16
```

Possiamo dare dei nomi sia alle righe che alle colonne:

```
rownames(M) = letters[1:nrow(M)]
colnames(M) = LETTERS[1:ncol(M)]
M
##    A  B  C  D
## a  0  2  3  4
## b  4  0  6  8
## c  7  8  0 12
## d 10 11 12 16
M["a", "A"]
## [1] 0
M["a", ]
## A B C D
## 0 2 3 4
```

Array

È una matrice (ocio) multidimesionale.

```
A = array(1:27, dim=c(3, 3, 3))
A

## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   10   13   16
## [2,]   11   14   17
## [3,]   12   15   18
```

```
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   19   22   25
## [2,]   20   23   26
## [3,]   21   24   27
```

```
A[ , , 1]
```

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

```
A[ , 1, 1]
```

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

```
A[1 , , 1]
```

```
## [1] 1 4 7
```

```
A[1, 1, 1]
```

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

Dataframe

Un **dataframe** è una lista di vettori chiamati *colonne* che però rispetto alla lista devono avere tutti la stessa dimensione, ottengo così una struttura rettangolare. È qualcosa di molto vicino alla tabella di un DB relazionale. Ovviamente essendo una lista le colonne possono avere tipi diversi, all'interno della stessa colonna ci deve essere dello stesso tipo. Esempio:

```
team = c("Inter", "Milan", "Roma", "Palermo")
score = c(59, 58, 53, 46)
win = c(17, 17, 15, 13)
tie = c(8, 7, 8, 7)
lost = c(3, 4, 5, 8)

league = data.frame(team, score, win, tie, lost, stringsAsFactors = FALSE)
league
```

```
##      team score win tie lost
## 1   Inter    59  17   8    3
## 2   Milan    58  17   7    4
## 3    Roma    53  15   8    5
## 4 Palermo    46  13   7    8
```

Possiamo prendere solo alcuni elementi:

```
# first row
league[1, ]
```

```
##    team score win tie lost
## 1 Inter    59  17   8    3
```

```
# first column
league[ ,1]
```

```
## [1] "Inter"   "Milan"   "Roma"    "Palermo"
```

```
league[ ,"team"]
## [1] "Inter"   "Milan"   "Roma"    "Palermo"

league[1:2, 1:2]
##    team score
## 1 Inter    59
## 2 Milan    58

league[1:2, c("team", "score")]
## team score
## 1 Inter 59
## 2 Milan 58
```

Le colonne hanno un nome, cioè delle variabili che passo. Si può accedere alle colonne anche tramite il loro nome.

Possiamo aggiungere le colonne e le righe, quest'ultimo è più complicato, ma basta fare attenzione:

```
rbind(league, data.frame(team = "Lazio", score = 44, win = 12, tie = 8, lost = 8))

cbind(league, goals = c(45, 43, 38, 36))
```

I nomi della nuova tuple devono essere gli stessi del DFrame. Un DFrame è una lista e quindi posso accedere agli elementi con gli accessi tipici delle liste. Se tolgo la doppia parentesi e ne ho solo una, allora abbiamo un DFrame, se no abbiamo l'elemento cercato del suo tipo.

```
# a data frame is a list
typeof(league)
## [1] "list"

league$team
## [1] "Inter"   "Milan"   "Roma"    "Palermo"

# prima colonna DF
league[[1]]
## [1] "Inter"   "Milan"   "Roma"    "Palermo"

league[league$team == "Inter", ]
##    team score win tie lost
## 1 Inter    59  17   8    3

league[league$score == max(league$score), ]
##    team score win tie lost
## 1 Inter    59  17   8    3

ncol(league)
## [1] 5

rownames(league)
## [1] "1" "2" "3" "4"

colnames(league)
## [1] "team"  "score" "win"   "tie"   "lost"
```

Attenzione: dato che un DFrame è una lista di vettori e un vettore può essere una lista, possiamo creare due DFrame con delle colonne di tipo lista, questo fa sì che gli elementi di queste colonne-liste possano essere complicate finché vogliamo, quindi potrebbero essere anche dei DFrame, quindi possiamo avere strutture ricorsive o annidate, cosa che non può succedere nel modello relazionale perché gli elementi sono atomici.

```
# data frame che contiene delle strutture non atomiche (vettori)
df = data.frame( x = I(list(1:3, 4:6)),
        y = c("On the left there is a list",
        "The same here!"),
         stringsAsFactors = FALSE)
str(df)

## 'data.frame': 2 obs. of 2 variables:
## $ x:List of 2
## ..$ : int 1 2 3
## ..$ : int 4 5 6 ## ..- attr(*, "class")= chr "AsIs"
## $ y: chr "On the left there is a list" "The same here!"

# a data frame con elementi data frame, la prima colonna contiene una lista con de
ntro il DF df. E' un DF che contiene DF
df2 = data.frame(
  x = I(list(df, df)),
  y = c("On the left there is a list", "The same here!"),
  stringsAsFactors = FALSE
)


## str(df2)
## 'data.frame':    2 obs. of  2 variables:
##  $ x:List of 2
##   ..$ :'data.frame': 2 obs. of  2 variables:
##   .. ..$ x:List of 2
##   .. .. ..$ : int  1 2 3
##   .. .. ..$ : int  4 5 6
##   .. .. ..- attr(*, "class")= chr "AsIs"
##   .. ..$ y: chr  "On the left there is a list" "The same here!"
##   ..$ :'data.frame': 2 obs. of  2 variables:
##   .. ..$ x:List of 2
##   .. .. ..$ : int  1 2 3
##   .. .. ..$ : int  4 5 6
##   .. .. ..- attr(*, "class")= chr "AsIs"
##   .. ..$ y: chr  "On the left there is a list" "The same here!"
##   ..- attr(*, "class")= chr "AsIs"
##  $ y: chr  "On the left there is a list" "The same here!"
```

Questo lo utilizzeremo soprattutto nel caso di regressione lineare. Sapendo che in R una regressione lineare è un DFrame, lo immetteremo nel DFrame di interesse.

Condizionale e ripetizione

R è un linguaggio orientato all'oggetto. Quindi è possibile scrivere dei cicli if, while e for, ma bisogna usarli con parsimonia.

```
x = 49
if (x %% 7 == 0) x else -x
## [1] 49
```

```
x = 108
i = 2
while (i <= x/2) {
 if (x %% i == 0) print(i)
 i = i + 1;
}
## [1] 2
## [1] 3
## [1] 4
## [1] 6
## [1] 9
## [1] 12
## [1] 18
## [1] 27
## [1] 36
## [1] 54
```

Funzioni

Possiamo scrivere delle funzioni con R. Molte funzioni di R, anzi tutte, hanno dei valori prefissati, che possiamo andare a modificare, per questo R risulta molto flessibile, come la funzione log:

```
log
## function (x, base = exp(1))  .Primitive("log")

log(x = 128, base = 2)
## [1] 7
log(128, 2)
```

O possiamo creare funzioni:

```
euclidean = function(x=0, y=0) {sqrt(x^2 + y^2)}
euclidean(1, 1)
## [1] 1.414214
```

Possiamo scrivere delle funzioni ricorsive come il fattoriale:

```
factorial = function(x)
{ if (x == 0) 1
```

```
 else x * factorial(x-1) } factorial(5)
```

O possiamo creare **funzionali**, cioè delle funzioni che hanno come parametro delle funzioni:

```
g = function(f, n) {
 sum = 0
 for (i in 0:n) sum = sum + f(i)
 return(sum)
}

g(factorial, 5)
## [1] 154
```

Possiamo anche definire altri operatori binari, così creiamo la somma al quadrato

```
'%()%' = function(x, y) {(x + y)^2}
2 %()% 3
```

PACCHETTI

Ultima cosa, i pacchetti di R.

```
# installed packages
(.packages(all.available=TRUE))

# loaded packages (.packages())
# install a package install.packages("igraph")

# load a package
library(igraph)
```

Per l'aiuto in linea o help:

```
?log
?'+'
##  Per cercare la  parola regression dei comandi dell'help:
??"regression"
```

Quando usciamo da R attenzione che nella cartella ci saranno due dati **r.data** e **r.history** questi sono l'ambiente e la storia dei comandi. Possiamo scegliere se salvare in automatico o meno l'ambiente. L'**ambiente** è una tabella che associa le variabili ai nomi, la **storia** invece è la storia dei comandi che abbiamo digitato. Questo è interessante, ma pericoloso perché magari ci troviamo in un ambiente che avevamo creato mesi prima e se salviamo una variabile che pesa tanto diventa una rottura, conviene cancellarla.

2.1.7 Dataframe o Tibblles?

Introduzione (dal libro *R for Data science*)

Ora introdurremo tibbles che migliora leggermente la struttura DB e vedremo ora in che modo. Esso è presente in tibble. La maggior parte degli altri pacchetti R utilizza frame di dati regolari, quindi potresti voler forzare un frame di dati a un tibble. Puoi farlo con `as_tibble()`:

Oppure per creare un tibble la sintassi è simile a quella del DFrame

```
tibble(
  x = 1:5,
  y = 1,
  z = x ^ 2 + y
)
#> # A tibble: 5 x 3
#>       x     y     z
#>   <int> <dbl> <dbl>
#> 1     1     1     2
#> 2     2     1     5
#> 3     3     1    10
#> 4     4     1    17
#> 5     5     1    26
```

Ci sono due principali differenze nell'uso di un tibble rispetto a un classico data.frame: stampa e sottoinsiemi.

Tibbles ha un metodo di stampa raffinato che mostra solo le prime 10 righe e tutte le colonne che si adattano allo schermo. Ciò rende molto più semplice lavorare con dati di grandi dimensioni. Oltre al suo nome, ogni colonna riporta il suo tipo, una caratteristica piacevole presa in prestito da *str()*:

10.3.1 Stampa

```
tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
#> # A tibble: 1,000 x 5
#>   a                   b              c     d e
#>   <dttm>              <date>     <int> <dbl> <chr>
#> 1 2019-01-08 17:33:11 2019-01-15     1 0.368 h
#> 2 2019-01-09 11:38:20 2019-01-20     2 0.612 n
#> 3 2019-01-09 06:02:00 2019-01-30     3 0.415 l
#> 4 2019-01-08 19:23:17 2019-01-29     4 0.212 x
#> 5 2019-01-08 15:47:33 2019-01-26     5 0.733 a
#> 6 2019-01-09 02:48:30 2019-01-22     6 0.460 v
#> # … with 994 more rows
```

I Tibbles sono progettati in modo tale da non sovraccaricare accidentalmente la console quando si stampano frame di dati di grandi dimensioni.

10.3.2 Subsetting

Finora tutti gli strumenti che hai imparato hanno funzionato con frame di dati completi. Se si desidera estrarre una singola variabile, sono necessari alcuni nuovi strumenti "$" e "[[ ]]". [[può estrarre per nome o posizione; mentre $ solo estratti per nome, ma è un po' meno digitante.

