# Informatica

## Principi di programmazione ed analisi dati in linguaggio

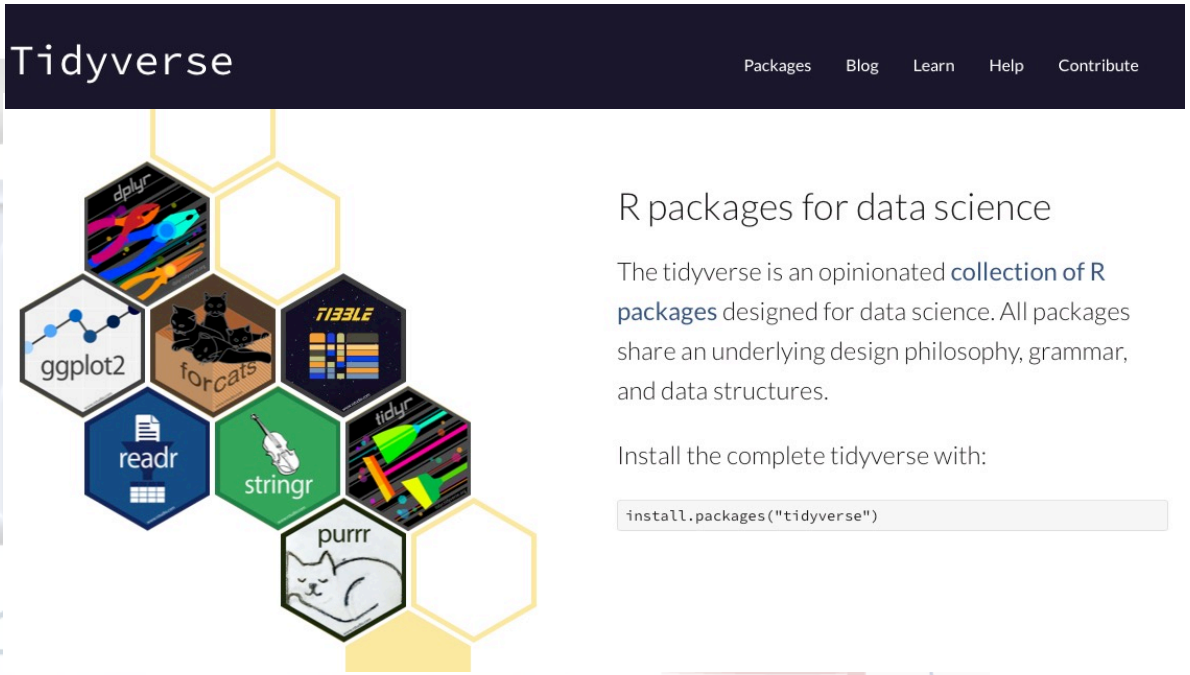
Scienze Biologiche

Dr. Bruno Bellisario, PhD

# 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 *tidyr*, un pacchetto che fornisce una serie di strumenti per aiutare a riordinare i vostri insiemi di dati disordinati. *tidyr* è un membro del core tidyverse.

```
library(tidyverse)
```

# Tidy data

## Possibili problemi legati alla installazione della libreria

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

```
WARNING: Rtools is required to build R packages but is not currently installed. Please download and
install the appropriate version of Rtools before proceeding:

https://cran.rstudio.com/bin/windows/Rtools/
Installing package into C:/Users/Ally/Documents/R/win-library/3.4
(as lib is unspecified)
also installing the dependencies lifecycle, blob, glue, tidyselect, vctrs, ellipsis, broom, dbplyr,
dplyr, haven, hms, purrr, rlang, tibble, tidyr, xml2
```

# Tidy data

## Possibili problemi legati alla installazione della libreria

**RTools** è un insieme di programmi richiesti su Windows per creare pacchetti R dal sorgente.

**Rtools** è un bundle di *toolchain* utilizzato per creare pacchetti R dal sorgente (quelli che richiedono la compilazione di codice C/C++ o Fortran) e per compilare R stesso.

