Appunti sul "servizio" repl.it

Luca Manini*

Ottobre 2020

1 Introduzione

Alcuni appunti su "servizio" `repl.it` che utilizzeremo per fare esercizio di programmazione, sia in Python che in SQL (database).

2 repl.it

repl.it è uno strumento on-line via browser (Firefox e parenti) che permette di fare delle sessioni di programmazione in tantissimi linguaggi diversi, tra cui Python.

2.1 Una nota sul nome del servizio.

repl è un acronimo molto usato in informatica e sta per **read eval print loop**. Come spesso capita con gli acronimi in Inglese, anche questo si legge da destra a sinistra quindi si tratta di un programma che lavora in un **ciclo** (*loop*) in cui **stampa** (*ptiny*) ciò che ottiene **valutando** (*evaluate*) l'espressione che ha **letto** (*read*).
Un repl è quindi, in generale, un'interfaccia utente (*user interface*) **testuale**, spesso chiamata anche **a linea di comando** (*command line*) verso un programma. In questo caso il programma è un **interprete Python**, ossia un programma che è in grado di leggere, comprendere ed eseguire codice Python.

2.2 Partenza

Io non ho mai usato questo strumento prima d'ora e quindi:

1. siamo tutti sulla stessa barca e sarà importante aiutarci a vicenda,
2. secondo in questi appunti non do niente per scontato, anche se forse voi avete già fatto pratica e forse ne sapete più di me!

Le prime funzionalità che mi pare utilizzeremo sono:

1. la possibilità di creare e salvare più file sorgente (ossia file che contengono codice Python) che compariranno poi sulla colonna di sinistra,

2. la possibilità di eseguire il codice presente in questi file, utilizzando il mega pulsante verde con la scritta RUN. **Attenzione**: repl esegue sempre e solo (direi) il file `main.py`, quindi almeno all'inizio useremo solo quello!

3. la possibilità utilizzare Python in modalità "interattiva" scrivendo direttamente il codice nella finestra di destra (la stessa dove viene scritto ciò che stampa (*print*) il codice eseguito da file,

4. la possibilità di usare, in futuro dopo che vi avrò spiegato cos'è a cosa serve e come si usa, un sistema di controllo di revisione,

2.3 Iscrizione (account)

Fortunatamente (ormai è purtroppo abbastanza raro), repl.it si può usare direttamente senza bisogno di registrarsi (*to sign up*) ma la registrazione è necessaria sia per poter "salvare" i sorgenti sia per poter utilizzare il sistema di controllo di revisione (vedi oltre).

3 Python

3.1 L'interprete interattivo

Python è un linguaggio **interpretato**, il che vuol dire che il codice sorgente viene eseguito "così com'è" direttamente da un programma chiamato **interprete** invece di essere tradotto, da un programma chiamato di solito **compilatore** in un altro linguaggio comprensibile (ed eseguibile) da "un altro esecutore" (ad esempio "la CPU").
Python, come succede spesso ma non sempre con i linguaggi interpretati, è anche un linguaggio **interattivo,** il che significa che è possibile "direttamente" scrivendo singole linee di codice che vengono eseguite "subito" e il cui risultato (se c'è) viene anche visualizzato (stampato) subito, **senza bisogno di usare la funzione print**!
Questa modalità non è comoda per scrivere programmi di una certa lunghezza, ma è comodissima, tra le altre cose, per fare un po' di pratica con i concetti, i tipi di dato e le funzionalità di base del linguaggio.

3.1.1 Pratica

Vediamo quindi di fare un po' di pratica.

1. Espressioni semplici

   (a) se scrivo 123 (e poi invio) l'interprete stampa 123, la ragione è che 123 è una **espressione** (anche se banale visto che è un literal), l'interprete la **valuta**, ottiene un **valore** a cui probabilmente siamo interessati e quindi lo stampa (senza dover usare print).

   (b) se scrivo `a = 123`, l'interprete non stampa nulla, perché in Python le assegnazioni non sono espressioni, non hanno un valore e quini c'è poco da stampare!

   (c) se però adesso scrivo `a`, l'interprete stampa 123, perché `a` (così da solo) è una espressione (un po' meno banale di un *literal*), quindi viene valutata, si ottiene un valore che viene stampato.