```
df <- tibble(
  x = runif(5),
  y = rnorm(5)
)

# Extract by name
```

```
df$x
#> [1] 0.434 0.395 0.548 0.762 0.254

df[["x"]]
#> [1] 0.434 0.395 0.548 0.762 0.254

# Extract by position
df[[1]]
#> [1] 0.434 0.395 0.548 0.762 0.254
```

Per utilizzarli in una pipe, è necessario utilizzare il segnaposto speciale "." :

```
df %>% .$x
#> [1] 0.434 0.395 0.548 0.762 0.254

df %>% .[["x"]]
#> [1] 0.434 0.395 0.548 0.762 0.254
```

Rispetto a a data.frame, i tibbles sono più rigidi: non eseguono mai una corrispondenza parziale e generano un avviso se la colonna a cui stai tentando di accedere non esiste. In tibble non posso creare una tabella con due colonne con lo stesso nome. Il DFrame lo lascia fare ma cambiando il nome della seconda colonna NomeColonna.1. Tibble non fa mai partial matching, cioè posso scrivere l'inizio del nome della colonna e vado a cercare la colonna che inizia in quel modo.

Iniziamo ora il ciclo della Data Science: importazione dei dati, tidy, fase ciclica: trasformazione, visualizzazione model, comunicazione.

Da adesso in poi vedremo una carrellata di tutti questi passaggi. La prima fase è

2.2 IMPORT

Abbiamo i dati in un qualche formato e li vogliamo leggere e scaricare in memoria. Gestiremo questa parte con il pacchetto *readr*. La funzione più importante è la *read_csv()* che legge appunto i file di tipo csv. RStudio prende le prime 1000 righe del file e specifica il tipo, come succede nel libro. Posso aggiungere dei parametri a *read_csv* che alcune volte mi servono. Il parametro *skip = n* salta le prime n righe, poiché magari c'è un commento.

Se mettiamo *comment="#"* diciamo di saltare tutti i commetti. Se c'è un file senza intestazione il comando *col_names=FALSE* legge il file senza intestazione. Oppure posso introdurre io i nomi. Una cosa molto importante l'interpretazione dei valori *NA*. Se abbiamo un file csv in cui il carattere NA è definito tramite il ".", allora si dovrà specificare scrivendo NA="." , se no immette il punto.

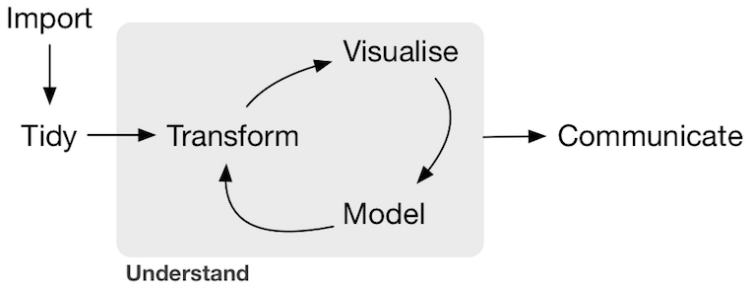
```
chal<-read_csv("challenge.csv", )
```

Questo csv ha dei problemi perché i primi 1000 tuple sono di tipi diversi da quelle successiva, cioè la prima colonna è integer e poi double e la seconda prima ha solo NA e successivamente solo date. Il View mostra infatti una tabella incompleta, questo esercizio ci porta a pensare che se sappiamo i tipi di colonna allora subito le definiamo. Allora possiamo fare così:

```
# Definiamo così i tipi
chal<-read_csv("challenge.csv", col_types=cols(x=col_double(), y=col_date()))
# Giochiamo sporco e facciamo controllare una riga in più
> chal<-read_csv("challenge.csv", guess_max=1001)
```

2.3 TIDY

09/04/2018
reg 1285



Vuol dire ***ordinato*** che è la parola che dà il nome a *tidyverse*, che contiene diversi pacchetti. L'idea è quella di creare una funzione che ordini dei dati disordinati. Una **variabile** è una quantità o una qualità che possiamo misurare. Un **valore** è lo stato di una variabile. Una **osservazione** è un insieme di misure fatte sotto simili condizioni. Ogni osservazione contiene un numero alto di valori associate a differenti variabili. Sulle **colonne** mettiamo le variabili sulle **righe** mettiamo le osservazioni, le **celle** sono i valori. Noi vogliamo i dati in questo modo perché un formato uniforme consente:

- **consistenza**: facilità di apprendimento degli strumenti (ggplot, dplyr, ….) che fanno l'ipotesi che i dati abbiano questa forma e di conseguenza avendo una forma consistente è più facile creare ed usare questi strumenti.
- R è vettoriale ed avere le variabili come colonna ci permette di sfruttare a pieno questa caratteristica di R.

2.3.2 Tidy data

Questo articolo si concentra su un aspetto piccolo ma importante della pulizia dei dati che viene chiamato il **riordino dei** dati : strutturare i set di dati per facilitare l'analisi.

I principi dei **tidy data** forniscono un modo standard per organizzare i valori dei dati all'interno di un data set. Uno standard semplifica la pulizia iniziale dei dati perché non è necessario iniziare da zero e reinventare la ruota ogni volta. Lo standard tidy data è stato progettato per facilitare l'esplorazione e l'analisi iniziale dei dati e per semplificare lo sviluppo di strumenti di analisi dei dati che funzionano bene insieme.
Come famiglie, i dataset ordinati sono tutti uguali ma ogni set di dati disordinato è disordinato a modo suo. I set di dati tidy forniscono un modo standardizzato per collegare la struttura di un set di dati (il suo layout fisico) con la sua semantica (il suo significato).

I dati ordinati sono un modo standard per mappare il significato di un set di dati alla sua struttura. **Un set di dati è in formato *tidy data* se:**

1. Ogni variabile forma una colonna
2. Ogni osservazione forma una riga
3. Ogni tipo di unità osservativa forma una tabella

I set di dati reali possono, e spesso lo fanno, violare i tre precetti dei dati ordinati in quasi tutti i modi immaginabili. I cinque problemi più comuni con i set di dati disordinati, insieme ai loro rimedi, sono:

- Le intestazioni di colonna sono valori, non nomi di variabili.
- Più variabili sono memorizzate in una colonna.
- Le variabili sono archiviate sia in righe che in colonne.
- Più tipi di unità osservative sono memorizzati nella stessa tabella.
- Una singola unità osservativa è memorizzata in più tabelle.
-

Sorprendentemente possiamo agevolmente riordinare i dati attraverso i comandi *gahtering* (raccolta), separating (separazione) e *spreading* (diffusione) che vedremo bene più avanti.

2.3.3 The tidy tools manifesto

Leggi i 4 principi base pensati da Wickham:

- Riutilizzare le strutture dati esistenti.
- Comporre semplici funzioni con la pipe %>%
- Abbracciare la programmazione funzionale
- Designati per gli umani

Una potente strategia per risolvere problemi complessi è combinare molti pezzi semplici. In R, questa strategia si risolve componendo singole funzioni con la pipe. La **pipe**, **%>%**, è uno strumento di composizione comune che funziona su tutti i pacchetti.

Alcune cose da tenere a mente quando si inventano nuove funzioni:

- Cerca di mantenere le funzioni più semplici possibili. Ciascuna funzione dovrebbe fare bene una cosa e dovresti essere in grado di descrivere lo scopo della funzione in una frase.
- I nomi delle funzioni dovrebbero essere i verbi.
- Progetta la tua API principalmente in modo che sia facile da usare per gli esseri umani. Investi tempo a nominare le tue funzioni. I nomi di funzioni evocativi rendono la tua API più facile da usare e da ricordare.

I pacchetti si concentrano su un aspetto piccolo ma importante della pulizia dei dati che è chiamato ***tidyng*** ossia il riordino dei dati, che mira a strutturare il set di dati in modo da facilitarne l'analisi. Il dataset Tidy fornisce un modo standardizzato per collegare la struttura di un set di dati (il suo layout fisico) con la sua semantica (il suo significato).

Un dataset è una raccolta di **valori**. Ogni valore appartiene a una variabile e ad un'osservazione. Una **variabile** contiene tutti i valori che misurano lo stesso attributo sottostante (come altezza, temperatura, durata) tra le unità. Un'**osservazione** contiene tutti i valori misurati sulla stessa unità (come una persona, un giorno o una gara) attraverso gli attributi.

Dati disordinati si dicono ***messy***. Noi dobbiamo trasformali in Tidy: per ogni 'tabella' ordinata di dati è verificata la terza forma normale di Codd cioé che:

- **Ogni colonna è una Variabile** = quantità o una qualità che posso misurare (es temperatura, colore di occhi)
- **Ogni Valore ha una propria cella**. Data una variabile e una misurazione ottengo un valore
- **Ogni Osservazione forma una riga** = insieme di rilevazioni fatte nello stesso luogo e momento su un solo oggetto. Ogni osservazione contiene un certo numero di variabili.

Perché assicurarsi che i dati siano in ordine? Ci sono due vantaggi principali:

- C'è un vantaggio generale nel scegliere un modo consistente di memorizzazione dei dati. Se si dispone di una struttura dati **consistente**, è più semplice imparare gli strumenti che funzionano con esso perché hanno un'uniformità sottostante.
- I dati Tidy sono particolarmente adatti per i linguaggi di programmazione vettorizzati come R, poiché il layout garantisce che i valori di diverse variabili della stessa osservazione siano sempre associati. Ciò rende la trasformazione dei dati in ordine particolarmente naturale.

Inoltre:

Un vantaggio della API (interfaccia di programmazione di un'applicazione) è che non è obbligatoria. Se non ti piace usare la pipe, puoi comporre le funzioni in qualsiasi modo.

2.3.4 Tidy data con tidyr

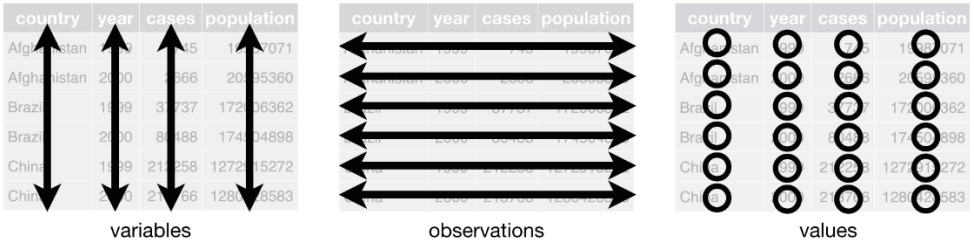
"*I set di dati ordinati sono tutti uguali, ma ogni set di dati disordinato è disordinato a modo suo.*" - Hadley Wickham.

Tratto da qui.

In questo capitolo impareremo un modo coerente per organizzare i tuoi dati in R, un'organizzazione chiamata t**idy data**. Ottenere i dati in questo formato richiede un lavoro in anticipo, ma che viene ripagato a lungo termine. Una volta che hai i tidy data e gli strumenti tidy forniti dai pacchetti inclusi nel *tidyverse*, passerai molto meno tempo a trasformare i dati da una rappresentazione all'altra, permettendoti di dedicare più tempo alle domande analitiche a portata di mano. Questo capitolo ti fornirà un'introduzione pratica ai dati in ordine e agli strumenti di accompagnamento nel pacchetto **tidyr**.

Le 3 regole correlate che rendono un DS tidy, cioè ordinato:

- Ogni variabile deve avere una propria colonna
- Ogni osservazione deve avere una propria riga
- Ogni valore deve avere una propria cella



12.7 Non-tidy data

Esistono due motivi principali per utilizzare altre strutture di dati:

- Le rappresentazioni alternative possono avere vantaggi sostanziali in termini di prestazioni o spazio.
- I campi specializzati hanno sviluppato le proprie convenzioni per l'archiviazione di dati che potrebbero essere molto diversi dalle convenzioni dei dati di ordinamento.

Perché assicurarti che i tuoi dati siano in ordine? Ci sono due vantaggi principali:

- C'è un vantaggio generale nello scegliere un modo consistente di memorizzazione dei dati. Se si dispone di una struttura dati consistente, è più facile imparare gli strumenti che funzionano con esso perché hanno un'uniformità sottostante.
- C'è un vantaggio specifico nel posizionare le variabili nelle colonne perché permette alla natura vettorizzata di R di brillare. Come hai imparato la maggior parte delle funzioni R integrate funziona con i vettori di valori. Ciò rende la trasformazione dei dati in ordine particolarmente naturale.

12.3 Spreading and gathering dei dati

La maggior parte dei dati che incontrerò è disordinata. Ci sono due ragioni principali:

- La maggior parte delle persone non ha familiarità con i principi dei dati ordinati, ed è difficile ricavarli da soli, a meno che non si passi molto tempo a lavorare con i dati. Consiglio: meglio perdere più tempo per capire come ordinare i dati, piuttosto che perderne di più dopo a lavoro avviato.

- I dati sono spesso organizzati per facilitare un altro loro utilizzo (diverso dall'analisi). Ad esempio, i dati sono spesso organizzati per rendere l'inserimento di nuove informazioni il più semplice possibile.

Il primo passo da fare è sempre capire quali sono le variabili e le osservazioni. Il secondo passo è risolvere uno dei due problemi comuni:

- Una variabile potrebbe essere distribuita su più colonne.
- Un'osservazione potrebbe essere sparpagliata su più righe.

Un insieme di dati può solo soffrire di uno di questi problemi, per risolvere questi problemi avrai bisogno delle due funzioni più importanti in tidyr: `gather()` e `spread()`

12.3.1 Gathering

Rende la **tabella più stretta e più lunga ("tabella in piedi"). ("Nomi colonne non utili").**
Gathering letteralmente vuol dire raccolta\raduno. Un problema comune è che in un set di dati alcuni nomi di colonne non sono nomi di variabili, bensì valori di una variabile.
Ogni riga rappresenta informazioni relative a due unità diverse. È quindi violato il vincolo che ogni osservazione va inserita in una sola riga.