Molte persone sono in grado di usare R, senza mai avere la necessità di compilare dal sorgente e, quindi, la necessità di installare **Rtools.** Tuttavia, se si desidera eseguire lo sviluppo di pacchetti, compilare dal sorgente o utilizzare specifici pacchetti (rstan o cmdstanr), sarà sicuramente necessario installare **Rtools.**

# Tidy data

## Possibili problemi legati alla installazione della libreria

**Step da seguire:**

1. Nella console di RStudio digitare il comando `R.version.string`.
2. Annotare la versione di R installata (es. **R version 4.0.2 (2020-06-22)**)
3. Chiudere RStudio.
4. Aprire la pagina https://cran.r-project.org/bin/windows/Rtools/
5. Eseguire il download della versione di **RTools** per la versione di R che abbiamo installata nel nostro computer (es. per la versione R 4.0.2, dobbiamo eseguire il download della rispettiva versione **RTools**, **RTools** 4.0).
6. Nel caso avessimo installata una versione antecedente ad R 4.x, nella pagina troviamo un link per le old version of RTools.
   **Tuttavia è sempre opportuno tenere R ed RStudio sempre aggiornati alla versione più recente supportata dal nostro SO.**
7. Uno volta effettuato il download eseguire il file .exe (è un file molto pesante e, quindi, potrebbe metterci del tempo).
8. Finito il processo di installazione avremo RTools installato nel nostro computer.
9. Riavviare la sessione di RStudio e tentiamo nuovamente l'installazione di tidyr (`install.packages("tidyr")`).

# Tidy data

## Possibili problemi legati alla installazione della libreria

**UTENTI MACOS**

Gli utenti che utilizzano piattaforme UNIX e MacOS non hanno necessità di installare RTools, essendo quest'ultimo un software esclusivo per Windows.
Tuttavia, le versioni più recenti dei pacchetti R sono spesso disponibili solo come codice sorgente. Quando installi o aggiorni il pacchetto, potrebbe indicare che necessita di compilazione. Per consentire al tuo computer di compilare il codice in questi pacchetti, devi eseguire una piccola quantità di configurazione aggiuntiva. Prima di iniziare, assicurati che R sia chiuso. Raccomando anche il riavvio dopo l'installazione, ma non è un requisito.

Su Mac, devi installare **Xcode Command Line Tools.**

Apri una finestra di terminale (clic su Ricerca Spotlight in alto a destra dello schermo, quindi cerca "Terminale")
Copia e incolla quanto segue nel terminale, quindi premi invio, **xcode-select --install**
Probabilmente dovrai fornire la tua password per abilitare l'installazione del software.
Segui le istruzioni sullo schermo e attendi che finisca.
Ora puoi compilare i pacchetti R!

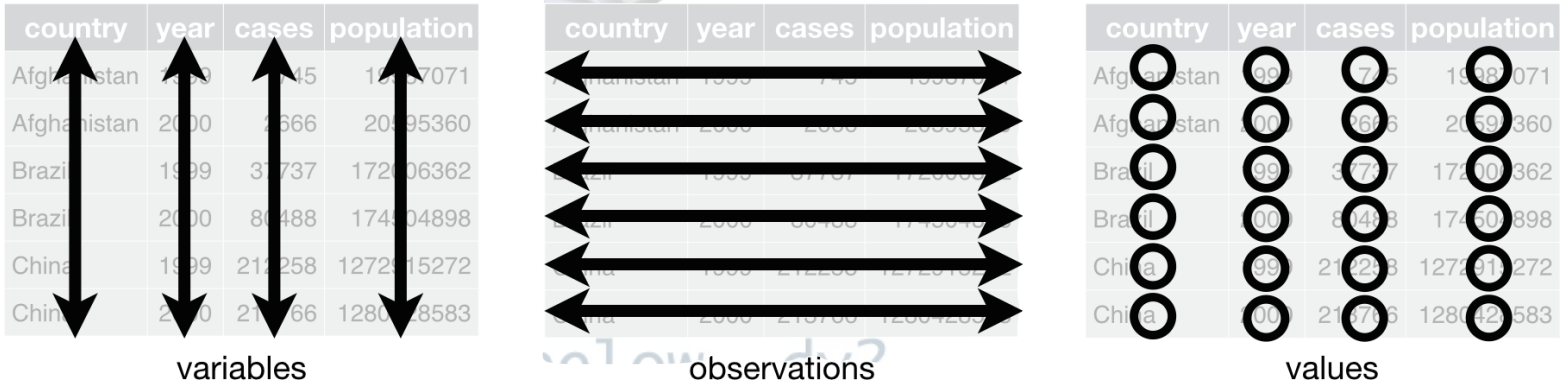
## Tidy data

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.



# Pivoting

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()`.

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.

