



Informatica

Principi di programmazione ed analisi dati in linguaggio
Scienze Biologiche



Dr. Bruno Bellisario, PhD

LAVORARE CON I DATASET

INDICIZZAZIONE

Per indicizzazione intendiamo tutti i metodi che consentono di selezionare e manipolare i dati utilizzando indici logici, interi o con nome. Poiché le capacità di indicizzazione sono importanti per la pulizia dei dati, esaminiamo rapidamente **vettori**, **data.frame** e tecniche di indicizzazione.

La variabile più elementare in R è un vettore. Un vettore è una sequenza di valori dello stesso tipo. Tutte le operazioni di base in R agiscono sui vettori. I tipi di base in R sono i seguenti.

Numerico - Dati numerici (approssimazioni dei numeri reali, \mathbb{R})

Intero - Dati interi (numeri interi, \mathbb{Z})

Fattore - Dati categorici (classificazioni semplici, come il genere)

Ordinato - Dati ordinali (classificazioni ordinate, come il livello di istruzione)

Carattere - Dati carattere (stringhe)

Raw - Dati binari



Tutte le operazioni di base in R funzionano per elemento su vettori in cui l'argomento più breve viene riciclato se necessario. Questo vale per operazioni aritmetiche (addizione, sottrazione,...), operatori di confronto (==, <=,...), operatori logici (&, |, !,...) e funzioni matematiche di base come **sin**, **cos**, **exp** e così via.

La “regola del riciclo” in R

Se si sommano due vettori di diversa lunghezza in R, il vettore più corto viene ripetuto tante volte fino a raggiungere la lunghezza del vettore di maggior lunghezza.

Es:

```
> x <- c(2,4)
> y <- c(3,5,7,9)
> x + y # il vettore x viene ripetuto due # volte (regola del riciclo)
[1] 5 9 9 13
La somma precedente equivale cioè a:
> xx <- c(x,x) # duplicazione del vettore x > xx + y
[1] 5 9 9 13
```



```
dens <- density(x)
dx <- dens$dx
dv <- dens$y
```

La “regola del riciclo” in R

La regola vale in generale in R, anche per altre strutture dati e per altre operazioni.

Es:

```
> x <- c(2,4)
> xx <- c(x,x)
> y <- c(3,5,7,9)
> xx * y
[1] 6 20 14 36
> x * y # il vettore x viene ripetuto due # volte (regola del riciclo)
[1] 6 20 14 36
```



Gli elementi di un vettore possono essere selezionati o sostituiti utilizzando l'operatore parentesi quadra [].

Le parentesi quadre accettano un vettore di nomi, numeri di indice o un logico. Nel caso di un logico, l'indice viene riciclato se è più corto del vettore indicizzato.

Nel caso di indici numerici, gli indici negativi omettono, invece di selezionare gli elementi.

Gli indici negativi e positivi non sono consentiti nello stesso vettore di indice.

È possibile ripetere un nome o un numero di indice, che risulta in più istanze dello stesso valore.

```
x <- c("red", "green", "blue")
capColor = c(huey = "red", duey = "blue", louie = "green")
capColor["louie"]
names(capColor)[capColor == "blue"]
x <- c(4, 7, 6, 5, 2, 8)
I <- x < 6
J <- x > 7
x[I | J]
x[c(TRUE, FALSE)]
x[c(-1, -2)]
```



Un elenco è una generalizzazione di un vettore in quanto può contenere oggetti di tipi diversi, inclusi altri elenchi. Ci sono due modi per indicizzare un elenco.

L'operatore parentesi singola restituisce sempre un sotto-elenco dell'elenco indicizzato. Cioè, il tipo risultante è di nuovo un elenco.

L'operatore a doppia parentesi ([[]]) può risultare solo in un singolo elemento e restituisce l'oggetto nell'elenco stesso.

Oltre all'indicizzazione, l'operatore dollaro \$ può essere utilizzato per recuperare un singolo elemento.

```
L <- list(x = c(1:5), y = c("a", "b", "c"), z = capColor)
L[[2]]
L$y
L[c(1, 3)]
L[c("x", "y")]
L[["z"]]
```



Un **data.frame** non è molto più di un elenco di vettori, possibilmente di tipi diversi, ma con ogni vettore (colonne) della stessa lunghezza.

L'operatore **\$** restituisce un vettore, non un **sub-data.frame**.

Le righe possono essere indicizzate utilizzando due indici nell'operatore parentesi, separati da una virgola. Il primo indice indica le righe, il secondo indica le colonne.

Se uno degli indici viene omesso, non viene effettuata alcuna selezione (quindi viene restituito tutto).

La selezione di una singola colonna utilizzando un risultato a due indici restituirà un vettore. Questo comportamento può essere disattivato utilizzando **drop=FALSE** come parametro aggiuntivo.

```
d <- data.frame(x = 1:10, y = letters[1:10], z = LETTERS[1:10])
d[1]
d[, 1]
d[, "x", drop = FALSE]
d[c("x", "z")]
d[d$x > 3, "y", drop = TRUE]
d[d$x > 3, "y", drop = FALSE]
d[2, ]
```



dens <- density(data) dy <- dens\$y if(add == TRUE) print(y[1.]) else dy

Valori speciali)

Come la maggior parte dei linguaggi di programmazione, R ha un numero di valori speciali che sono eccezioni ai normali valori di un tipo. Questi sono **NA**, **NULL**, **±Inf** e **NaN**.

NA sta per non disponibile (*not available*). NA è un segnaposto per un valore mancante. Tutte le operazioni di base in R gestiscono NA senza arresti anomali e per lo più restituiscono NA come risposta ogni volta che uno degli argomenti di input è NA.

```
NA + 1  
sum(c(NA, 1, 2))  
median(c(NA, 1, 2, 3), na.rm = TRUE)  
length(c(NA, 2, 3, 4))  
3 == NA  
NA == NA  
TRUE | NA
```

na.rm è una funzione che, in alcuni casi, permette di ignorare i valori NA in alcune operazioni.