2. Tipi di dato

   Abbiamo già parlato di tipi di dato (*data types*) in generale e del fatto che anche in Python un dato è sempre di un certo tipo, che posso ottenere con la "funzione" `type`. Vediamo cosa ottengo se uso `type` passando un *literal*, un nome o una espressione.

```
>>> type(123)
<class 'int'>
>>> a = 123
>>> type(a)
<class 'int'>
>>> type(100 + 23)
<class 'int'>
```

   Ottengo sempre lo stesso risultato, che indica che il tipo è `int`. Fate però attenzione: ciò non significa che il nome `a` è di tipo `int` o che l'espressione `123 + 543` è di tipo `int`; in tutti e tre i casi, come sempre, il **parametro attuale** che io passo ad una funzione viene prima di tutto valutato e poi **passato** alla funzione. Ciò implica che in tutti e tre i casi, alla funzione `type` attiva un `123`, che è di tipo `int`!

   Esercizio: provate ad usare `type` con altri tipi di dato che conoscete.

3. Help

   Un'altra funzionalità da sfruttare nella modalità interattiva è la funzione `help`, che dà accesso alla documentazione *on-line*. Notare: l'informazione fornita da `help` è "interna" agli oggetti di cui chiedete la documentazione stessa ed è quindi sempre disponibile (non viene cercata su un qualche file esterno o in rete!).

   Vediamo un esempio con `help(print)`:

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

   Breve spiegazione:

   (a) la prima riga ci dice che `print` è una funzione (*function*), in particolare è una funzione *built-in* (ossia "già presente" nell'interprete) e che è definita nel modulo `builtins` (vedremo poi cosa sono i moduli).

   (b) la riga successiva mostra la *signature* della funzione, in pratica (potete pensare che sia) ciò che c'è scritto dopo `def` nella definizione di una funzione qualsiasi.

   (c) `value` indica è il dato che vogliamo stampare,

   (d) i tre puntini indicano che è possibile passare altri argomenti oltre al primo,

(e) `sep`, `end`, `file` e `flush` rappresentano degli argomenti opzionali (che vedremo nelle prossime lezioni...), e il valore dopo il segno di uguale indica il valore per difetto (*default value*) ossia il valore usato se non ne forniamo uno esplicitamente,

(f) le righe successive danno informazioni di dettaglio sui vari argomenti.

Nota importante: quando definiamo delle funzioni, possiamo fornire la documentazione in una *docstring*, ossia in una stringa posta subito dopo la riga `def`. Un esempio vale mille parole:

```
>>> def hello(who):
...     "Say hello to WHO"
...     print("Hello " + who)
...
>>> hello("World")
Hello World
>>> help(hello)
Help on function hello in module __main__:

hello(who)
    Say hello to WHO
```

4. Completion (completamento?)

   Un'altra funzionalità molto utile è il completamento automatico. Molti "strumenti di sviluppo" forniscono un qualche meccanismo che "scrive per voi" o che almeno suggerisce cosa potreste scrivere. Uno molto utile è il completamento, in particolare il completamento delle "proprietà" (attributi, metodi etc.).

   Per esempio: se io ho una stringa e non sono bene "cosa posso farci", posso cercare tra i **metodi** (parenti delle funzioni) disponibili e per fare ciò mi basta scrivere un nome (purtroppo non funziona con i *literal*) e usare il tasto `TAB`.

   Esempio (in cui ho tolto alcune righe... e in cui nella seconda riga, dopo `name.` ho premuto il tasto `TAB`).

```
>>> name = "Luca"
>>> name.
name.capitalize(    name.isalpha(       name.ljust(         name.split(
name.casefold(      name.isascii(       name.lower(         name.splitlines(
name.center(        name.isdecimal(     name.lstrip(        name.startswith(
.....................................................................................
name.index(         name.isupper(       name.rsplit(
name.isalnum(       name.join(          name.rstrip(
>>> help(name.lower)
Help on built-in function lower:

lower() method of builtins.str instance
    Return a copy of the string converted to lowercase.

>>> name.lower()
'luca'
```

3.2 Qualche "trucco" comodo

Adesso abbiamo fatto una prova per vedere se la funzione `sum_pos` funziona correttamente; ma ovviamente una singola prova è un po' poco e soprattutto il "controllo" dobbiamo farlo noi, controllando ciò che stampa il codice di esempio.