# La programmazione è importante

## I dati di pulizia a mano sono:

Incline a errori

Richiede tempo

Non riproducibile

**Apre molte porte**

Nuove vie di ricerca

Lavori non accademici

Flussi di lavoro migliorati anche senza dati

# pivot_longer

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")
```

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 |
|---|---|---|
| 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

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)
```

# pivot_wider()

`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            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
```

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`.

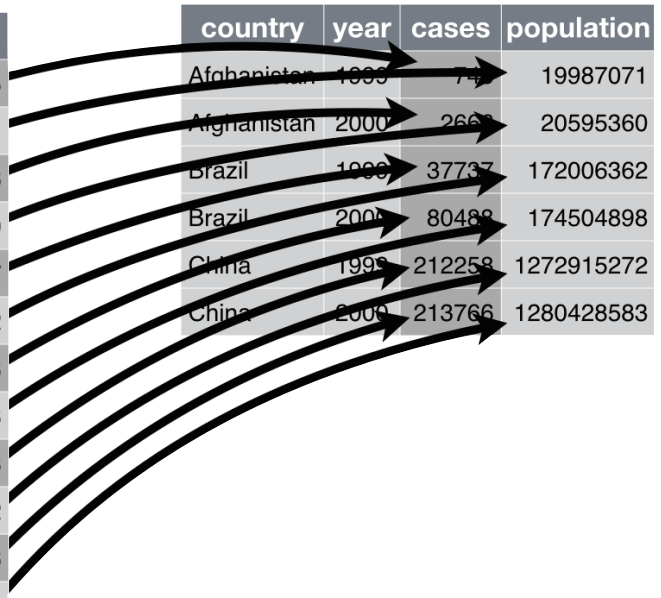
Una volta capito questo, possiamo usare `pivot_wider()`

```
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 |
|---|---|---|---|
| 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



## 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 |

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 questi valori usando `convert = TRUE`.:

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

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`.

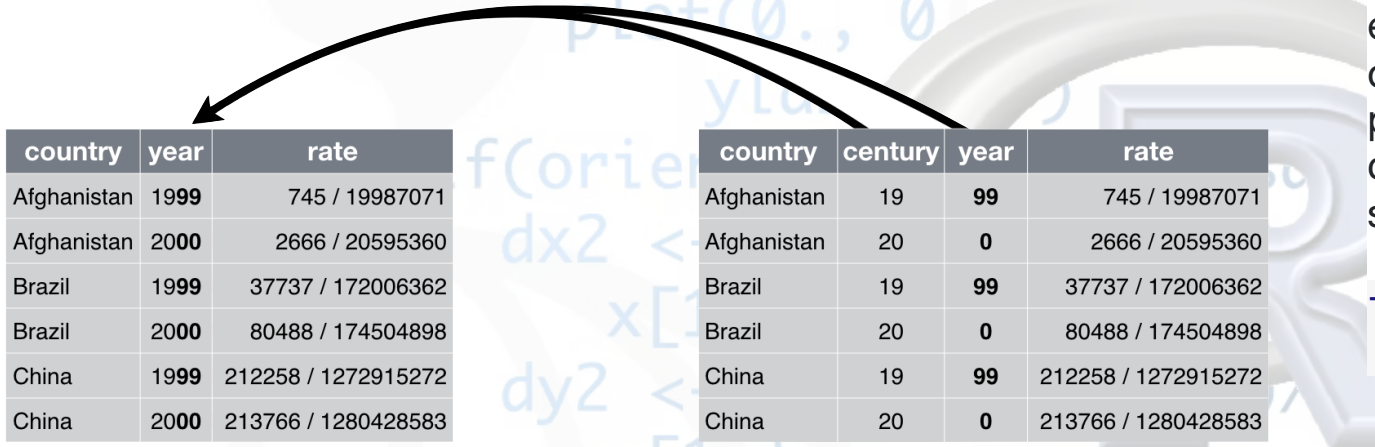
E' possibile usare questa disposizione per separare le ultime due cifre di ogni anno. Questo rende questi dati meno ordinati, ma è utile in altri casi.