Proviamo la seguente funzione:

```
median(c(NA, 1, 2, 3), na.rm = FALSE)
```



```
dens <- density(data, n=NULL npts)
```

```
dx <- dens$
```

Possiamo considerare NULL come un'insieme vuoto.

```
dy <- dens$
```

if(add == TRUE)

NULL è speciale poiché non ha classi di appartenenza (la sua classe è NULL) e ha lunghezza 0, quindi non occupa spazio in un vettore.

```
if(orientati
```

NULL è un oggetto e viene restituito quando un'espressione o una funzione restituisce un valore non definito. Nel linguaggio R, NULL è una parola riservata e può anche essere il prodotto dell'importazione di dati di tipologia sconosciuta.

```
nulldata=c(1, 2, NULL, 4)
length(nulldata)
sum(nulldata)
x <- NULL
c(x, 2)

if(Fill == T)
  confshade(dx2, seqbelow, dy2)
```



```
dens <- density(data, nInf npts)
```

```
dx <- dens$x
```

Inf sta per infinito e si applica solo ai vettori di classe numerica.

Un vettore di classe intero non può mai essere Inf. Questo perché Inf in R deriva direttamente dallo standard internazionale per l'aritmetica in virgola mobile. Tecnicamente, Inf è un valore numerico valido che risulta da calcoli come la divisione di un numero per zero.

Poiché Inf è un valore numerico, le operazioni tra Inf e un numero finito sono ben definite e gli operatori di confronto funzionano come previsto.

Un calcolo produrrà Inf o -Inf quando il risultato è un numero troppo grande per essere gestito dalla memoria di R.

```
pi/0  
2 * Inf  
Inf - 1e+10  
Inf + Inf  
3 < -Inf  
Inf == Inf
```

```
dy2 <- (dx - min(dx)) / max(dy)  
y[1.]  
seqbelow <- rep(y[1.], length(dx))  
if(Fill == T)  
    confshade(dx2, seqbelow, dy2)
```



NaN

NaN sta per non numerico.

Questo è generalmente il risultato di un calcolo di cui non si conosce l'esito, ma sicuramente non è un numero. In particolare operazioni come 0/0, Inf-Inf e Inf/Inf danno come risultato NaN.

Tecnicamente, NaN è di classe numerica, il che può sembrare strano poiché è usato per indicare che qualcosa non è numerico. I calcoli che coinvolgono numeri e NaN risultano sempre in NaN.

Differenze tra NA e NaN in R

NaN ("Non è un numero") significa 0/0

NA ("Non disponibile") è generalmente interpretato come un valore mancante e ha varie forme:
NA_integer_, NA_real_, ecc.

Pertanto, NaN ≠ NA e c'è bisogno di NaN e NA.

is.na() restituisce VERO sia per NA che NaN, tuttavia **is.nan()** restituisce VERO per NaN (0/0) e FALSO per NA.

NA significa che l'errore era **già presente** quando hai importato il foglio di calcolo in R. NaN significa che hai causato l'errore **dopo** aver importato i dati.

Rilevamento e localizzazione degli errori

Valori mancanti

Un valore mancante, rappresentato da NA in R, identifica un dato di cui è noto il tipo ma il cui valore non lo è. Pertanto, è impossibile eseguire analisi statistiche su dati in cui mancano uno o più valori nei dati. Si può scegliere di omettere elementi da un set di dati che contengono valori mancanti o di attribuire un valore, ma la mancanza è qualcosa da affrontare prima di qualsiasi analisi.

Spesso, molti software confondono un valore mancante con un valore o una categoria di default. Ad esempio, in Excel, il risultato dell'aggiunta del contenuto di un campo contenente il numero 1 con un campo vuoto risulta pari ad 1.

Questo comportamento è decisamente indesiderato poiché Excel imputa silenziosamente "0" dove avrebbe dovuto dire qualcosa sulla falsariga di "impossibile calcolare". Dovrebbe spettare all'analista decidere come gestire i valori vuoti, poiché un'attribuzione predefinita può produrre risultati imprevisti o errati per motivi difficili da rintracciare.



```
dens <- density(data[, n - points])
dx <- dens$x
dy <- dens$y
min_dx <- min(dx)
max_dx <- max(dx)
ymin <- min(dy)
ymax <- max(dy)
dx2 <- (dx - min_dx) / (max_dx - min_dx) * 100
ymin2 <- (ymin - min(dy)) / (max(dy) - min(dy)) * 100
y[1.]
seqbelow <- rep(y[1.], length(dx))
if(Fill == T)
  confshade(dx2, seqbelow, dy2)
```

Rilevamento e localizzazione degli errori

Valori mancanti

Un altro errore comune è quello di confondere un NA nei dati categorici con la categoria sconosciuta.

Se sconosciuto è effettivamente una categoria, dovrebbe essere aggiunto come livello di fattore in modo che possa essere adeguatamente analizzato.

Il comportamento della funzionalità principale di R è coerente con l'idea che l'analista debba decidere cosa fare con i dati mancanti. Una scelta comune, vale a dire "lasciare fuori i record con dati mancanti" è supportata da molte funzioni di base attraverso l'opzione **na.rm**

Funzioni come **sum**, **prod**, **quantile**, **sd** e così via hanno tutte questa opzione. Le funzioni che implementano statistiche bivariate come **cor** e **cov** offrono opzioni per includere valori completi o completi a coppie.

Oltre alla funzione **is.na**, R implementa di default alcune altre funzioni che facilitano la gestione di NA.

La funzione **complete.cases** rileva le righe in un **data.frame** che non contengono alcun valore mancante.



Rilevamento e localizzazione degli errori

Valori speciali

I calcoli che coinvolgono valori speciali spesso danno luogo a valori speciali e poiché un'affermazione statistica su un fenomeno reale **non dovrebbe** mai includere un valore speciale, è consigliabile gestire valori speciali prima dell'analisi.