Per rendere il tutto un po' più comodo possiamo usare qualche trucco (in realtà non ci sono trucchi, ci sono solo modi intelligenti di usare le funzionalità del linguaggio!).

3.2.1 Esecuzione di più esempi

Prima di tutto vediamo come eseguire la funzione `sum_pos` più volte con dati diversi. La funzione richiede come argomento una lista (o simile) di interi, e quindi per fare più "giri" ci serve una lista di liste.

```
data = (
    (1,2,3),
    (0, -5, 7),
    (3 -4, 0, 2),
    )
for ii in data:
    t = sum_pos(ii)
    print(ii, "->", t)
```

```
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/tmp/babel-UlpH5T/python-khUZCV", line 7, in <module>
    t = sum_pos(ii)
NameError: name 'sum_pos' is not defined
python.el: native completion setup loaded
```

3.3 Codice "da file"

Vediamo adesso anche un esempio in cui scriviamo del codice in un file e poi lo eseguiamo.

1. usiamo il file `main.py` già creato da `repl`,
2. inseriamo una funzione `sum_pos` che dato un "insieme di numeri" **calcola e restituisce** la somma dei soli positivi.

```
def sum_pos(ints):
    "The sum of the elements in INTS"
    t = 0               # creo un "accumulatore" nullo
    for i in ints:      # ciclo su tutti gli elementi di INTS
        if i > 0:       # e solamente se quello corrente è positivo
            t += i      # lo sommo all'accumulatore
    return t            # restituisco il risultato
```

1. per fare un primo controllo proviamo ad usare la funzione in un paio di esempi,

```
ii = (2,3,4,0-12)
t = sum_pos(ii)
print(ii, "->", t)
```

```
(2, 3, 4, -12) -> 9
```

4 SQL

Oltre che per la programmazione in un linguaggio "generico" (*general purpose*) come Python, useremo repl.it anche per imparare e fare pratica con SQL, un linguaggio orientato ai database.
Il linguaggio SQL nasce negli anni settanta, ed è quindi uno dei linguaggio più vecchi tra quelli ancora in uso. In tutti questi anni è ovviamente cambiato moltissimo, ne sono nate molte versioni (dialetti) e nonostante gli sforzi di standardizzazione ancora adesso è meglio essere pronti a trovare differenze anche significative tra le varie implementazioni.
Anche l'SQL, come Python, si può considerare, almeno per l'uso che ne faremo noi, un linguaggio "interpretato e interattivo"; ciò fa sì che si presti ad essere utilizzato in un ambiente come quello fornito da repl.it.

4.1 sql

Come sapete, i database sono dei "contenitori di dati strutturati", che servono da "base" per multiple applicazioni, e che sono generalmente "gestiti" da software molto complessi chiamati DBMS (database management systems) che funzionano tipicamente in modalità *client server* (con il DBMS che gira su un computer e le "applicazioni" che girano su altre macchine). Esempi di database di questo tipo sono Postgres o Oracle.
Noi però useremo sql, un sistema molto più semplice e "leggero" che ha vari vantaggi:

1. accettare un SQL relativamente completo e "standard",
2. viene usato spesso *embedded* in altre applicazioni (ad esempio per l'agenda dei telefoni Android o per le "preference" di vari browser),
3. è molto comodo da usare anche da Python.

XXX vi consiglio di scaricarvi in locale la documentazione di riferimento.

4.2 Esempio

In questo semplice esempio vedremo come usare le funzionalità di base dell'SQL per creare una tabella ed inserire ed "estrarre" alcuni dati. Questa è la sequenza tipica di qualsiasi "esempio", grande o piccolo che sia. Vedremo poi alcuni "comandi" di SQLite che non fanno parte del linguaggio SQL ma che sono molto utili nell'interazione con l'interprete.

4.2.1 Creazione del database

Nella maggioranza dei DBMS, la prima operazione da fare è creare un database, nel senso di creare il file (o più spesso l'insieme di file) su cui vengono "salvati" (in termine tecnico "resi persistenti") i dati.
Siccome SQLite è pensato per operare fondamentalmente in memoria questa operazione non è in linea di principio necessaria; esiste comunque un comando (non SQL) per salvare i dati in un file (con "db" come estensione di *default*) e un altro comando per "ricaricarlo".
In pratica SQLite crea automaticamente un database in memoria di nome `main:` ed è per questo che anche usando `repl.it` non avremo bisogno di preoccuparci di questo aspetto.