```
#> # A tibble: 3 x 3
#>   country     `1999` `2000`
#> * <chr>        <int>  <int>
#> 1 Afghanistan    745   2666
#> 2 Brazil       37737  80488
```

Per riordinare un set di dati come questo, **dobbiamo riunire queste colonne in una nuova coppia di variabili**.
Per descrivere quell'operazione abbiamo bisogno di 3 parametri, con questa sintassi:

1) Il set di colonne che rappresentano valori, non le variabili. In questo esempio, quelle sono le colonne *1999* e *2000*.
2) Il nome della (nuova) variabile i cui valori formano i nomi delle colonne Io chiamo ***key***, e qui è *year*.
3) Il nome della (nuova) variabile (in corrispondenza ai valori dell'altra nuova variabile) i cui valori sono distribuiti sulle celle lo chiamo ***value*** e qui è *cases*

Le colonne da raccogliere sono specificate con la notazione *dplyr::select()*. Qui ci sono solo due colonne, quindi le elenchiamo singolarmente. Insieme questi parametri generano la chiamata ***gather()***:

➔

```
table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
```

```
#> # A tibble: 6 x 3
#>   country     year   cases
#>   <chr>       <chr>  <int>
#> 1 Afghanistan 1999     745
#> 2 Brazil      1999   37737
#> 3 China       1999  212258
#> 4 Afghanistan 2000    2666
#> 5 Brazil      2000   80488
#> 6 China       2000  213766
```

country	year	cases
Afghanistan	1999	745
Afghanistan	2000	2666
Brazil	1999	37737
Brazil	2000	80488
China	1999	212258
China	2000	213766

country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

table4

12.3.2 Spreading (=espansione)

Rende la **tabella più larga e più corta ("tabella seduta"). ("una osservazione su due righe").**

E' l'operazione opposta alla *gathering()*. Letteralmente significa "spalmare". Si applica quando una singola osservazione è spalmata su più righe.
Ad esempio, prendi la table2: un'osservazione è un paese in un anno, ma ogni osservazione è distribuita su due file.

```
table2
#> # A tibble: 12 x 4
#>    country      year type           count
#>    <chr>       <int> <chr>          <int>
#> 1 Afghanistan  1999 cases            745
#> 2 Afghanistan  1999 population  19987071
#> 3 Afghanistan  2000 cases           2666
#> 4 Afghanistan  2000 population  20595360
#> 5 Brazil       1999 cases           37737
#> 6 Brazil       1999 population 172006362
#> # … with 6 more rows
```

Mettiamolo in ordine, prima analizziamo la rappresentazione in modo simile a `gather()`. Questa volta, tuttavia, abbiamo solo bisogno di 2 parametri:

- la colonna che contiene i nomi delle variabili, è la colonna "chiave" ***key***. Qui è la colonna *type*.
- la colonna che contiene i valori di più variabili è la colonna ***value***. Qui è la colonna *count*.

➔

```
spread(table2, key = type, value = count)
#> # A tibble: 6 x 4
#>    country      year  cases population
#>    <chr>       <int>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
#> 3 Brazil       1999  37737  172006362
#> 4 Brazil       2000  80488  174504898
#> 5 China        1999 212258 1272915272
#> 6 China        2000 213766 1280428583
```

country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

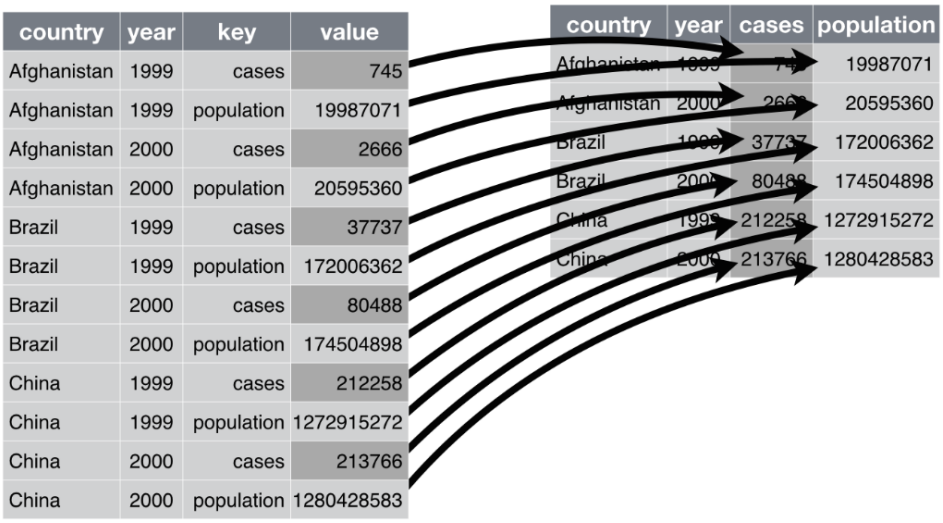


table2

12.4 Separating and uniting

Qui abbiamo una colonna (rate) che contiene due variabili (cases e population). Per risolvere questo problema, avremo bisogno della funzione *separate().* Potresti anche conoscere il complemento di separate(): *unite(),* che si usa se una singola variabile è distribuita su più colonne.

12.4.1 Separate

Risolve il problema di avere più valori in una cella. Qui l'errore è che una colonna contiene più variabili. ***separate()*** separa una colonna in più colonne, dividendo dove appare un carattere separatore. Utilizzo separate() quando i valori delle celle non sono atomici.

```
table3
#> # A tibble: 6 x 3
#>   country      year rate
#> * <chr>       <int> <chr>
#> 1 Afghanistan  1999 745/19987071
#> 2 Afghanistan  2000 2666/20595360
#> 3 Brazil       1999 37737/172006362
#> 4 Brazil       2000 80488/174504898
#> 5 China        1999 212258/1272915272
#> 6 China        2000 213766/1280428583
```

country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

table3

Per impostazione predefinita, separate () divide i valori ovunque visualizzi un carattere non alfanumerico (cioè un carattere che non è un numero o una lettera).

Sintassi: colonna da separare, nuove colonne, separatore, convert.

```
separate(rate, into = c("cases", "population"), sep = "/") , convert = TRUE)
```

Aggiungendo sep = 2 separo dopo il secondo carattere.

```
table3 %>%
  separate(year, into = c("century", "year"), sep = 2)

#> # A tibble: 6 x 4
#>   country     century year  rate
#>   <chr>       <chr>   <chr> <chr>
#> 1 Afghanistan 19      99    745/19987071
#> 2 Afghanistan 20      00    2666/20595360
#> 3 Brazil      19      99    37737/172006362
#> 4 Brazil      20      00    80488/174504898
#> 5 China       19      99    212258/1272915272
#> 6 China       20      00    213766/1280428583
```

In generale interpreterà gli interi come posizioni su cui dividere. I valori positivi iniziano da 1 sull'estrema sinistra delle stringhe; il valore negativo inizia a -1 all'estremità destra delle stringhe.

12.4.2 Unite

unite() è l'inverso di ***separate()***: combina più colonne in una singola colonna. Ne avrò bisogno molto meno frequentemente di separate(), ma è comunque uno strumento utile da avere nelle proprie tasche.

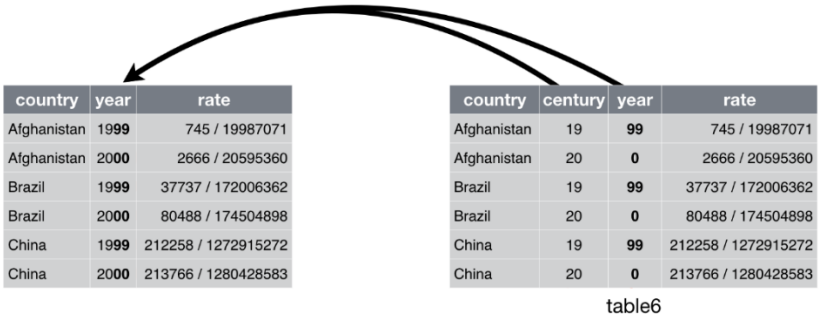
```
table5 %>%
  unite(new, century, year)

#> # A tibble: 6 x 3
#>   country     new   rate
#>   <chr>       <chr> <chr>
#> 1 Afghanistan 19_99 745/19987071
#> 2 Afghanistan 20_00 2666/20595360
#> 3 Brazil      19_99 37737/172006362
#> 4 Brazil      20_00 80488/174504898
#> 5 China       19_99 212258/1272915272
#> 6 China       20_00 213766/1280428583
```

In questo caso dobbiamo anche usare l'argomento ***sep***. Qui non vogliamo alcun separatore, quindi usiamo "":

➔