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

| 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 |

# Unite

`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 |

Possiamo usare `unite()` per riunire le colonne *century* e *year* che abbiamo creato nell'ultimo esempio. Questi dati vengono salvati come `tidyr::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)
```

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 = "")
```

## Valori mancanti

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

```
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 espliciti 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` espliciti dove necessario.

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)
```

# Importare ed esportare dati

In R esistono diverse funzioni in grado di importare una dataset, a seconda del formato disponibile:

I formati più comuni sono:
**.cvs** – comma separated value –
**.txt** – file di teso –
**.xls** – file excel (vecchie e nuove versioni) –
**.xlsx** – file excel (solo nuove versioni) –
**.tsv** – tabulated separated values –

Formati specifici:
**.FASTA** – utilizzato per la memorizzazione di sequenze biologiche –

**.SPSS**
**.Stata** → **Software specifici**
**.SAS**

# Importare ed esportare dati

Per leggere dati da file R fornisce numerose funzioni, tra le quali read.table() e read.csv() che permettono di caricare file di testo **.txt** e **.csv** (*comma separated value*).
Questi file possono utilizzare differenti separatori di colonna; i principali sono la virgola, il punto e virgola e la tabulazione.
Nel caricare i dati in R occorre prestare attenzione a quale separatore viene utilizzato nel file!

```
read.table("percorso/file.txt", header = TRUE, sep = " \t", dec = ".") - Legge file .txt
read.csv("percorso/file.csv", header = TRUE, sep = " \t", dec = ".")   - Legge file .csv
```

Andiamo ad analizzarne gli argomenti:

**header** = TRUE: identifica la prima riga della matrice dei dati come quella contenente i nomi delle variabili;
**sep** = "," specifica il tipo di separatore di colonna. Tra virgolette potremmo perciò inserire la virgola, il punto e virgola e per la tabulazione **\t**.
**dec** = "." È l'argomento che indica il tipo di notazione decimale utilizzato. In notazione scientifica internazionale, il punto (.) solitamente indica il separatore dei decimali, mentre la virgola (,) indica il separatore delle migliaia……tutto il contrario rispetto alla nostra notazione.

**Notazione internazionale:** 1.3 1,234.65
**Notazione non-internazionale:** 1,3 1.234,65

# Importare ed esportare dati

I formati **.xls** e **.xlsx** sono invece formati binari, nei quali i dati non sono accessibili in chiaro, e non sono formati standardizzati per cui la struttura dei file potrebbe cambiare senza preavviso nelle nuove versioni dei programmi che salvano in questi formati, causando errori imprevedibili e pertanto non gestibili nell'importazione dei dati in R.
Per la lettura dei file Excel (**.xls** e **.xlsx**), R non ha delle *built-in functions*, per questo bisogna installare ed utilizzare delle librerie esterne.

```
install.packages("readxl")
library(readxl)
read_excel("percorso/file.xls", sheet = "nomedelfoglio", na = "")  - legge il formato .xls
read_excel("percorso/file.xlsx", sheet = "data", na = "")          - legge il formato .xlsx
```

Andiamo ad analizzarne gli argomenti:

**sheet** = "nomedelfoglio" indica il **nome** del foglio Excel o il **numero** (primo, secondo, terzo foglio, etc..) da importare in R.
**na** = "" Vettore di caratteri delle stringhe da interpretare come valori mancanti. Per impostazione predefinita, readxl tratta le celle vuote come dati mancanti.

# Importare ed esportare dati

Il TSV è l'acronimo di *Tab Separated Values*, in R questi tipi di file possono essere importati utilizzando due metodi: uno è utilizzando le funzioni presenti nel pacchetto **readr** e un altro metodo è importare il file **.tsv** specificando il delimitatore come tab space(" \t") nella funzione read.delim().

```
install.packages("readr")
library(readr)
read_tsv("percorso/file.tsv")
```

In generale, la funzione read.delim() in R viene utilizzata per leggere i file separati da spazi per impostazione predefinita, ma specificando il delimitatore (separatore) come "\t" possiamo anche leggere i file TSV in R.

```
read.delim("percorso/file.tsv", sep ="\t")
```

**sep** = "\t" specifica il tipo di separatore di colonna. Per leggere file .tsv è necessario inserire la tabulazione **\t**.

# Importare ed esportare dati

## La libreria **readr**

L'obiettivo di readr è fornire un modo rapido e intuitivo per leggere i dati rettangolari da file delimitati, come i valori separati da virgole (CSV) e i valori separati da tabulazione (TSV). È progettato per analizzare molti tipi di dati, fornendo al contempo un rapporto informativo sui problemi quando l'analisi porta a risultati imprevisti.