4.2.2 Creazione tabella

La prima cosa che viene in mente di fare è giustamente creare una tabella, ma quando si stanno facendo degli esempi è normale "rieseguire" ogni volta tutto il codice, ma cercare di creare una tabella se questa esiste già è un errore. La soluzione che si usa normalmente è cancellare la tabella (con l'istruzione SQL `drop table`) per poi ricrearla. Peccato che anche cancellare una tabella che non esiste è un errore! Fortunatamente, in molti SQL si può usare il seguente codice, che credo non richieda spiegazioni.
Nota: le istruzioni SQL **devono** essere terminate con un punto e virgola (*semicolon*)!

```
drop table if exists Person;
```

Per creare la tabella si usa l'istruzione `create table`, che come si può immaginare richiede di specificare il nome della tabella e la lista delle colonne.
Nota: come già detto l'SQL è un linguaggio molto "variabile", quindi quasi tutte le affermazioni sulla sua sintassi vanno prese un po' con le pinze. In ogni caso, l'SQL è **fondamentalmente non** *case sentitive*, quindi **in genere** "non distingue maiuscole e minuscole". È però assolutamente vitale essere coerenti e non sfruttare questo fatto. Se decidete, come faccio io, di scrivere quasi tutto minuscolo, fatelo sempre!
Qui creo una tabella `Person` con tre colonne `id`, `name` e `code`. La colonna `id` serve da chiave primaria.
Per una lista dei tipi di dato gestiti da SQLite basta consultare la documentazione online.
Nota: nella sua semplicità, SQLite ha in realtà pochi tipi di dato "nativi" (TEXT, NUMERIC, INTEGER, REAL, BLOB) che vengono poi utilizzati "al posto" di quelli più normali per l'SQL (come `CHAR` o `VARCHAR`).
Nota: la chiave primaria è spesso un intero senza particolare "significato" (per dire, non è come il "codice fiscale" o la coppia "nome/cognome") e l'unica caratteristica importante è che deve essere **sempre presente** (vincolo `not null` in SQL) e **deve avere un valore** in ogni riga (vincolo `unique` in SQL). Quando si inseriscono delle righe è quindi obbligatorio specificare un valore, e con il vincolo di unicità la cosa diventa una scocciatura! Tutti (?) i database forniscono dei meccanismi per generare questi valori in modo automatico, spesso semplicemente specificando un attributo `autoincrement`. SQLite ha un meccanismo diverso; **in pratica**, se si dichiara una colonna `integer primary key`, il suo valore, se non specificato nelle `insert` viene gestito automaticamente dal sistema (il come non ci interessa).

```
-- Esempio di creazione di una tabella (questo è un commento!)
create table Person (               -- nomi delle tabelle "capitalized"
    id integer primary key,         -- vincolo di PK
    name varchar[25] not null,      -- vincolo di unicità (sulla singola colonna)
    code char[5] not null unique,
    age integer not null            -- ultima riga SENZA virgola!
);
```

4.2.3 Giusto per farsi un'idea

Una piccola parentesi (che potete saltare) giusto per mostrare quanto sia "strano" l'SQL in quanto alla questione *case*.

```
sqlite> create table "foo" (id integer);
sqlite> drop TABLE foo;
```

```
sqlite> create table table (id integer);
Error: near "table": syntax error
sqlite> create table "TABLE" (id integer);
sqlite> drop table Table;
Error: near "Table": syntax error
sqlite> drop table "Table";
sqlite> .tables
```