```
table5 %>%
  unite(new, century, year, sep = "")
#> # A tibble: 6 x 3
#>   country     new   rate
#>   <chr>       <chr> <chr>
#> 1 Afghanistan 1999  745/19987071
#> 2 Afghanistan 2000  2666/20595360
#> 3 Brazil      1999  37737/172006362
#> 4 Brazil      2000  80488/174504898
#> 5 China       1999  212258/1272915272
#> 6 China       2000  213766/1280428583
```

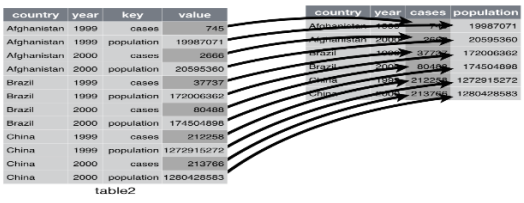


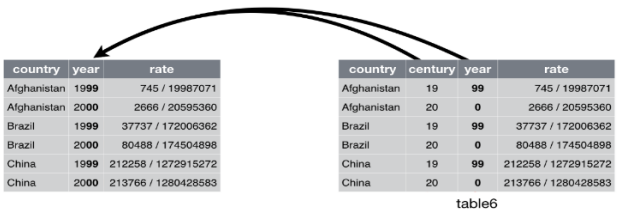
country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	century	year	rate
Afghanistan	19	99	745 / 19987071
Afghanistan	20	0	2666 / 20595360
Brazil	19	99	37737 / 172006362
Brazil	20	0	80488 / 174504898
China	19	99	212258 / 1272915272
China	20	0	213766 / 1280428583

table6

Sintesi mia

gather ()	Ogni riga rappresenta informazioni relative a due unità diverse  table4	%>% gather(year, return, `2015`:`2016`, na.rm = TRUE)
spread()	Una singola osservazione è spalmata su più righe.  table2	%>% spread(table2, key = type, value = count)
separate()	separa una colonna in più colonne  table3	%>% separate(rate, into = c("cases", "population"))
unite()	combina più colonne in una singola colonna.	%>%



country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	century	year	rate
Afghanistan	19	99	745 / 19987071
Afghanistan	20	0	2666 / 20595360
Brazil	19	99	37737 / 172006362
Brazil	20	0	80488 / 174504898
China	19	99	212258 / 1272915272
China	20	0	213766 / 1280428583

table6

```
unite(new, century, year, sep = "")
```

Esercizio su dataset WHO (vd r.proj 9/4)

12.5 Valori mancanti

La modifica della rappresentazione di un set di dati mostra un'importante sottigliezza dei valori mancanti. Sorprendentemente, un valore può mancare in uno dei due modi possibili:

- Esplicitamente, cioè contrassegnato con *NA*.
- Implicitamente, ovvero semplicemente non presente nei dati.

Un valore mancante esplicito è la presenza di un'assenza; un valore mancante implicito è l'assenza di una presenza.

Che fare?

na.rm = TRUE in gather()

è possibile impostare `na.rm = TRUE` in **`gather()`**modo da rendere espliciti i valori mancanti impliciti:

```
stocks %>%
  spread(year, return)
#> # A tibble: 4 x 3
#>     qtr `2015` `2016`
#>   <dbl>  <dbl>  <dbl>
#> 1     1   1.88  NA
#> 2     2   0.59   0.92
#> 3     3   0.35   0.17
```

➔

```
stocks %>%
  spread(year, return) %>%
  gather(year, return, `2015`:`2016`, na.rm = TRUE)
#> # A tibble: 6 x 3
#>     qtr year  return
#>   <dbl> <chr>  <dbl>
#> 1     1 2015    1.88
#> 2     2 2015    0.59
#> 3     3 2015    0.35
#> 4     2 2016    0.92
```

complete()

Un altro importante strumento per rendere espliciti i valori mancanti nei dati di ordine è **`complete()`**. prende una serie di colonne e trova tutte le combinazioni uniche. Quindi assicura che il set di dati originale contenga tutti quei valori, compilando in modo esplicito *NA* se necessario.

➔

```
stocks %>%
  complete(year, qtr)
#> # A tibble: 8 x 3
#>    year   qtr return
```

```
#>   <dbl> <dbl>  <dbl>
#> 1  2015     1   1.88
#> 2  2015     2   0.59
#> 3  2015     3   0.35
#> 4  2015     4  NA
```

fill()

Infine, un ultimo comando, a volte quando un'origine dati è stata utilizzata principalmente per l'immissione dei dati, i valori mancanti indicano che il valore precedente deve essere riportato:

```
treatment <- tribble(
  ~ person,           ~ treatment, ~response,
  "Derrick Whitmore", 1,           7,
  NA,                 2,           10,
  NA,                 3,           9,
  "Katherine Burke",  1,           4
)
```

Puoi inserire questi valori mancanti con `fill()`. Richiede un set di colonne in cui si desidera sostituire i valori mancanti con il valore non mancante più recente (talvolta chiamato ultima osservazione riportata in avanti).

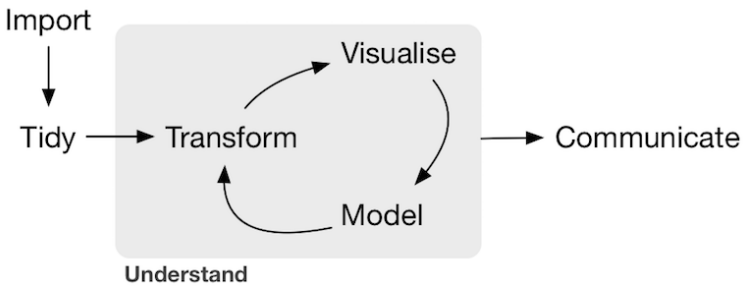
➔

```
treatment %>%
  fill(person)
#> # A tibble: 4 x 3
#>   person           treatment response
#>   <chr>                <dbl>    <dbl>
#> 1 Derrick Whitmore         1        7
#> 2 Derrick Whitmore         2       10
#> 3 Derrick Whitmore         3        9
#> 4 Katherine Burke          1        4
```

Arriviamo così alla trasformazione

2.4 Transform

9/4/2018



Fatta l'importazione dei dati possiamo partire con la parte fondante di tidyverse. Ora siamo nel blocco centrale delle fasi della datascience. Vediamo ora l'operatore Pipe e le sue alternative. Il pipe ci serve per accorciare la distanza tra quello che diremo e le frasi che utilizzeremo per fare il processo di trasformazione e il linguaggio che faremo al pc. Avere un linguaggio di programmazione che ci permette di scrivere questioni vicino al linguaggio naturale è molto comodo. Ci sono tuttavia dei casi in cui il pipe è meglio non utilizzarlo. Se abbiamo comandi molto lunghi è meglio spezzarli per facilitare il debug. Il pipe è un

operatore lineare, cioè ci permette di formare delle sequenze, quindi si sposa bene con i operatori unari (select, order,….) e non con quelli binari come il join.

18 Pipes

Tratto da qui

Il pipe (pipa o tubo) è uno strumento potente per esprimere chiaramente una sequenza di operazioni multiple. Aiuta ad accorciare le istruzioni per il calcolatore rendendole più comprensibili per chi legge il codice. È il momento di esplorare la pipa in modo più dettagliato. Imparerai le alternative alla pipe, quando non dovresti usare la pipe, e alcuni utili strumenti correlati.

18.1.1 Prerequisiti

Il pacchetto che contiene il pipe è ***magrittr***, oltre a ***%>%***. I pacchetti tidyverse caricano automaticamente anche questo pacchetto. C'è anche %T>% che ritorna il lato sinistro e non il destro.

18.2 Piping alternatives

Lo scopo di pipe è aiutarti a scrivere codice in un modo che è più facile da leggere e capire. Per capire perché la pipa è così utile, esploreremo diversi modi di scrivere lo stesso codice: traduzione della filastrocca in 4 modi diversi

Little bunny Foo Foo
Went hopping through the forest
Scooping up the field mice
And bopping them on the head

Useremo una funzione per ciascun verbo della filastrocca, e utilizzeremo lo schema seguente:

- Salva ogni passaggio intermedio come nuovo oggetto.
- Sovrascrivi l'oggetto originale molte volte.
- Comporre funzioni.
- Usa la pipa.

1- Lo svantaggio principale di questa forma è che ti costringe a nominare ogni elemento intermedio. Problemi: Il codice è ingombro di nomi non importanti e devi incrementare attentamente il suffisso su ogni riga.

2- Invece di creare oggetti intermedi in ogni fase, potremmo sovrascrivere l'oggetto originale. Risolve problema precedente, ma aumenta la probabilità di errore. Inoltre, ha 2 problemi:
   - Il debugging è doloroso: se si commette un errore, è necessario rieseguire la pipeline completa dall'inizio.
   - La ripetizione dell'oggetto che viene trasformato (abbiamo scritto foo_foo sei volte!) E' ignoto ciò che cambia su ogni riga.

3- Un altro approccio è quello di abbandonare l'assegnazione e stringere semplicemente una funzione che chiama l'insieme. Il problema qui è che devi leggere da dentro-fuori, da destra a sinistra, e che gli argomenti finiscono per diffondersi a parte (è chiamato il problema del *sandwhich di dagwood*). In breve, questo codice è difficile da utilizzare per un essere umano.

➔

4- Usare la pipa %>%
Infine, possiamo usare la pipe. Puoi leggere questa serie di composizioni di funzioni come se fosse un insieme di azioni imperative. Foo Foo salta, poi scava, poi salta.

```
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mice) %>%
  bop(on = head)
```

La pipe funziona eseguendo una "trasformazione lessicale": dietro le quinte, magrittr riassembla il codice nella pipe in una forma che funziona sovrascrivendo un oggetto intermedio. Ciò significa che la pipe non funzionerà per due classi di funzioni:

- Funzioni che utilizzano l'ambiente corrente. Ad esempio, *assign()* creerà una nuova variabile con il nome specificato nell'ambiente corrente `"x" %>% assign(100).`
  ➔
  Se si desidera utilizzare `assign()` (idem per `get()` e `load()`) con la pipe, è necessario essere espliciti sull'ambiente

```
env <- environment()
"x" %>% assign(100, envir = env)
X
#> [1] 100
```

- Funzioni che utilizzano la valutazione lazy. In R, gli argomenti della funzione vengono calcolati solo quando la funzione li utilizza, non prima di chiamare la funzione. La pipe calcola ogni elemento a turno, quindi non puoi fare affidamento su questo comportamento.

18.3 Quando non usare pipe

È un operatore lineare, quindi si sposa bene con operatori unari. Ma non con join! Quando la relazione fra input e output assomiglia ad una struttura ad albero non conviene usare la pipe.

Le pipe sono più utili per riscrivere una sequenza lineare di operazioni piuttosto breve. Penso che non dovresti usarla quando:

- Le tue pipe sono più lunghe di (diciamo) dieci passi. In tal caso, crea oggetti intermedi con nomi significativi.
- Hai più ingressi o uscite. Se non è stato trasformato un oggetto primario, ma due o più oggetti combinati insieme, non utilizzare la pipe
- Stai iniziando a pensare a un grafico diretto con una struttura di dipendenza complessa.

18.4 Altri strumenti di *magrittr*

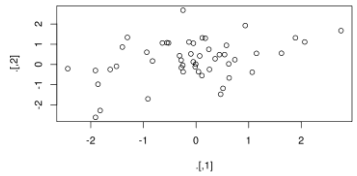
In questo pacchetto ci sono altri strumenti utili:

- a **pipe "tee". %T>%** funziona come %>% se non fosse che restituisce il lato sinistro invece del lato destro. Si chiama "tee" perché è come un tubo a forma di T letterale.

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
```

```
  str()
#>  NULL

rnorm(100) %>%
  matrix(ncol = 2) %T>% (#ocio)
  plot() %>%
  str()
#>  num [1:50, 1:2] -0.387 -0.785 -1.057 -0.796 -1.756 ...
```



- Se stai lavorando con funzioni che non dispongono di un'API basata sui frame di dati. L'operatore **%$%** rende visibili le variabili

```
mtcars %$%
  cor(disp, mpg)
#> [1] -0.848
```

- (tralasciare) Per incarico magrittr fornisce l' operatore **%<>%** che ti permette di sostituire il codice come:

```
mtcars <- mtcars %>%
transform(cyl = cyl * 2)
```

con

```
mtcars %<>% transform(cyl = cyl * 2)
```

Non sono un fan di questo operatore perché ritengo che l'assegnazione sia un'operazione così speciale che dovrebbe essere sempre chiara quando si verifica.