**Installazione**

```
# The easiest way to get readr is to install the whole tidyverse:
install.packages("tidyverse")

# Alternatively, install just readr:
install.packages("readr")
```

# Importare ed esportare dati

## La libreria **readr**

**readr** fa parte del core tidyverse, quindi puoi caricarlo con:

```
library(tidyverse)
#> ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.2 ──
#> ✔ ggplot2 3.4.0          ✔ purrr   1.0.1
#> ✔ tibble  3.1.8          ✔ dplyr   1.1.0
#> ✔ tidyr   1.3.0          ✔ stringr 1.5.0
#> ✔ readr   2.1.3.9000     ✔ forcats 1.0.0
#> ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
#> ✖ dplyr::filter() masks stats::filter()
#> ✖ dplyr::lag()    masks stats::lag()
```

Naturalmente, puoi anche caricare readr come singolo pacchetto:

```
library(readr)
```

# Importare ed esportare dati

## La libreria **readr**

Per leggere un dataset rettangolare con **readr**, si combinano due pezzi: una funzione che analizza le righe del file in singoli campi e una specifica di colonna.
readr supporta i seguenti formati di file con queste funzioni `read_*()`:

`read_csv()`: valori separati da virgola (CSV)
`read_tsv()`: valori separati da schede (TSV)
`read_csv2()`: semicolon-separated values with , as the decimal mark
`read_delim()`: file delimitati (CSV e TSV sono importanti casi speciali)
`read_fwf()`: file a larghezza fissa
`read_table()`: file separati da spazi bianchi
`read_log()`: file di registro Web

# Importare ed esportare dati

## La libreria **readr**

La “specifica di colonna” descrive il tipo di dati che è contenuto in essa (ad es. carattere, numerico, data/ora, ecc.). In assenza di una specifica di colonna, **readr** indovinerà i tipi di colonna dai dati.
Se digitiamo nell’help di RStudio `vignette("column-types")` è possibile ottenere maggiori dettagli su come **readr** indovina i tipi di colonna.
L'ipotesi del tipo di colonna è molto utile, soprattutto durante l'esplorazione dei dati, ma è importante ricordare che si tratta solo di ipotesi.

# Importare ed esportare dati

## La libreria **readr**

L'esempio seguente carica un file di esempio in bundle con readr e indovina i tipi di colonna:

```
chickens <- read_csv(readr_example("chickens.csv"))

#> Rows: 5 Columns: 4
#> ── Column specification ────────────────────────────────────────────────────────
#> Delimiter: ","
#> chr (3): chicken, sex, motto
#> dbl (1): eggs_laid
#>
#> ℹ Use `spec()` to retrieve the full column specification for this data.
#> ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
#> # A tibble: 5 × 4
#>   chicken                sex     eggs_laid       motto
#>   <chr>                  <chr>       <dbl>       <chr>
#> 1 Foghorn Leghorn        rooster         0       That's a joke, ah say, that's a jok…
#> 2 Chicken Little         hen             3       The sky is falling!
#> 3 Ginger                 hen            12       Listen. We'll either die free chick…
#> 4 Camilla the Chicken    hen             7       Bawk, buck, ba-gawk.
#> 5 Ernie The Giant Chicken rooster        0       Put Captain Solo in the cargo hold.
```

# Importare ed esportare dati

## La libreria **readr**

Si noti che **readr** stampa i tipi di colonna, in questo caso i tipi di colonna indovinati. Questo è utile perché ti permette di verificare che le colonne siano state lette come previsto. In caso contrario, significa che devi fornire la specifica della colonna.
A primo impatto sembra un grosso problema ma, fortunatamente, **readr** offre un buon flusso di lavoro per questo. Possiamo utilizzare la funzione `spec()` per recuperare la specifica della colonna (indovinata).

```
spec(chickens)
#> cols(
#>   chicken = col_character(),
#>   sex = col_character(),
#>   eggs_laid = col_double(),
#>   motto = col_character()
#> )
```

# Importare ed esportare dati

## La libreria **readr**

Volendo, ora possiamo copiare, incollare e modificare il tutto per creare una chiamata **readr** più esplicita che esprima i tipi di colonna desiderati.
Qui esprimiamo che il sesso deve essere un fattore con 2 livelli (gallo e gallina), in quest'ordine, e che eggs_laid deve essere un numero intero.

```
chickens <- read_csv(
  readr_example("chickens.csv"),
    col_types = cols(
    chicken   = col_character(),
    sex       = col_factor(levels = c("rooster", "hen")),
    eggs_laid = col_integer(),
    motto     = col_character()
  )
)
```

# Importare ed esportare dati

I formati **.txt** e **.csv** sono formati di esportazione/importazione universale. Tutti i programmi consentono infatti di importare i files da uno di questi formati, o da entrambi. Il formato migliore di interscambio dei dati fra R e altre applicazioni di gestione ed analisi dei dati è senz'altro il formato CSV:

`write.csv()` è la funzione base per esportare un data frame in un file CSV con i campi separati da delimitatore virgola, intestazione (nomi di colonna), indice di righe e valori racchiusi tra virgolette..