Per le variabili numeriche, i valori speciali indicano i valori che non sono un elemento dell'insieme matematico dei numeri reali.

La funzione **is.finite** determina quali valori sono valori ‘normali’.

```
dy2 <- (dx - min(dx)) / max(dx) * dy
y[1.]
seqbelow <- rep(y[1.], length(dx))
if(Fill == T)
  confshade(dx2, seqbelow, dy2)
```



MANIPOLAZIONE DEI DATI IN R

Introduzione alla library **dplyr**

Data transformation with dplyr :: CHEAT SHEET



dplyr functions work with pipes and expect **tidy data**. In tidy data:



Each **variable** is in its own **column**

&

A/B/C

Each **observation**, or **case**, is in its own **row**

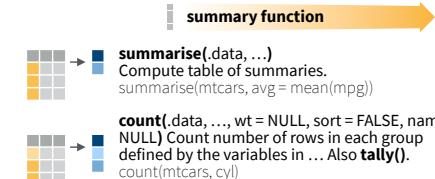


pipes

`x %>% f(y)` becomes `f(x, y)`

Summarise Cases

Apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).



Group Cases

Use `group_by(data, ..., .add = FALSE, .drop = TRUE)` to create a "grouped" copy of a table grouped by columns in ... dplyr functions will manipulate each "group" separately and combine the results.



Use `rowwise(data, ...)` to group data into individual rows. dplyr functions will compute results for each row. Also apply functions to list-columns. See tidyR cheat sheet for list-column workflow.

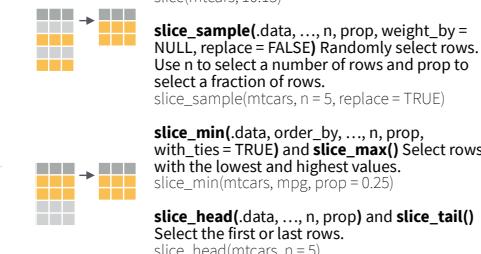


`ungroup(x, ...)` Returns ungrouped copy of table.
`ungroup(g_mtcars)`

Manipulate Cases

EXTRACT CASES

Row functions return a subset of rows as a new table.



Logical and boolean operators to use with filter()

<code>==</code>	<code><</code>	<code><=</code>	<code>is.na()</code>	<code>%in%</code>	<code> </code>	<code>xor()</code>
<code>!=</code>	<code>></code>	<code>>=</code>	<code>!is.na()</code>	<code>!</code>	<code>&</code>	

See `?base::Logic` and `?Comparison` for help.

ARRANGE CASES



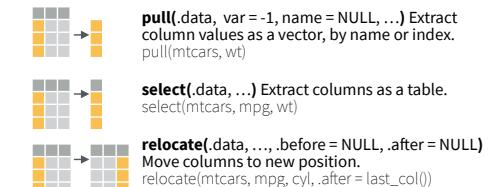
ADD CASES



Manipulate Variables

EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.



Use these helpers with select() and across()

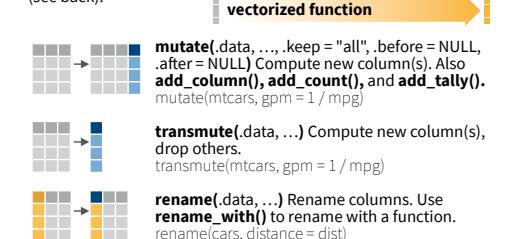
e.g. <code>select(mtcars, mpg:cyl)</code>	<code>contains(match)</code>	<code>num_range(prefix, range)</code>	; e.g. <code>mpg:cyl</code>
	<code>ends_with(match)</code>	<code>all_of(x)/any_of(x, ..., vars)</code>	; e.g. <code>-gear</code>
	<code>starts_with(match)</code>	<code>matches(match)</code>	<code>everything()</code>

MANIPULATE MULTIPLE VARIABLES AT ONCE



MAKE NEW VARIABLES

Apply **vectorized functions** to columns. Vectorized functions take vectors as input and return vectors of the same length as output (see back).



`dplyr` introduce una diversa grammatica della manipolazione dei dati in R. Fornisce un'interfaccia coerente per lavorare con i dati, indipendentemente da come sono archiviati: `data.frame`, `data.table` o un database. Le parti chiave di `dplyr` sono scritte usando `Rcpp`, questa caratteristica rende molto veloce l'elaborazione dei dati caricati in memoria.

La filosofia di `dplyr` è quella di avere piccole funzioni che facciano bene una cosa.

I comandi sono facili da ricordare perché richiamano le azioni che i comandi andranno a svolgere. Inoltre è possibile concatenare una serie di funzioni grazie all'utilizzo dell'operatore **pipe** `%>%`. Questa nuova sintassi permette di rendere il codice più semplice da leggere. La libreria ha 7 funzioni principali che vengono mostrate in tabella insieme all'equivalente comando in **SQL**:

Funzione	Descrizione	Equivalenza SQL
<code>select()</code>	Selecting columns (variables)	SELECT
<code>filter()</code>	Filter (subset) rows.	WHERE
<code>slice()</code>	Choose rows (by position)	-
<code>group_by()</code>	Group the data	GROUP BY
<code>summarise()</code>	Summarise (or aggregate) data	-
<code>arrange()</code>	Sort the data	ORDER BY
<code>join()</code>	Joining data frames (tables)	JOIN
<code>mutate()</code>	Creating New Variables	COLUMN ALIAS



```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
```