4.2.4 Inserimento dati

Per "popolare" una tabella, ossia per inserire dati, ossia per aggiungere righe, si usa il comando `insert into` che richiede il nome della tabella, la lista delle colonne in cui si vogliono inserire (tra parentesi separati da virgole) seguita dai dati (una lista di valori tra parentesi per ciascuna riga).
Nota: la lista dei nomi delle colonne è **opzionale** nel qual caso si devono specificare dei valori per **tutte** le colonne (una scocciatura) e **esattamente nell'ordine** in cui sono state definite (ordine che di solito non si conosce!). Quindi in pratica: **non è opzionale**!
Nota: in SQL il **delimitatore** per le stringhe è l'**apice singolo** e non le **virgolette** (o apice doppio). Le virgolette servono per **quotare** i nomi degli identificatori quando si ci vuole complicare la vita! Il fatto che quasi tutti i database (incluso SQLite) "si bevano" anche le virgolette non è un buon motivo per derogare (e il fatto di aver programmato in `C` per una vita nemmeno!).

```
insert into Person      -- nome della tabella
(name, code, age)            -- nomi delle colonne
values
('Linus', '123-L', 51),     -- valori, nelleo stesso ordine
('Richard', '444-R', 67),   -- in cui sono state specificate
('Bob', '666-B', 63);       -- le colonne!
```

4.2.5 Estrazione dei dati

Con la `insert` abbiamo **scritto** dei nel database, adesso vediamo come, usando l'istruzione `select`, si possono **leggere**.
Nota: come sinonimo di "leggere" si usa spesso il termine "estrarre" che però può generare confusione perché da l'idea che i dati vengono "tolti" dal database, cosa che non avviene.
Per vedere il contenuto di una tabella si usa spesso questa espressione, che in pratica chiede di vedere il contenuto di tutte le colonne (l'asterisco) e di tutte le righe (perché questo è il comportamento normale in assenza di **filtri**).

```
select * from Person;
```

E questo è un tipico risultato (il separatore di colonne dipende da molti fattori, ma la virgola è abbastanza normale).

```
1,Linus,123-L,51
2,Richard,444-R,67
3,Bob,666-B,63
```

In realtà, la "select con asterisco" è da evitare per vari motivi, primo tra tutti il fatto che si può sapere in anticipo quante saranno le colonne e quali "nomi" avranno. Ricordarsi che le tabelle di una tabella possono cambiare nel tempo! È quindi molto più corretto esplicitare sempre la lista delle colonne.

Nota: per i "nomi" che seguono la *keyword* `select` si usa spesso il termine di "colonna", ma il termine corretto è **descrittore di colonna**, per motivi che chiarirò tra poco.

```
select code, name from Person;
```

```
123-L,Linus
444-R,Richard
666-B,Bob
```

4.2.6 Descrittori di colonna

Il motivo per cui è più corretto parlare di "descrittori" di colonna, è che i dati contenuti nelle colonne del "risultato" di una `select` sono sempre solo quelli contenuti nelle colonne delle tabelle a cui la `select` fa riferimento. Un esempio per tutti:

```
select code, 'Hello World!', 123 * 2 from Person;
```

che da il seguente risultato!

```
123-L,"Hello World!",246
444-R,"Hello World!",246
666-B,"Hello World!",246
```

4.2.7 Funzionalità "non SQL" di SQLite

La `select` appena vista è corretta, ma c'è un "piccolo problema": l'output non include i nomi (intestazioni) delle colonne. È giusto così, perché non fanno parte del contenuto della tabella, sono dei **metadati**, ma ciò non toglie che sarebbe utile poterli includere nel risultato, almeno quando lo si vuol "visualizzare".
Per fare questo si possono usare dei **comandi** che non fanno parte del linguaggio, ma che sono specifici dell'**interprete** SQLite.

```
.header on
.separator |
.mode column
.width 3 5 8
select id, code, name from Person;
```

E il risultato, molto più "bellino" del precedente, è:

```
id   code   name
---  -----  --------
1    123-L  Linus
2    444-R  Richard
3    666-B  Bob
```

I significato dei quattro comandi è il seguente:

header indica se mostrare (on) o no (off) le intestazioni,

separator mi pare chiaro!

mode indica il tipo di "formattazione" (html, csv, etc.)

width la larghezza delle varie colonne.