Entriamo nel vivo delle trasformazioni con il pacchetto più famoso: *dplyr*. Che è una grammatica per la trasformazione dei dati. Esso contiene dei verbi, cioè delle funzioni, che servono per manipolare i dati. Questi comandi non fanno altro che simulare le operazioni che abbiamo nell'algebra relazionale e nel SQL. Sono chiamati ***verbi*** perché viene usato un nome che sta nella grammatica dei linguaggi naturali. Questi verbi lavorano in modo molto simile. Anzitutto lavorano su DSet che sono normalizzati secondo tidy. Hanno come argomento un DFrame e come output hanno un DFrame. Lavoreremo spesso con il DFrame dei voli.

Tratto da qui

5 Trasformazione dei dati

La visualizzazione è uno strumento importante per la generazione di informazioni, ma è raro che si ottengano i dati esattamente nella forma corretta di cui si ha bisogno. Spesso è necessario creare nuove variabili o riepiloghi, o forse semplicemente rinominare le variabili usando il pacchetto dplyr e un nuovo set di dati sui voli in partenza da New York nel 2013 (useremo ggplot2 per aiutarci a capire i dati).

```
library(nycflights13)
library(tidyverse)
```

I tipi di dati del DS sono:

int sta per numeri interi.
dbl sta per doppio, o numeri reali.
chr sta per vettori di caratteri o archi.
dttm indica le date (una data + una volta).

lgl sta per logico, vettori che contengono solo TRUEo FALSE.
fctr sta per fattori, che R usa per rappresentare le variabili categoriali con valori possibili fissi.
date sta per le date.

5.1.3 Nozioni di base di dplyr

- Scegli le osservazioni in base ai loro valori (`filter()`).
- Riordina le righe (`arrange()`).
- Scegli variabili con il loro nome (`select()`).
- Crea nuove variabili con funzioni di variabili esistenti (`mutate()`).
- Comprimi molti valori in un singolo riepilogo (`summarise()`)

Questi possono essere usati insieme a quelli ***group_by()*** che modificano l'ambito di ciascuna funzione dal funzionamento sull'intero set di dati per operare su di esso gruppo per gruppo. Queste sei funzioni forniscono i verbi per un linguaggio di manipolazione dei dati.

Tutti i verbi funzionano in modo simile, vediamo la loro struttura grammaticale:

1. Il primo argomento è un frame di dati.
2. Gli argomenti successivi descrivono cosa fare con il frame di dati, usando i nomi delle variabili (senza virgolette).
3. Il risultato è una nuova cornice dati.

5.2 filter(), filtrare le righe

- Il primo argomento è un frame di dati.
- Gli argomenti successivi descrivono cosa fare con il frame di dati, usando i nomi delle variabili (senza virgolette)
- Il risultato è un nuovo dataframe.

```
filter(flights, month == 1, day == 1)
```

5.2.1 Confronti

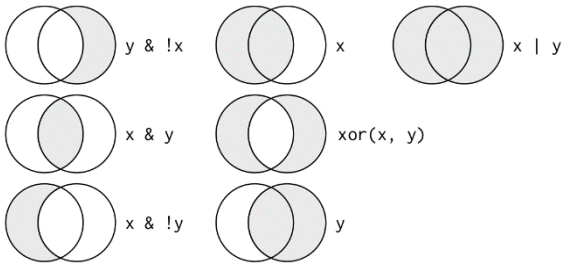
Per utilizzare il filtro in modo efficace, devi sapere **come selezionare le osservazioni** che desideri utilizzando gli operatori di confronto. R fornisce la suite standard: >, >=, <, <=, != (non uguale), e == (pari a).

I computer usano un'aritmetica di precisione finita (ovviamente non possono memorizzare un numero infinito di cifre!), quindi ricorda che ogni numero che vedi è un'approssimazione. Invece di fare affidamento ==, usa near():

```
sqrt(2) ^ 2 == 2
#> [1] FALSE
near(1 / 49 * 49, 1)
#> [1] TRUE
```

5.2.2 Operatori logici

Per esprimere condizioni più elaborate possiamo usare: "and", "|" e "not".



x è il cerchio di sinistra, y è il cerchio di destra, e la regione ombreggiata mostra quali parti ciascun operatore selezionato.

Il primo codice è sbagliato!

```
filter(flights, month == 11 | month == 12) ➔
 nov_dec <- filter(flights, month %in% c(11, 12))
```

Altri esempi con Leggi De Morgan:

```
filter(flights, !(arr_delay > 120 | dep_delay > 120))
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

5.2.3 Valori mancanti

Una caratteristica importante di R che può rendere difficile il confronto sono i valori mancanti o *NA* ("non disponibile"). *NA* rappresenta un valore sconosciuto, quindi i valori mancanti sono "contagiosi": quasi tutte le operazioni che implicano un valore sconosciuto saranno anche sconosciute.

```
NA > 5
#> [1] NA
10 == NA
#> [1] NA
```

Se vuoi determinare se manca un valore, usa is.na(): `is.na(x)`

filter() include solo le righe in cui si trova la condizione TRUE; esclude sia i *FALSE* che i valori *NA*. Se vuoi conservare i valori mancanti, chiedili esplicitamente:

```
filter(df, is.na(x) | x > 1)
```

5.3 arrange(), ordinare le righe

arrange() funziona in modo simile ad *filter()* eccezione del fatto che invece di selezionare le righe, cambia l'ordine delle righe. Prende un frame di dati e una serie di nomi di colonne (o espressioni più complicate) per ordinare.

```
arrange(flights, year, month, day)
```

Utilizzare *desc()* per riordinare una colonna in ordine decrescente:

```
arrange(flights, desc(dep_delay))
```

I valori mancanti vengono sempre messi per ultimi.

5.4 select(), seleziona colonne

select() permette di ingrandire rapidamente un sottogruppo utile usando operazioni basate sui nomi delle variabili

```r
# Select columns by name
select(flights, year, month, day)
# Select all columns between year and day (inclusive)
select(flights, year:day)
# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
```

Esistono numerose funzioni di supporto che puoi utilizzare all'interno di *select()*:

- `starts_with("abc")`: corrisponde ai nomi che iniziano con "abc".
- `ends_with("xyz")`: corrisponde ai nomi che terminano con "xyz".
- `contains("ijk")`: corrisponde ai nomi che contengono "ijk".
- `matches("(.)\\1")`: seleziona le variabili che corrispondono a un'espressione regolare. Questo corrisponde a qualsiasi variabile che contiene caratteri ripetuti.
- `num_range("x", 1:3)`: partite x_1, x_2 e x_3

Usa `select()`, che è una variante di select() quello che mantiene tutte le variabili che non sono esplicitamente menzionate:

```r
rename(flights, tail_num = tailnum)
```

Un'altra opzione è quella di usare select() insieme everything(). Questo è utile se hai una manciata di variabili che desideri spostare all'inizio del frame di dati.

```r
select(flights, time_hour, air_time, everything())
```

5.5 mutate(): aggiungi nuove variabili

```r
mutate(flights_sml,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
```

Se vuoi solo mantenere le nuove variabili, usa `transmute()`:

```r
transmute(flights,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
```

Circa i logaritmi, raccomando di usare log2() perché è facile da interpretare: una differenza di 1 sulla scala del log corrisponde al raddoppio sulla scala originale e una differenza di -1 corrisponde al dimezzamento.

R fornisce funzioni per l'esecuzione di somme, prodotti, minuti e arriva al massimo:

`sumsum()`, `cumprod()`,`cummin()`, `cummax()`;

```r
x
#>  [1]  1  2  3  4  5  6  7  8  9 10
cumsum(x)
```

```
#>  [1]  1  3  6 10 15 21 28 36 45 55
cummean(x)
#>  [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

Ci sono un certo numero di funzioni di classifica, ma dovresti iniziare con **min_rank()**. Fa il tipo più comune di classifica (ad es. 1 °, 2 °, 2 °, 4 °). utilizzare *desc(x)* per dare i valori più grandi ai ranghi più piccoli.

```
y <- c(1, 2, 2, NA, 3, 4)
min_rank(y)
#> [1]  1  2  2 NA  4  5
min_rank(desc(y))
#> [1]  5  3  3 NA  2  1
```

Altre varianti sono:

```
row_number(y)
#> [1]  1  2  3 NA  4  5
dense_rank(y) # ocio alla differenza
#> [1]  1  2  2 NA  3  4
percent_rank(y)
#> [1] 0.00 0.25 0.25   NA 0.75 1.00
cume_dist(y)
#> [1] 0.2 0.6 0.6  NA 0.8 1.0
```

10/4/2018
reg 290

5.6 summarise(), riepiloghi raggruppati

L'ultimo verbo chiave è ***summarise()***. Collassa un frame di dati su una singola riga: min(x), quantile(x, 0.25), max(x), quantile(x, 0.25)

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 1 x 1
#>   delay
#>   <dbl>
#> 1  12.6
```

summarise() è terribilmente utile se abbiano a group_by()!!

Ad esempio, se applichiamo esattamente lo stesso codice a un frame di dati raggruppati per data, otteniamo il ritardo medio per data:

```
by_day <- group_by(flights, year, month, day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 365 x 4
#> # Groups:   year, month [?]
#>    year month   day delay
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1 11.5
#> 2  2013     1     2 13.9
#> 3  2013     1     3 11.0
#> 4  2013     1     4  8.95
#> 5  2013     1     5  5.73
#> 6  2013     1     6  7.15
#> # … with 359 more rows
```

5.6.2 Valori mancanti

Potresti esserti chiesto quale argomento na.rm abbiamo usato sopra. Cosa succede se non lo impostiamo? Riceviamo molti valori mancanti! Questo perché le funzioni di aggregazione obbediscono alla solita regola dei valori mancanti: se c'è un valore mancante nell'input, l'output sarà un valore mancante. Fortunatamente, tutte le funzioni di aggregazione hanno un argomento na.rm che rimuove i valori mancanti prima del calcolo:

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay, na.rm = TRUE))
```

5.6.3 Conta

Ogni volta che si fa qualsiasi aggregazione, è sempre una buona idea includere un conteggio (n()) o un conteggio di valori non mancanti (sum(!is.na(x))). In questo modo puoi verificare che non stai tracciando conclusioni basate su quantità molto piccole di dati.

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
```

Suggerimento RStudio: una scorciatoia da tastiera utile è Cmd / Ctrl + Shift + P. Questo rinvia il blocco inviato in precedenza dall'editor alla console. Questo è molto comodo quando si sta esplorando (es.) Il valore n dell'esempio sopra. Si invia l'intero blocco una volta con Cmd / Ctrl + Invio, quindi si modifica il valore di ne si preme Cmd / Ctrl + Maiusc + P per rinviare il blocco completo

5.6.4 Funzioni di riepilogo utili

Misure di posizione: abbiamo usato mean(x), ma median(x) è anche utile.

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    avg_delay1 = mean(arr_delay),
    avg_delay2 = mean(arr_delay[arr_delay > 0]) # the average positive delay
  )
```

Misure di diffusione: sd(x), IQR(x), mad(x). La deviazione quadratica media alla radice o deviazione standard sd(x) è la misura standard di diffusione. L'intervallo interquartile IQR(x) e la deviazione assoluta mediana mad(x) sono equivalenti robusti che possono essere più utili se si hanno valori anomali.

```
not_cancelled %>%
  group_by(dest) %>%
  summarise(distance_sd = sd(distance)) %>%
  arrange(desc(distance_sd))
```

Misure di rango: min(x), quantile(x, 0.25), max(x)
Misure di posizione: first(x), nth(x, 2), last(x). Questi funzionano in modo simile a x[1], x[2]e x[length(x)]

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first_dep = first(dep_time),
    last_dep = last(dep_time)
  )
```

Conteggi: hai visto `n()`, che non accetta argomenti e restituisce la dimensione del gruppo corrente.

- Per contare il numero di valori non mancanti, utilizzare `sum(!is.na(x))`.
- Per contare il numero di valori distinti (unici), utilizzare `n_distinct(x)`.