- Uno dei vantaggi di queste funzioni è che il risultato di ciascun comando sarà sempre un nuovo dataframe. In questo modo sarà molto più semplice eseguire una serie manipolazioni in sequenza.
- Una caratteristica aggiuntiva è la capacità di lavorare con i dati memorizzati direttamente in un database esterno. I vantaggi di fare questo sono che i dati possono essere gestiti in modo nativo in un database relazionale, le query possono essere condotte su quel database e solo i risultati della query restituiti nell'ambiente di lavoro.
- Le colonne possono essere specificate direttamente usando nomi delle variabili senza usare \$ o le virgolette.

```
uy<-c(x=1:10,y=c(1,2,3,4,5,6,7,8,9,10))
y[1.]
seqbelow <- rep(y[1.], length(dx))
if(Fill == T)
  confshade(dx2, seqbelow, dy2)
```

La libreria dplyr

Iniziamo con l'installazione e il caricamento della libreria `dplyr`:

```
install.packages("dplyr")
library(dplyr)
```

Utilizzeremo il famoso dataset `iris`:

```
data("iris")
```

Select

Il comando `select()` permette di selezionare una o più variabili del dataframe. Per selezionare alcune colonne è sufficiente aggiungere i nomi delle variabili come argomenti di `select`. L'ordine in cui vengono aggiunti, determinerà l'ordine in cui verranno visualizzati nell'output.

```
select(iris[1:5], Sepal.Length, Sepal.Width) == 1)
```

Mutate

È possibile creare nuove colonne utilizzando quelle del dataframe con la funzione `mutate()`. Le opzioni utilizzabili all'interno dell'istruzione sono quasi infinite: praticamente tutto ciò che si può fare per i vettori, può essere fatto all'interno della funzione `mutate()`. Solitamente per una nuova colonna si utilizza un nuovo nome ma se viene utilizzato il nome di una colonna già esistente essa viene sostituita.

```
mutate(iris[1:5], Sepal.LWratio = Sepal.Length/
Sepal.Width)
```

Filter

In molti casi non si vogliono includere tutte le righe nell'analisi, ma solo una selezione. La funzione `filter()` riduce le righe/osservazioni in base alle condizioni assegnate. La sintassi generale del filtro è: `filter(dataset, condizione)`. È possibile filtrare le variabili numeriche in base ai rispettivi valori. Gli operatori più utilizzati per questo sono:

- `>` (maggiore)
- `>=` (maggiore uguale)
- `<` (minore)
- `<=` (minore uguale)
- `==` (uguale)
- `!=` (diverso)

```
filter(iris, Sepal.Width > 3.7)
```



Filter

```
dens <- density(data, n = Slice)
dx <- dens$x
```

In molti casi non si vogliono includere tutte le righe nell'analisi, ma solo una selezione. La funzione `filter()` riduce le righe/osservazioni in base alle condizioni assegnate. La sintassi generale del filtro è: `filter(dataset, condizione)`. È possibile filtrare le variabili numeriche in base ai rispettivi valori. Gli operatori più utilizzati per questo sono:

- `>` (maggiore)
- `>=` (maggiore uguale)
- `<` (minore)
- `<=` (minore uguale)
- `==` (uguale)
- `!=` (diverso)

```
filter(iris, Sepal.Width > 3.7)
```

Oppure per le variabili categoriche sono utilizzati i segni `==` o `!=`. Una o più condizioni possono essere usate contemporaneamente:

```
filter(iris, Species == "virginica", Sepal.Width > 3.7)
```

Slice

La funzione `slice()` consente di selezionare le righe tramite la loro posizione:

```
slice(iris, 8:12)
```

summarize

La funzione `summarize()` permette di calcolare funzioni statistiche riassuntive. Per utilizzare la funzione è sufficiente aggiungere il nome nuovo della colonna `e`, dopo il segno di uguale, la funzione matematica con cui vogliamo riassumere i dati `column_name = funzione(variabile)`. È possibile aggiungere più funzioni di riepilogo all'interno dell'istruzione `summarize()`.

Il codice seguente mostra il calcolo della funzione `mean` e `sd` in riferimento alla variabile `Sepal.Length`:

```
summarize(iris, mean_sl = mean(Sepal.Length), sd_sl=sd(Sepal.Length))
```

Operatore PIPE: %>%

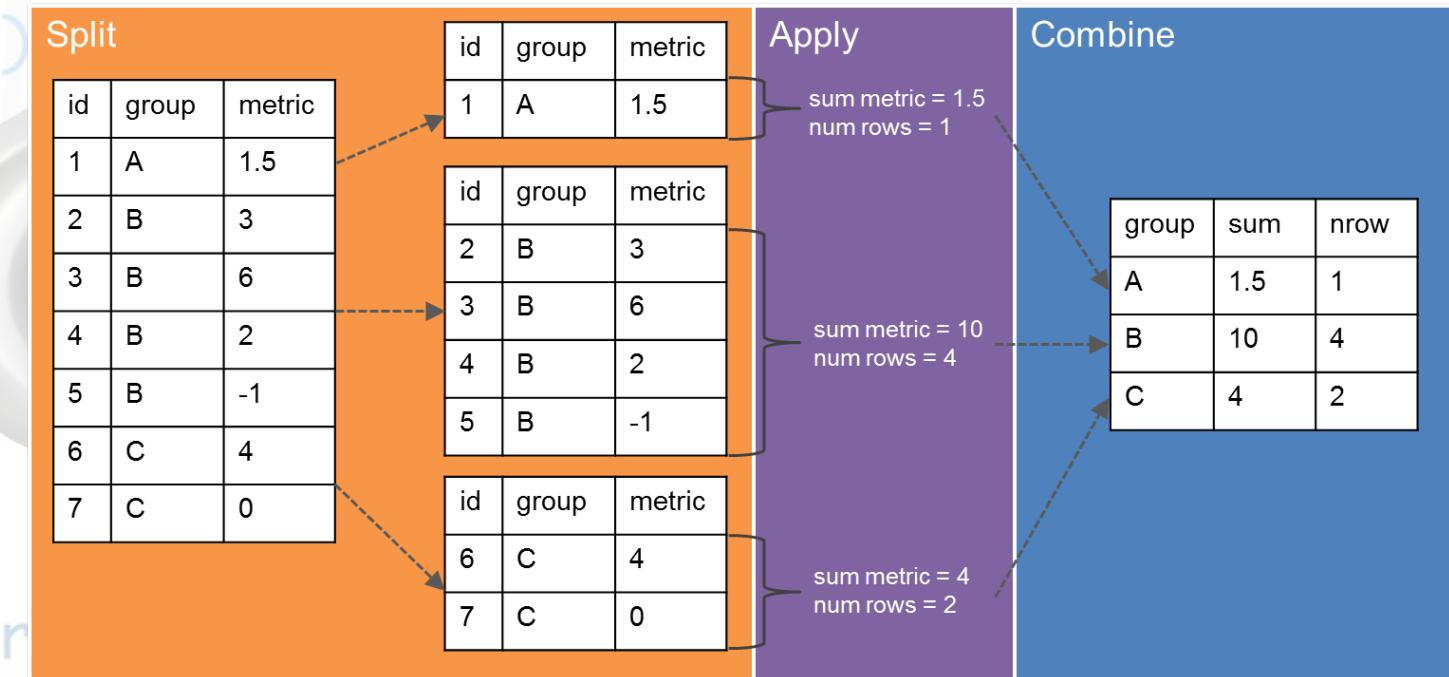
L'operatore **pipe**, `%>%` permette di concatenare facilmente una sequenza di funzioni. Quando la **pipe** è presente tra due funzioni esempio: `fun1() %>% fun2()` questa sequenza può essere tradotta con **esegui fun1()** poi, con **output di fun1(), esegui fun2()**. Tale comando è mostrato nella slide successiva in cui `group_by()` e `summarize()` sono usati in successione. Per richiamare l'operatore **pipe** dalla tastiera è possibile utilizzare la shortcut di RStudio:

Ctrl + Shift + M Windows

Cmd + Shift + M Mac

split-apply-combine

Molte delle attività di analisi dati possono essere affrontate utilizzando il paradigma *split-apply-combine*: dividere i dati in gruppi, applicare alcune analisi a ciascun gruppo e combinare i risultati in un nuovo data frame.



Il pacchetto `dplyr` è stato scritto appositamente per ottimizzare le analisi di tipo Split-Apply-Combine.

La funzione `group_by()` raggruppa i dati utilizzando i livelli di una variabile categorica ed è utilizzata insieme alla funzione `summarize` per fornire statistiche a riguardo i diversi gruppi. In questo esempio le funzioni sono concatenate con l'operatore `%>%`:

```
iris %>%
  group_by(Species) %>%
  summarize(mean_sl=mean(Sepal.Length))
```

arrange

La funzione `arrange()` ordina le righe in modo crescente:

```
iris %>%
  arrange(Sepal.Length) %>%
  head()
```

Per ordinare in modo decrescente utilizzare la funzione `desc()` come mostrato nell'esempio:

```
iris %>%
  arrange(desc(Sepal.Length)) %>%
  head()
```



join

Le funzioni `join` permettono di unire due tabelle tramite una colonna in comune, chiamata anche key. Quando si unisce due tabelle si definisce *tabella di sinistra* la prima tabella che viene codificata e *tabella di destra* la seconda. I nomi delle funzioni richiamano i rispettivi comandi utilizzati in SQL.

Per comprendere il funzionamento delle funzioni della famiglia `join` facciamo riferimento agli esempi riportati nell'immagine di fianco.

Nell'esempio riportato la colonna in comune è chiamata `x1`. La tabella risultante dall'unione conterrà la colonna `x1` e le altre colonne delle tabelle unite.

Combine Data Sets

a		b	
x1	x2	x1	x3
A	1	A	T
B	2	B	F
C	3	D	T

+

a		b	
x1	x2	x1	x3
A	1	A	T
B	2	B	F
C	3	D	T

=

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::left_join(a, b, by = "x1")
Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::right_join(a, b, by = "x1")
Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

dplyr::inner_join(a, b, by = "x1")
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F

dplyr::full_join(a, b, by = "x1")
Join data. Retain all values, all rows.

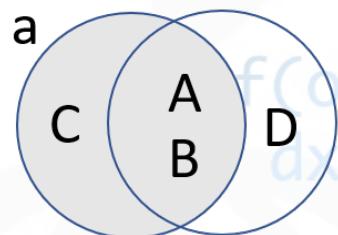
In molti casi la colonna in comune non è perfettamente coincidente tra le due tabelle. Infatti la tabella di destra può contenere valori di `x1` assenti nella tabella di sinistra e viceversa.



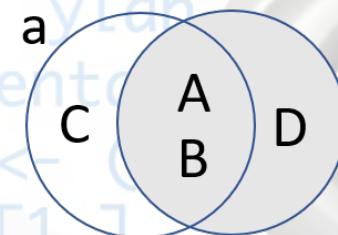
Il meccanismo di unione dipende da come si selezionano i valori della colonna `x1` che andranno a costituire la tabella finale. Per questo sono possibili i seguenti meccanismi:

dplyr joins

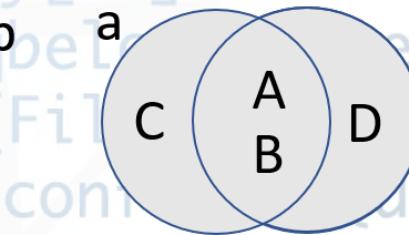
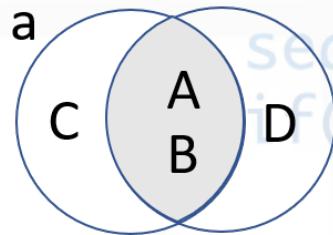
left_join(a,b)



right_join(a,b)



inner_join(a,b) **full_join(a,b)**



- `left_join()`: estrae tutti i valori della tabella a sinistra anche se non hanno corrispondenza nella tabella a destra;
- `right_join()` estrae tutti i valori della tabella a destra anche se non hanno corrispondenza nella tabella di sinistra.
- `inner_join()`: il suo scopo è quello di unire due tabelle restituendo un risultato combinato sulla base di uno o più osservazioni che trovano corrispondenza in tutte le tabelle coinvolte nella join. Il comando corrispondente è
- `full_join()`: estrae tutte le righe delle due tabelle.