Il formato di file CSV è il modo più semplice per archiviare dati scientifici, analitici o qualsiasi dato strutturato (bidimensionale con righe e colonne). I dati in CSV sono separati dal delimitatore più comunemente virgola (,) ma puoi anche utilizzare qualsiasi carattere come ; , tab…

# Importare ed esportare dati

```r
# Esporta in CSV senza nomi di riga
write.csv(df,file='/Users/admin/new_file.csv', row.names=FALSE)

# Esporta in CSV con spazio vuoto per NA
write.csv(df,file='/Utenti/admin/nuovo_file.csv',na='')

# Esporta in CSV senza virgolette
write.csv(df,file='/Users/admin/new_file.csv',quote=FALSE)

# Esporta in CSV senza intestazione
write.csv(df,file='/Users/admin/new_file.csv',col.names=FALSE)

# Scrivi con la codifica UTF-8
write.csv(df,file='/Users/admin/new_file.csv',fileEncoding = "UTF-8")
```

# Importare ed esportare dati

La funzione `write.table()` esporta un file dopo averlo convertito in dataframe, cioè in oggetto a due dimensioni.
Per questione di compatibilità, è necessario fare attenzione da una parte al separatore di campo (cioè delle colonne), e dall'altra al separatore dei decimali (la virgola come in italiano, il punto in inglese e nello standard scientifico internazionale).
Per quanto riguarda la struttura del dataset:
se esiste una variabile con gli identificativi di caso (riga), è preferibile esplicitare l'argomento `row.names = FALSE`;
è anche preferibile definire il codice per i dati mancanti, con l'argomento `na =`.

# Importare ed esportare dati

**Formato csv**

```
write.table(DATAFRAME_DA_SALVARE, "DATAFRAME_SALVATO.csv",
            sep = ";",              # punto e virgola
            row.names = FALSE,      # se abbiamo la variabile ID
            dec = ",",              # separatore di decimali
            na = "",                # dati mancanti come celle vuote
            quote = TRUE
)
```

**Formato testo tab-delimited**

```
write.table(DATAFRAME_DA_SALVARE, "DATAFRAME_SALVATO.txt",
            sep = "\t",             # tabulazione
            row.names = FALSE,
            dec = ",",
            na = ""
)
```

# Importare ed esportare dati

Dobbiamo sempre ricordarci di assegnare il nome ai nostri oggetti in R.

Se abbiamo impostato correttamente la directory non abbiamo necessità di specificare dove salvare il dframe.

**Formato csv**

```
write.table(DATAFRAME_DA_SALVARE, "DATAFRAME_SALVATO.csv",
            sep = ";",                # punto e virgola
            row.names = FALSE,        # se abbiamo la variabile ID
            dec = ",",                # separatore di decimali
            na = "",                  # dati mancanti come celle vuote
            quote = TRUE
)
```

**Formato testo tab-delimited**

```
write.table(DATAFRAME_DA_SALVARE, "DATAFRAME_SALVATO.txt",
            sep = "\t",               # tabulazione
            row.names = FALSE,
            dec = ",",
            na = ""
)
```

# Importare ed esportare dati

Salvare in formato Excel…
Il salvataggio in formato Excel è quasi sempre sconsigliato perché poco pratico ai fini della gestione di grandi quantità di dati. Inoltre, consideriamo che Excel permette l'importazione di dati .csv e .txt.
Possiamo utilizzare il pacchetto **writexl** per esportare il dataframe in Excel da R:

```
install.packages("writexl")
library("writexl")
write_xlsx(DATAFRAME_DA_SALVARE,"DATAFRAME_SALVATO.xlsx")
```

# Esercitazione

Applichiamo quanto più possibile abbiamo imparato finora utilizzando il database fish, un file Excel in cui sono tabulati una serie di dati relativi alle caratteristiche funzionali di 635 diverse specie di pesci. Il file Excel è suddiviso in tre fogli:

1. Alimentazione
2. Dimensioni
3. Comportamento

Scopo della esercitazione è mettere in pratica tutta una serie di comandi imparati finora per la gestione delle tabelle dati.