```
# Which destinations have the most carriers?
not_cancelled %>%
  group_by(dest) %>%
  summarise(carriers = n_distinct(carrier)) %>%
  arrange(desc(carriers))
#> # A tibble: 104 x 2
#>   dest  carriers
#>   <chr>    <int>
#> 1 ATL          7
#> 2 BOS          7
```

5.6.5 Raggruppamento in base a più variabili

Quando si raggruppano in base a più variabili, ogni riepilogo elimina un livello del raggruppamento. Ciò semplifica il rollup progressivo di un set di dati:

```
daily <- group_by(flights, year, month, day)
(per_day   <- summarise(daily, flights = n()))
#> # A tibble: 365 x 4
#> # Groups:   year, month [?]
#>    year month   day flights
#>   <int> <int> <int>   <int>
#> 1  2013     1     1     842
#> 2  2013     1     2     94
```

Fai attenzione quando riassumi progressivamente: è OK per somme e conteggi, ma devi pensare a ponderare medie e varianze, e non è possibile farlo esattamente per le statistiche basate sulla classifica come la mediana. In altre parole, la somma delle somme di gruppo è la somma complessiva, ma la mediana delle mediane di gruppo non è la mediana complessiva.

5.6.6 Sgruppa (ungroup)

Se è necessario rimuovere il raggruppamento e tornare alle operazioni sui dati non raggruppati, utilizzare *ungroup()*.

```
daily %>%
  ungroup() %>%             # no longer grouped by date
  summarise(flights = n())  # all flights
#> # A tibble: 1 x 1
#>   flights
#>     <int>
#> 1  336776
```

5.7 Mutazioni raggruppate (e filtri)

altra reg 291

Il raggruppamento è più utile in combinazione con summarise(), ma puoi anche fare operazioni convenienti con ***mutate()*** e ***filter()***:

Trova i peggiori membri di ciascun gruppo:

```
flights_sml %>%
  group_by(year, month, day) %>%
  filter(rank(desc(arr_delay)) < 10)
```

Trova tutti i gruppi più grandi di una soglia:

```
popular_dests <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
popular_dests
```

Standardizza per calcolare le metriche di gruppo:

```
popular_dests %>%
  filter(arr_delay > 0) %>%
  mutate(prop_delay = arr_delay / sum(arr_delay)) %>%
  select(year:day, dest, arr_delay, prop_delay)
```

Un filtro raggruppato è un mutate raggruppato seguito da un filtro non raggruppato. Generalmente li evito tranne che per manipolazioni veloci e sporche: altrimenti è difficile controllare di aver eseguito correttamente la manipolazione.

13 Relational data

Tratto da qui

13.1 Introduzione

È raro che un'analisi dei dati coinvolga solo una singola tabella di dati. In genere, esistono molte tabelle di dati e devi combinarle per rispondere alle domande a cui sei interessato. Collettivamente, più tabelle di dati sono chiamate ***dati relazionali***, perché sono le relazioni, non solo i singoli set di dati, che sono importanti.

Le relazioni sono sempre definite tra una coppia di tabelle. Tutte le altre relazioni sono costruite da questa semplice idea: le relazioni di tre o più tabelle sono sempre una proprietà delle relazioni tra ciascuna coppia. A volte entrambi gli elementi di una coppia possono essere la stessa tabella! Questo è necessario se, ad esempio, hai una tabella di persone e ogni persona ha un riferimento ai suoi genitori.

Per lavorare con i dati relazionali sono necessari verbi che funzionano con coppie di tabelle. Esistono tre famiglie di verbi progettate per lavorare con i dati relazionali:

- ✓ I **join mutanti**, che aggiungono nuove variabili a un frame di dati da corrispondenti osservazioni in un altro DF.
- ✓ **Filtraggio dei join**, che filtrano le osservazioni da un frame di dati in base al fatto che corrispondano o meno ad un'osservazione nell'altra tabella.
- ✓ **Imposta le operazioni**, che trattano le osservazioni come se fossero elementi impostati.

Il posto più comune per trovare i dati relazionali è in un sistema di **gestione di database relazionali** (o **RDBMS**), termine che comprende quasi tutti i database moderni. Se hai già usato un database, hai quasi certamente usato SQL. In tal caso, dovresti trovare familiari i concetti di questo capitolo, sebbene la loro espressione in dplyr sia leggermente diversa. Generalmente, dplyr è un po 'più facile da usare rispetto a SQL

perché dplyr è specializzato nell'analisi dei dati: rende più semplici le operazioni di analisi dei dati, a scapito di rendere più difficile fare altre cose che non sono comunemente necessarie per l'analisi dei dati.

(vd sezione 1.5 di *1 DATABASE RELAZIONALI DM4BD*)

```
library(tidyverse)
library(nycflights13)
```

13.3 Chiavi

Le variabili utilizzate per connettere ogni coppia di tabelle sono chiamate **chiavi**. Una chiave è una variabile (o un insieme di variabili) che identifica univocamente un'osservazione. In casi semplici, una singola variabile è sufficiente per identificare un'osservazione. Ad esempio, ciascun aereo è identificato in modo univoco dal suo *tailnum*. In altri casi, potrebbero essere necessarie più variabili. Ad esempio, per identificare un'osservazione weather avete bisogno di cinque variabili: *year, month, day, hour, e origin*.

Esistono due tipi di chiavi:

- Una **chiave primaria** identifica in modo univoco un'osservazione nella ***propria*** tabella. Ad esempio, *planes$tailnum* è una chiave primaria perché identifica univocamente ciascun piano nella tabella planes.
- Una **chiave esterna** identifica in modo univoco un'osservazione in ***un'altra*** tabella. Ad esempio, *flights$tailnum* è una chiave esterna perché appare nella tabella flights in cui corrisponde a ciascun volo su un piano univoco.

Una variabile può essere sia una chiave primaria che una chiave esterna. Ad esempio, *origin* fa parte della chiave primaria weather ed è anche una chiave esterna per la tabella *airport*.

Una volta identificate le chiavi primarie nelle tabelle, è buona norma verificare che effettivamente identificano in modo univoco ogni osservazione. Un modo per farlo è quello fare il *count()* delle chiavi primarie e cercare le voci dove n è maggiore di una:

```
planes %>%
  count(tailnum) %>%
  filter(n > 1)
```

Se una tabella manca di una chiave primaria, a volte è utile aggiungerne una con *mutate()* e *row_number()*. Ciò rende più semplice abbinare le osservazioni se hai fatto qualche filtraggio e vuoi ricontrollare con i dati originali. Questa è chiamata una **chiave surrogata**.

Una chiave primaria e la chiave esterna corrispondente in un'altra tabella formano una **relazione**. Le relazioni sono in genere uno-a-molti. Ad esempio, ogni volo ha un aereo, ma ogni aereo ha molti voli. In altri dati, occasionalmente vedrai una relazione 1 a 1. Puoi pensare a questo come a un caso speciale di 1-a-molti. Puoi modellare relazioni molti-a-molti con una relazione molti-a-1 più una relazione 1-a-molti.

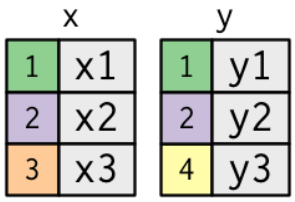
13.4 Vincoli mutanti

Un **join mutante** consente di combinare le variabili da due tabelle. Prima confronta le osservazioni con le loro chiavi, quindi copia le variabili da una tabella all'altra.

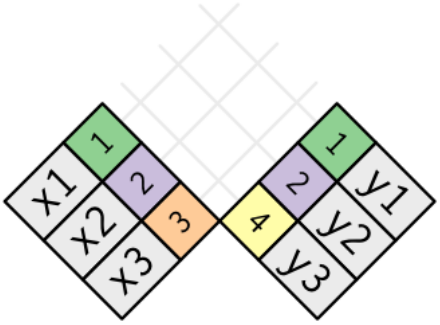
Le quattro funzioni di join mutanti sono l'*inner join* e le tre *outer join*.

13.4.1 Comprensione dei join

Per aiutarti a capire come funzionano i join, userò una rappresentazione visiva. La **colonna colorata** rappresenta la variabile "chiave": questi sono usati per abbinare le righe tra le tabelle. La **colonna grigia** rappresenta la colonna "valore" che viene trasportata per la corsa.

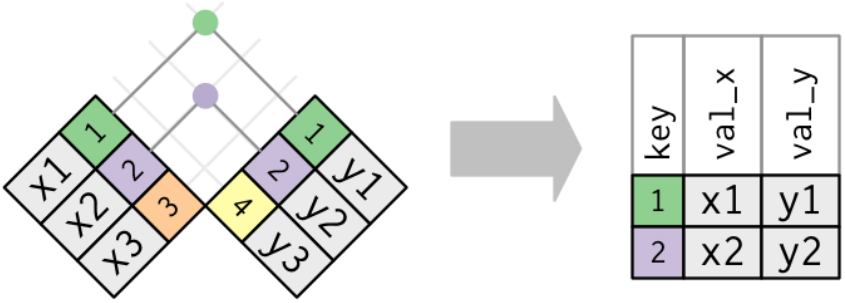


Il diagramma seguente mostra ogni **potenziale corrispondenza** come intersezione di una coppia di linee.



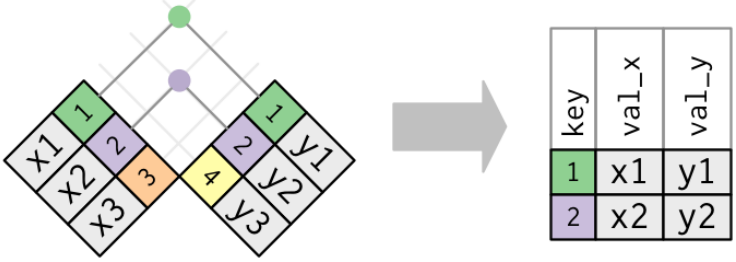
(Se si guarda da vicino, si potrebbe notare che abbiamo cambiato l'ordine delle colonne chiave e valore x. Questo per sottolineare che i join si fondano in base alla chiave, il valore viene semplicemente portato avanti per la corsa.)

In un join effettivo, le partite saranno indicate con punti.



13.4.2 Inner join (valori chiavE in comune)

La traduzione suona come “unione interna”. Il tipo più semplice di join è l’**inner join**. Un inner join corrisponde a coppie di osservazioni ogni volta che le loro chiavi sono uguali:



L'output di un join interno è un nuovo frame di dati che contiene la chiave, i valori x e i valori y. Usiamo *by* per dire a dplyr quale variabile è la chiave:

```
x %>%
  inner_join(y, by = "key")
#> # A tibble: 2 x 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1 x1    y1
#> 2     2 x2    y2
```

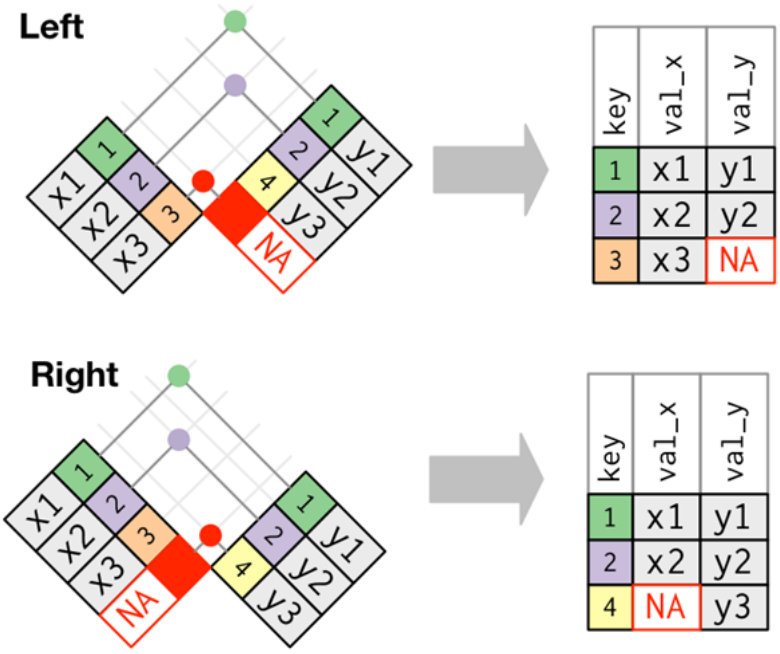
13.4.3 Outer joins (valori chiavE in almeno una table)

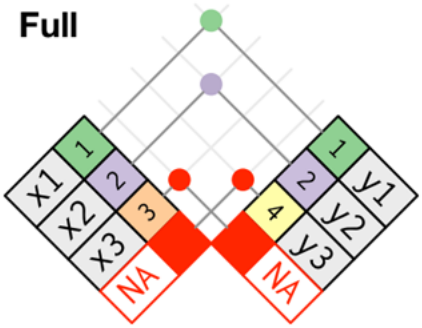
La traduzione suona come "join esterni". Mentre un join interno mantiene le osservazioni che appaiono in entrambe le tabelle, un **outer join** mantiene le osservazioni che appaiono in almeno una delle due tabelle. Esistono tre tipi di join esterni:

- Un ***left join*** (join sinistro) mantiene tutte le osservazioni di sinistra (anche *NA*). Con chi matcha NA?
- Un ***right join*** (join destro) mantiene tutte le osservazioni di destra (anche *NA*). Con chi matcha NA?
- Un **full join** (join completo) mantiene tutte le osservazioni di sinistra e destra (anche *NA*). Ma gli NA non matchiano fra loro!

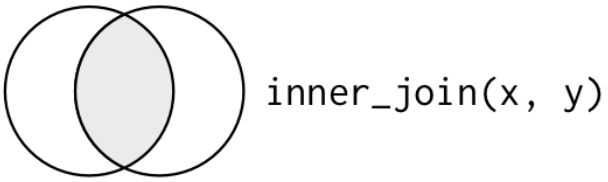
Questi join funzionano aggiungendo un'osservazione "virtuale" aggiuntiva a ogni tabella. Questa osservazione ha una chiave che corrisponde sempre (se nessun'altra chiave corrisponde) e un valore riempito con *NA*.

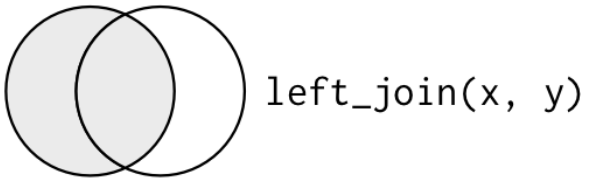
Graficamente

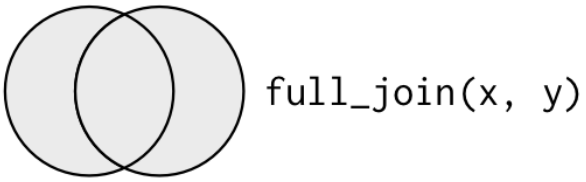


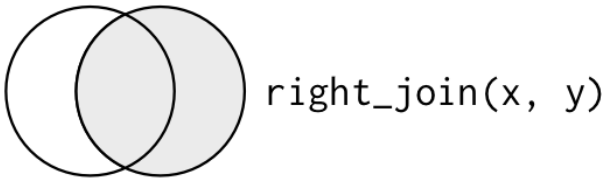


key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3









Il **join più utilizzato è il left-join**: lo si utilizza ogni volta che si cercano dati aggiuntivi da un'altra tabella, perché conserva le osservazioni originali anche quando non c'è una corrispondenza. Il join di sinistra dovrebbe essere il tuo join predefinito: usalo sempre a meno che tu non abbia una buona ragione per preferire uno degli altri.

13.4.5 Definizione delle colonne chiave

Finora, le coppie di tabelle sono sempre state unite da una singola variabile e quella variabile ha lo stesso nome in entrambe le tabelle. Questo vincolo è stato codificato da *by = "key"*. Puoi utilizzare altri valori per *by* collegando le tabelle in altri modi:

- L'impostazione predefinita, by = NULL utilizza tutte le variabili visualizzate in entrambe le tabelle, il cosiddetto **join naturale**. Ad esempio, i voli e le tabelle meteorologiche corrispondono alle loro variabili comuni: year, month, day, hour e origin.

```
flights2 %>%
  left_join(weather)
#> Joining, by = c("year", "month", "day", "hour", "origin")
#> # A tibble: 336,776 x 18
#>    year month   day  hour origin dest  tailnum carrier  temp  dewp humid
#>   <dbl> <dbl> <int> <dbl> <chr>  <chr> <chr>   <chr>   <dbl> <dbl> <dbl>
#> 1  2013     1     1     5 EWR    IAH   N14228  UA       39.0  28.0  64.4
```

- Un vettore di carattere, by = "x". Questo è come un join naturale, ma utilizza solo alcune delle variabili comuni. Ad esempio, *flights* e *planes* hanno le variabili *year*, ma significano cose diverse, quindi vogliamo solo unire *tailnum*.

```
flights2 %>%
  left_join(planes, by = "tailnum")
#> # A tibble: 336,776 x 16
#>   year.x month   day  hour origin dest  tailnum carrier year.y type
#>    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>    <int> <chr>
#> 1   2013     1     1     5 EWR    IAH   N14228  UA        1999 Fixe…
```

Ad esempio, se vogliamo disegnare una mappa, dobbiamo combinare i dati dei voli con i dati degli aeroporti che contengono la posizione (*lat* e *lon*) di ciascun aeroporto. Ogni volo ha un'origine e una destinazione *airport*, quindi dobbiamo specificare a quale vogliamo unirci:

```
flights2 %>%
  left_join(airports, c("dest" = "faa"))
#> # A tibble: 336,776 x 15
#>    year month   day  hour origin dest  tailnum carrier name    lat   lon
#>   <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <chr> <dbl> <dbl>
#> 1  2013     1     1     5 EWR    IAH   N14228  UA      Geor…  30.0 -95.3
```

SQL è l'ispirazione per le convenzioni di dplyr, quindi la traduzione è semplice:

dplyr	SQL
inner_join(x, y, by = "z")	SELECT * FROM x INNER JOIN y USING (z)
left_join(x, y, by = "z")	SELECT * FROM x LEFT OUTER JOIN y USING (z)
right_join(x, y, by = "z")	SELECT * FROM x RIGHT OUTER JOIN y USING (z)
full_join(x, y, by = "z")	SELECT * FROM x FULL OUTER JOIN y USING (z)

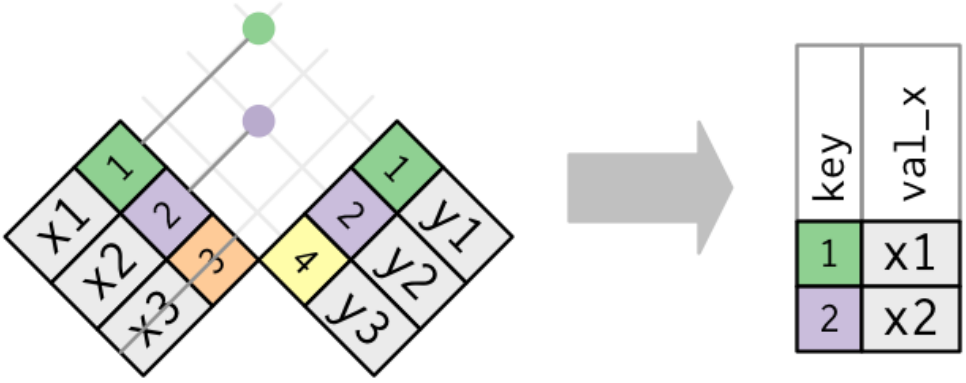
Nota che "INNER" e "OUTER" sono opzionali e spesso omessi.

SQL supporta una gamma più ampia di tipi di join rispetto a dplyr perché è possibile connettere le tabelle utilizzando vincoli diversi dall'uguaglianza (a volte chiamati *non-equijoins*).

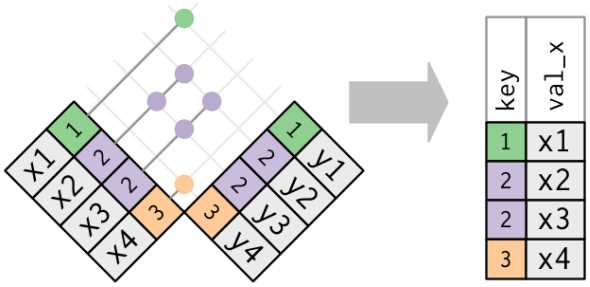
13.5 Filtering joins

Suona come “Filtraggio dei join”. I join di filtro corrispondono alle osservazioni nello stesso modo dei join mutanti, ma influenzano le osservazioni, non le variabili. Ci sono due tipi:
- ***semi_join****(x, y)* **mantiene** tutte le osservazioni in cui x è presente una corrispondenza y.
- **anti_join**(x, y) **elimina** tutte le osservazioni in cui x è presente una corrispondenza y.

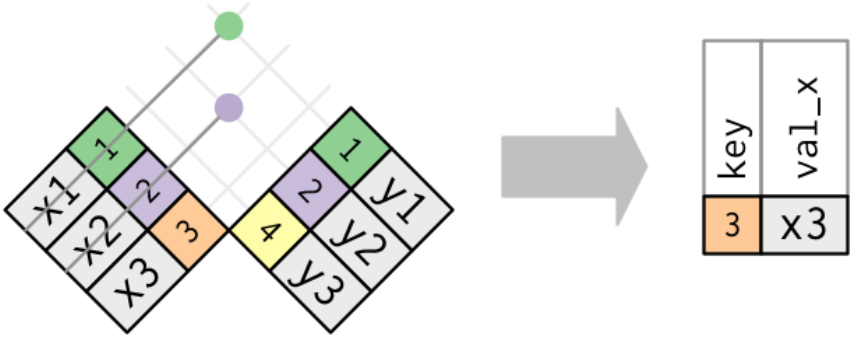
Graficamente, un **semi-join** si presenta così:



Solo l'esistenza di una partita è importante; non importa quale osservazione è abbinata.
NB Ciò significa che i **<u>join di filtro non duplicano mai le righe come i join mutanti</u>** (vd figura)



L'inverso di un semi-join è un anti-join. Un **anti-join** mantiene le righe che non hanno una corrispondenza:



Gli anti-joins sono utili per diagnosticare le mancate corrispondenze di join.

Ad esempio

Table_1

id	t1_value
A	2
B	5
C	8
D	12
E	10

Table_2

id	t2_value
A	4
A	3
A	2
C	3
C	4

Results

id	t1_value
B	5
D	12
E	10

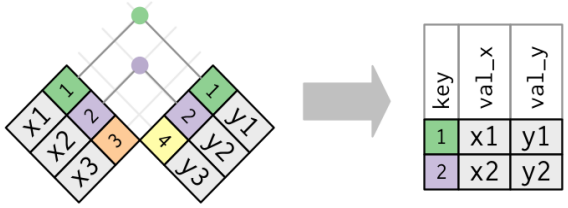
Ad esempio, quando connetti flights e planes potresti essere interessato a sapere che ci sono molti flights che non hanno una corrispondenza in planes:

```r
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(tailnum, sort = TRUE)
```

Riassunto mio su tutti i join

Inner join (valori chiavE in comune)

```
dplyr
 x %>% inner_join(y, by = "key")# oppure
inner_join(x,y, by = "key")
```



SQL

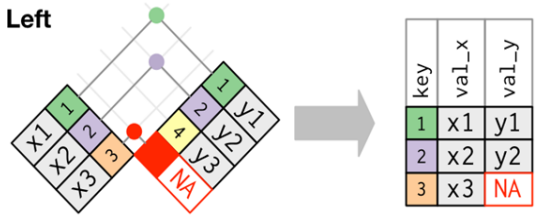
SELECT * FROM x INNER JOIN y USING (key)

Nota che "INNER" è opzionale e spesso è omesso.

3 join esterni (valori chiavE in almeno una table)

1) left join (natural join) mantiene tutte le osservazioni di sinistra (anche *NA)*

```
dplyr
x %>% left_join(y, by = "key") # oppure
left _join(x,y, by = "key")
```



SQL

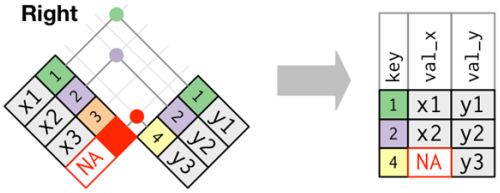
SELECT * FROM x LEFT OUTER JOIN y USING (key)

Nota che "OUTER" è opzionale e spesso è omesso.

➔ Si utilizza ogni volta che si cercano dati aggiuntivi da un'altra tabella, perché conserva le osservazioni originali anche quando non c'è una corrispondenza

2) right join mantiene tutte le osservazioni di destra (anche *NA*)

```
dplyr
x %>% right_join(y, by = "key") # oppure
right_join(x,y, by = "key")
```



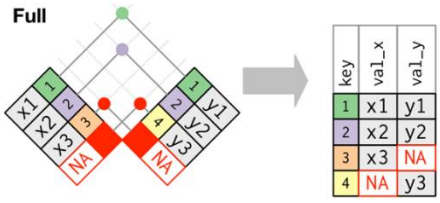
SQL

SELECT * FROM RIGHT OUTER JOIN y USING (key)

Nota che "OUTER" è opzionale e spesso è omesso.

3) full join mantiene tutte le osservazioni di sinistra e destra (anche *NA*, che però non matchiano fra loro)

```
dplyr
x %>% full_join(y, by = "key") # oppure
full_join(x,y, by = "key")
```



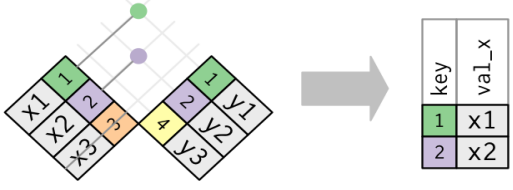
SQL

SELECT * FROM x FULL OUTER JOIN y USING (key)

Filtering joins

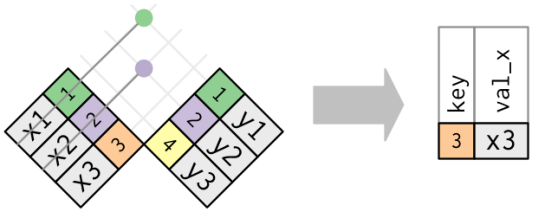
semi_join(x, y)

mantiene tutte le osservazioni in cui x è presente una corrispondenza y.



anti_join(x, y)

elimina tutte le osservazioni in cui x è presente una corrispondenza y.



13.6 Controlli preliminari su un DS (per problemi pratici)

I dati con cui hai lavorato in questo capitolo sono stati ripuliti in modo da avere il minor numero possibile di problemi. È improbabile che i tuoi dati siano così belli, quindi ci sono alcune cose che dovresti fare con i tuoi dati personali per rendere i tuoi contatti fluidi.

1. Inizia identificando le variabili che formano la chiave primaria in ogni tabella. Di solito dovresti farlo basandoti sulla tua comprensione dei dati, non empiricamente cercando una combinazione di variabili che fornisca un identificatore univoco. Se cerchi le variabili senza pensare a cosa significano, potresti ottenere (s) fortuna e trovare una combinazione unica nei tuoi dati attuali, ma la relazione potrebbe non essere vera in generale.

2. Verifica che nessuna delle variabili nella chiave primaria manchi. Se manca un valore, allora non può identificare un'osservazione!

3. Verifica che le tue chiavi esterne corrispondano alle chiavi primarie in un'altra tabella. Il modo migliore per farlo è con un anti_join(). È comune che le chiavi non corrispondano a causa di errori di immissione dei dati. Fissare questi è spesso un sacco di lavoro.

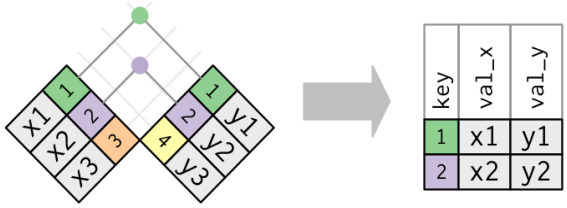
4. Se mancano le chiavi, è necessario riflettere sull'utilizzo dei join interni e esterni, valutando attentamente se si desidera eliminare le righe che non hanno una corrispondenza

5. Tieni presente che il semplice controllo del numero di righe prima e dopo il join non è sufficiente per garantire che il tuo join sia andato a buon fine. Se hai un inner join con chiavi duplicate in entrambe le tabelle, potresti essere sfortunato dato che il numero di righe eliminate potrebbe essere esattamente uguale al numero di righe duplicate!

Riassunto mio sui join

Inner join (valori chiavE in comune)

```
dplyr
 x %>% inner_join(y, by = "key")# oppure
inner_join(x,y, by = "key")
```



SQL

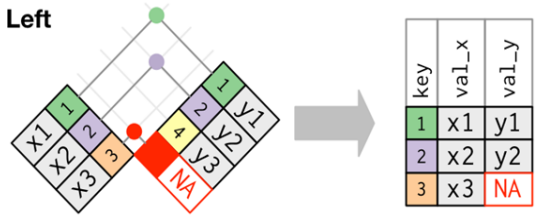
SELECT * FROM x INNER JOIN y USING (key)

Nota che "INNER" è opzionale e spesso è omesso.

3 join esterni (valori chiavE in almeno una table)

1) left join (natural join) mantiene tutte le osservazioni di sinistra (anche *NA)*

```
dplyr
x %>% left_join(y, by = "key") # oppure
left _join(x,y, by = "key")
```



SQL

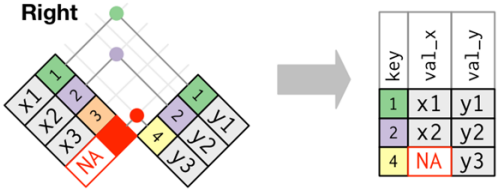
SELECT * FROM x LEFT OUTER JOIN y USING (key)

Nota che "OUTER" è opzionale e spesso è omesso.

➔ Si utilizza ogni volta che si cercano dati aggiuntivi da un'altra tabella, perché conserva le osservazioni originali anche quando non c'è una corrispondenza

2) right join mantiene tutte le osservazioni di destra (anche *NA*)

```
dplyr
x %>% right_join(y, by = "key") # oppure
right_join(x,y, by = "key")
```



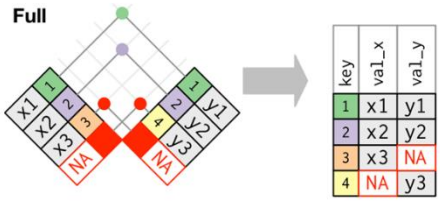
SQL

SELECT * FROM RIGHT OUTER JOIN y USING (key)

Nota che "OUTER" è opzionale e spesso è omesso.

3) full join mantiene tutte le osservazioni di sinistra e destra (anche *NA*, che però non matchiano fra loro)

```
dplyr
x %>% full_join(y, by = "key") # oppure
full_join(x,y, by = "key")
```



SQL

SELECT * FROM x FULL OUTER JOIN y USING (key)

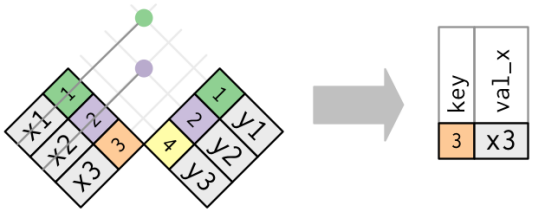
Filtering joins

semi_join(x, y)

mantiene tutte le osservazioni in cui x è presente una corrispondenza y.

anti_join(x, y)

elimina tutte le osservazioni in cui x è presente una corrispondenza y.



13.7 Imposta operazioni

Tutte queste operazioni funzionano con una riga completa, confrontando i valori di ogni variabile. Questi si aspettano che gli input x e y abbiano le stesse variabili e trattino le osservazioni come set:

- **intersect**(x, y): restituisce solo le osservazioni in entrambi x e y
- **union**(x, y): restituisce osservazioni univoche in xe y.
- **setdiff**(x, y): restituisce le osservazioni in x, ma non in y.

ESERCIZIO- Base R or dplyr?

Scrivere i pacchetti di R di base che simulino pacchetto diplyr e con comandi in R

Data transformation in base R. Read chapters from 5.18 to 5.31 in book T11

http://users.dimi.uniud.it/~massimo.franceschet/ds/r4ds/syllabus/make/dplyr/dplyr.html

Suggerimenti

- subset to filter rows and select columns;
- order to arrange rows; permutazione ordinata di una dataframe risp una variabile
- transform to add variables; aggiunge nuove variabili
- aggregate to group rows; serve per fare raggruppamenti
- merge to join data frames.

Aggiungere la chiave surrogata:

Flights= mutate(flights, id= row_number()) %>% select(id, everything())

(vedi file R)

Soluzione esercizi da rifare!

http://users.dimi.uniud.it/~massimo.franceschet/ds/r4ds/syllabus/make/dplyr/dplyr.html

DA LEGGERE

https://stackoverflow.com/questions/1299871/how-to-join-merge-data-frames-inner-outer-left-right

5 LINGUAGGI **MODI**

INDEX 1 REG 294 (riascoltare)

Il SEGUITO TRATTO DA

http://users.dimi.uniud.it/~massimo.franceschet/ds/r4ds/syllabus/learn/database/db4ds.html

Data base and data science

Il modello concettuale entità-relazione, il modello relazionale e SQL sono gli strumenti di database necessari per il data scientist.

Dobbiamo conoscerli