Vedremo adesso il loro funzionamento utilizzando i seguenti data set:

- band_members
- band_instruments

I due dataset hanno una chiave in comune, la colonna name.

```
band_members$name
```

```
band_instruments$name
```

Il comando `left_join` estrae tutti i valori della colonna name contenuti nella tabella di sinistra, nel caso che segue `band_members`

```
left_join(band_members,band_instruments,by="name")
```

Il comando `right_join` estrae tutti i valori della colonna name contenuti nella tabella di destra, nel caso che segue `band_instruments`

```
right_join(band_members,band_instruments,by="name")
```

Il comando `full_join` estrae tutti i valori della colonna name contenuti di entrambe le tabelle:

```
full_join(band_members,band_instruments,by="name")
```

Il comando `inner_join` estrae solo i valori della colonna name che sono contemporaneamente contenuti in entrambe le tabelle:

```
inner_join(band_members,band_instruments,by="name")
```

Tidy data

Introduzione

Le famiglie felici sono tutte uguali; ogni famiglia infelice è infelice a modo suo

Leo Tolstoy

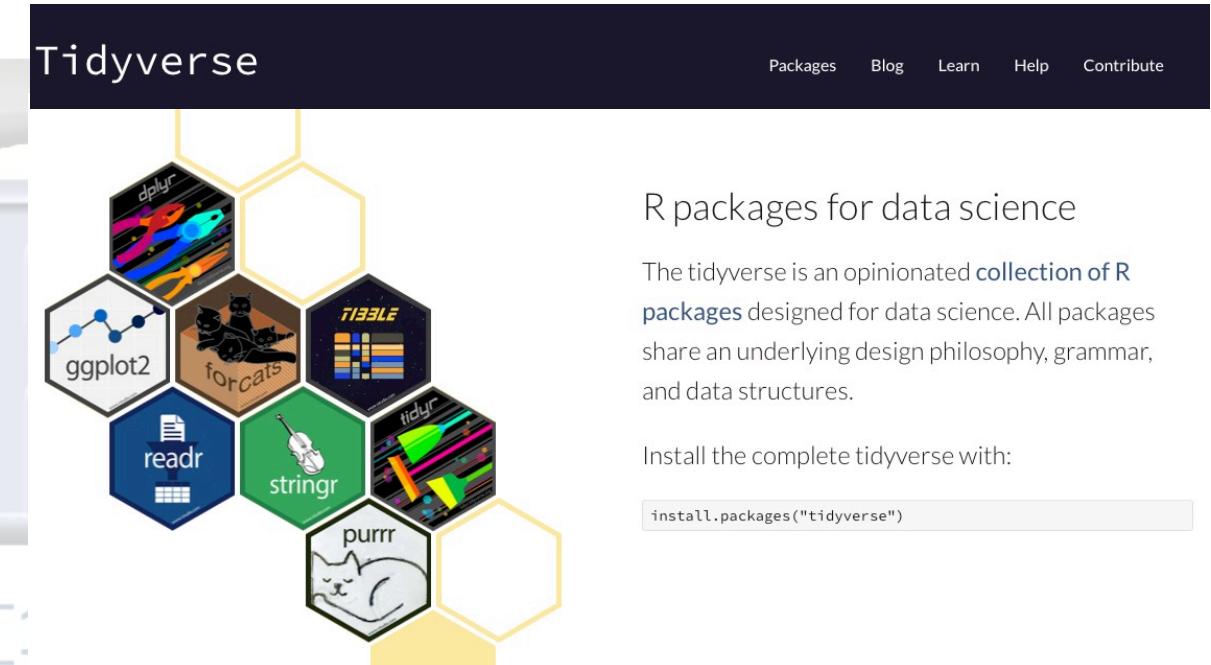
Le serie di dati ordinate sono tutte uguali, ma ogni serie di dati disordinata è disordinata a modo suo

Hadley Wickham

Prerequisiti

Ci concentreremo su *tidyverse*, un pacchetto che fornisce una serie di strumenti per aiutare a riordinare i vostri insiemi di dati disordinati. *tidyverse* è un membro del core [tidyverse](#).

```
library(tidyverse)
```



R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```



Tidy data

Normalmente è possibile rappresentare gli stessi dati in più modi. Le tabelle sottostanti mostrano gli stessi valori di quattro variabili (*paese*, *anno*, *popolazione*, e *casi*), ma ogni set di dati organizza i valori in un modo diverso.

table1

table2

table3

table4

Ci sono tre regole correlate che rendono un dataset ordinato:

1. Ogni variabile deve avere la propria colonna.
2. Ogni osservazione deve avere la sua riga.
3. Ogni valore deve avere la propria cella.

country	year	cases	population
Afghanistan	1990	45	193071
Afghanistan	2000	5666	20595360
Brazil	1999	37737	17206362
Brazil	2000	80488	17404898
China	1999	210258	1272015272
China	2000	210566	128042583

variables

country	year	cases	population
Afghanistan	1990	45	193071
Afghanistan	2000	5666	20595360
Brazil	1999	37737	17206362
Brazil	2000	80488	17404898
China	1999	210258	1272015272
China	2000	210566	128042583

observations

country	year	cases	population
Afghanistan	1990	45	193071
Afghanistan	2000	5666	20595360
Brazil	1999	37737	17206362
Brazil	2000	80488	17404898
China	1999	210258	1272015272
China	2000	210566	128042583

values

Queste tre regole sono interrelate perché è impossibile soddisfare solo due delle tre. Questa interrelazione porta ad un insieme ancora più semplice di istruzioni pratiche:

1. Metti ogni set di dati in una **tibble**.
2. Metti ogni variabile in una colonna.

In questo esempio, solo la `table1` è ordinata. È l'unica rappresentazione in cui ogni colonna è una variabile.

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
```

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

1. C'è un vantaggio generale nel scegliere un modo coerente di memorizzare dati. Se hai una struttura di dati coerente, è più facile imparare gli strumenti che lavorano con essa perché hanno un'uniformità di fondo.
 2. C'è un vantaggio specifico nel mettere le variabili in colonne perché permette alla natura vettoriale di R di operare al meglio. Infatti, la maggior parte funzioni *built-in* di R lavorano con i vettori.

`dplyr`, `ggplot2` e tutti gli altri pacchetti del *tidyverse* sono progettati per lavorare con dati ordinati.

dens <- density(data, Pivoting pts)

I principi dei *tidy-data* sembrano così ovvi che viene naturale chiedersi se incontreremo mai un set di dati che non sia ordinato. Sfortunatamente, però, la maggior parte dei dati che incontreremo saranno **SEMPRE** disordinati. 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.

I dati sono spesso organizzati per facilitare usi diversi dall'analisi. Per esempio, i dati sono spesso organizzati per rendere l'inserimento il più facile possibile.

Questo significa che per la maggior parte delle analisi reali, dovremmo fare un po' di ordine. Il primo passo è sempre quello di capire quali sono le variabili e le osservazioni. A volte questo è facile; altre volte si avrà bisogno di consultare le persone che hanno originariamente generato i dati. Il secondo passo è quello di risolvere uno dei due problemi comuni:

Una variabile potrebbe essere distribuita su più colonne.

Un'osservazione potrebbe essere sparsa su più righe.

Tipicamente un set di dati soffrirà solo di uno di questi problemi; soffrirà di entrambi solo se siete davvero sfortunati!

Per risolvere questi problemi, possiamo utilizzare due delle funzioni più importanti di *tidyr*: `pivot_longer()` e `pivot_wider()`.

```
dens <- density(data, ..., na.rm = TRUE)
dx <- dens$x
```

Un problema comune è il trovarsi di fronte ad un dataset dove alcuni dei nomi delle colonne non sono nomi di variabili, ma *valori* di una variabile. Prendiamo come esempio la `table4a`: i nomi delle colonne `1999` e `2000` rappresentano i valori della variabile `year`, i valori nelle colonne `1999` e `2000` rappresentano i valori della variabile `cases`, e ogni riga rappresenta due osservazioni, non una.

table4a

```
#> # A tibble: 3 × 3
#>   country `1999` `2000`
#>   <chr>     <dbl>   <dbl>
#> 1 Afghanistan    745    2666
#> 2 Brazil        37737   80488
#> 3 China         212258  213766
```

Per riordinare un set di dati come questo, abbiamo bisogno di una funzione di **pivoting**. Per descrivere questa operazione abbiamo bisogno di tre parametri:

- L'insieme delle colonne i cui nomi sono valori, non variabili. In questo esempio, queste sono le colonne `1999` e `2000`.
- Il nome della variabile in cui spostare i nomi delle colonne. Qui è `year`.
- Il nome della variabile in cui spostare i valori della colonna. Qui è `case`.

Insieme questi parametri generano la chiamata a `pivot_longer()`:

```
table4a %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to =
  "cases")
```

```
dens <- density(data, n = npts)
```

Le colonne per fare il **pivoting** sono specificate con la notazione in stile `dplyr::select()`. Qui ci sono solo due colonne, quindi le elenchiamo singolarmente. Notate che “1999” e “2000” sono nomi non sintattici (perché non iniziano con una lettera), quindi dobbiamo circondarli di *backtick*.

year e cases non esistono in `table4a` quindi mettiamo i loro nomi tra virgolette.

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

Nel risultato finale, le colonne ‘pivotate’ vengono eliminate e si ottengono nuove colonne year e cases. Le relazioni tra le variabili originali sono conservate.

La funzione `pivot_longer()` rende i set di dati più lunghi aumentando il numero di righe e diminuendo il numero di colonne.



Possiamo usare `pivot_longer()` per riordinare `table4b` in modo simile. L'unica differenza è la variabile memorizzata nei valori delle celle:

```
table4b %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "population")
```

Per combinare le versioni riordinate di `table4a` e `table4b` in una singola tibble, dobbiamo usare `dplyr::left_join()`

```
tidy4a <- table4a %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases")
tidy4b <- table4b %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "population")
```

```
left_join(tidy4a, tidy4b)
```



```
dens <- density(dat)pivot_wider(s)
dx <- dens$x
```

pivot_wider() è l'opposto di **pivot_longer()**. Si usa quando un'osservazione è sparsa su più righe.

Per esempio, prendete la **table2**: un'osservazione è un paese in un anno, ma ogni osservazione è sparsa su due righe.

table2

```
#> # A tibble: 12 × 4
#>   country      year type     count
#>   <chr>        <dbl> <chr>    <dbl>
#> 1 Afghanistan  1999 cases     7451
#> 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
```

```
seqbelow <- rep(y[1.], length(dx))
if(Fill == T)
  confshade(dx2, seqbelow, dy2)
```



```
dens <- density(data, n = npts)  
dx <- dens$x
```

Per riordinare il tutto, analizziamo prima la rappresentazione in modo simile a `pivot_longer()`. Questa volta, però, abbiamo bisogno solo di due parametri:

- La colonna da cui prendere i nomi delle variabili. Qui è `type`.
- La colonna da cui prendere i valori. Qui è `count`.

Una volta capito questo, possiamo usare `pivot_wider()`

```
table2 %>%  
  pivot_wider(names_from = type, values_from =  
  count)
```

`pivot_wider()` e `pivot_longer()` sono complementari.

`pivot_longer()` rende le tabelle larghe più strette e lunghe;

`pivot_wider()` rende le tabelle lunghe più corte e larghe.

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

table2



Separare e unire

Finora abbiamo imparato come mettere in ordine table2 e table4, ma non table3.

La table3 ha un problema diverso: abbiamo una colonna (rate) che contiene due variabili (casi e popolazione). Per risolvere questo problema, avremo bisogno della funzione separate().

La funzione separate() separa una colonna in colonne multiple, dividendo ogni volta che appare un carattere separatore. Prendiamo ad esempio table3.

La colonna rate contiene entrambe le variabili cases e population, e abbiamo bisogno di dividerla in due variabili.

La funzione separate() prende il nome della colonna da separare e i nomi delle colonne in cui separare

```
table3 %>%
  separate(rate, into = c("cases",
  "population"))
```

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

table3

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

```
dens <- density(data, n = npts)
dx <- dens$x
```

Di default, separate() dividerà i valori ovunque veda un carattere non alfanumerico (cioè un carattere che non sia un numero o una lettera). Per esempio, nel codice precedente, separate() divide i valori di rate in corrispondenza dei caratteri ‘forward slash’. Se volete usare un carattere specifico per separare una colonna, potete passare il carattere all’argomento sep di separate(). Per esempio, potremmo riscrivere il codice sopra come:

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/")
```

Guardate attentamente i tipi di colonna: noterete che cases e population sono colonne di caratteri. Questo è il comportamento predefinito di separate(): lascia il tipo di colonna così com’è. Qui, tuttavia, non è molto utile, dato che sono davvero numeri. Possiamo chiedere a separate() di provare a convertire in tipi migliori usando convert = TRUE:

```
table3 %>%
  separate(rate, into = c("cases", "population"), convert = TRUE)
```



```
dens <- density(data, n = npts)
dx <- dens$x
```

Potete anche passare un vettore di interi a `sep`. `separate()` interpreterà gli interi come posizioni in cui dividere. I valori positivi partono da 1 all'estrema sinistra delle stringhe; i valori negativi partono da -1 all'estrema destra delle stringhe. Quando si usano gli interi per separare le stringhe, la lunghezza di `sep` dovrebbe essere inferiore al numero di nomi in `into`.

E' possibile usare questa disposizione per separare le ultime due cifre di ogni anno. Questo rende questi dati meno ordinati, ma è utile in altri casi.

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



```
dens <- density(data, r npts)
```

unite() è l'inverso di separate(): combina colonne multiple in una singola colonna. Servirà molto meno frequentemente di separate(), ma è comunque uno strumento utile da avere in tasca.

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

In questo caso dobbiamo anche usare l'argomento sep. Il default metterà un underscore (_) tra i valori delle diverse colonne. Qui non vogliamo nessun separatore quindi usiamo "":

```
table5 %>%
  unite(new, century, year, sep = "")
```

Possiamo usare unite() per riunire le colonne century e year che abbiamo creato nell'ultimo esempio. Questi dati vengono salvati come tidyverse::table5. unite() prende un data frame, il nome della nuova variabile da creare e un insieme di colonne da combinare, sempre specificate in stile dplyr::select():

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

dens <- density(data) Valori mancanti(s)

Cambiare la rappresentazione di un set di dati fa emergere un'importante sottigliezza sui valori mancanti. Sorprendentemente, un valore può essere mancante in uno dei due modi possibili:

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

Illustriamo questa idea con un insieme di dati molto semplice:

```
stocks <- tibble(orientati ==  
  year = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),  
  qtr = c(1, 2, 3, 4, 2, 3, 4),  
  return = c(1.88, 0.59, 0.35, NA, 0.92, 0.17, 2.66)  
)
```

Ci sono due valori mancanti in questo set di dati:

- Il rendimento per il quarto trimestre del 2015 è esplicitamente mancante, perché la cella dove dovrebbe avere il suo valore, ma contiene invece NA.
- Il rendimento per il primo trimestre del 2016 è implicitamente mancante, perché semplicemente non appare nel set di dati.

Un valore mancante esplicito è la presenza di un'assenza; un valore mancante implicito è l'assenza di una presenza. Il modo in cui un set di dati è rappresentato può rendere esplicativi i valori impliciti. Per esempio, possiamo rendere esplicito il valore mancante implicito mettendo gli anni nelle colonne:

```
stocks %>%  
  pivot_wider(names_from = year, values_from = return)
```

Poiché questi valori mancanti esplicativi potrebbero non essere importanti in altre rappresentazioni dei dati, potete impostare `values_drop_na = TRUE` in `pivot_longer()` per rendere impliciti i valori mancanti esplicativi:

```
stocks %>%  
  pivot_wider(names_from = year, values_from = return) %>%  
  pivot_longer(  
    cols = c(`2015`, `2016`),  
    names_to = "year",  
    values_to = "return",  
    values_drop_na = TRUE  
)
```

Un altro importante strumento per rendere esplicativi i valori mancanti nei dati ordinati è `complete()`:

```
stocks %>%  
  complete(year, qtr)
```

`complete()` prende un insieme di colonne e trova tutte le combinazioni uniche. Poi si assicura che il dataset originale contenga tutti quei valori, riempiendo i NA esplicativi dove necessario.



```
dens <- density(data, n = npts)
dx <- dens$x
```

C'è un altro importante strumento che è importante conoscere per lavorare con i valori mancanti. A volte, quando una fonte di dati è stata usata principalmente per l'inserimento di dati, potrebbe accadere che i campi di alcune variabili (ad esempio il nome o "person" nell'esempio sottostante) siano omessi:

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

Potete riempire questi valori mancanti con fill(). Prende un insieme di colonne in cui volete che i valori mancanti siano sostituiti dal più recente valore non mancante (a volte chiamato ultima osservazione portata avanti).

```
treatment %>%
  fill(person)
seqbelow <- rep(y[1.], length(dx))
if(Fill == T)
  confshade(dx2, seqbelow, dy2)
```