4.2.8 Selezione di righe (WHERE)

Abbiamo visto che in una `select` si possono "selezionare alcune colonne", se invece vogliamo selezionare delle righe, limitando così l'insieme delle righe su cui operiamo, dobbiamo usare la **clausola** `where`.

```
select name, code
from Person
where age > 60;
```

```
Richard,444-R
Bob,666-B
```

che come si vede "produce" solo due righe, quelle in cui la colonna `age` contiene un valore maggiore di 60. La clausola `where` deve essere sempre seguita, come nell'esempio, da una **espressione logica** e seleziona solo le righe per cui l'espressione, il cui valore tipicamente dipende dal contenuto delle righe, è "vero".
Nota: la clausola `where` non è legata alle `select` ma, come vedremo, può essere utilizzata anche in altre operazioni come l'aggiornamento o la cancellazione di dati e anche per "collegare" più tabelle.
L'uso della `where` è semplice e direi intuitivo; nella pratica poi le difficoltà nascono dalla complessità dell'espressione logica che spesso coinvolge multipli operatori su multiple colonne.
Un esempio appena più complicato:

```
.header on
.separator |
.mode column
select name, code, age
from Person
where age > 60 and name = 'Bob';
```

che genera una sola riga perché ho aggiunto, con l'operatore `and` una seconda condizione che ha reso la condizione complessiva più "stringente".

```
name        code        age
----------  ----------  ----------
Bob         666-B       63
```

Nota: è in generale buona norma, sopratutto quando il risultato è pensato per essere "letto" includere nella lista delle colonne della `select` anche le colonne utilizzate nei vari "filtri", altrimenti si fa fatica a capire perché alcune righe sono presenti e altre no!.

4.2.9 Due tabelle (o una sola?)

Vediamo adesso un esempio in cui sono "necessarie" due tabelle: l'agenda telefonica. Abbiamo una lista di persone (nomi) e una lista di numeri telefonici, con l'ipotesi di base che una persona può avere più numeri di telefono ma non il viceversa, quindi ciascun numero di telefono è associato ad una sola persona.
Nota: lasciamo per ora l'interessante da parte il caso di persone senza numeri di telefono e telefoni senza "padrone".

1. Tabella singola

```
drop table if exists Agenda;
create table Agenda (
    name varchar[25] not null,
    phone varchar[20] not null unique
);

insert into Agenda (name, phone)
values
('Linus'   , '123-44-55-666'),
('Linus'   , '123-44-00-888'),
('Richard' , '300-00-11-22'),
('Richard' , '300-00-33-00'),
('Bob'     , '400-99-55-777');

select name, phone from Agenda;
```

```
Linus,123-44-55-666
Linus,123-44-00-888
Richard,300-00-11-22
Richard,300-00-33-00
Bob,400-99-55-777
```

Gestire il dati in questo mode, dal punto di vista delle teoria dei database relazionali, non è corretto, quantomeno perché ci sono dei dati duplicati (i nomi). Ci sono varie ragioni per cui questo è un problema, ma al momento non ci interessano. Ci interessa però la "soluzione standard": dividere i dati in due tabelle, una per i nomi e una per i numeri.

2. Due tabelle

```
drop table if exists Person;
create table Person (
    id integer primary key,
    name varchar[25] not null unique
);

drop table if exists Phone;
create table Phone (
    id integer primary key,
    id_person integer references Person,
    number varchar[20] not null unique
);

insert into Person (id, name) values
(10, 'Linus'),
(20, 'Richard'),
(30, 'Bob');

insert into Phone (id_person, number) values
(10, '123-44-55-666'),
(10, '123-44-00-888'),
(20, '300-00-11-22'),
(20, '300-00-33-00'),
(30, 'Bob,400-99-55-777');
```

3. Collegamento (via where)

```
select name, number
from Person, Phone
where Person.id = Phone.id_person;
```

```
Linus,123-44-55-666
Linus,123-44-00-888
Richard,300-00-11-22
Richard,300-00-33-00
Bob,"Bob,400-99-55-777"
```

4. Collegamento (via join)

```
select name, number
from Person, Phone
where Person.id = Phone.id_person;
```

```
Linus,123-44-55-666
Linus,123-44-00-888
Richard,300-00-11-22
Richard,300-00-33-00
Bob,"Bob,400-99-55-777"
```

5. Collegamento (full join)

```
select name, number
from Person, Phone;
```

```
Linus,123-44-00-888
Linus,123-44-55-666
Linus,300-00-11-22
Linus,300-00-33-00
Linus,"Bob,400-99-55-777"
Richard,123-44-00-888
Richard,123-44-55-666
Richard,300-00-11-22
Richard,300-00-33-00
Richard,"Bob,400-99-55-777"
Bob,123-44-00-888
Bob,123-44-55-666
Bob,300-00-11-22
Bob,300-00-33-00
Bob,"Bob,400-99-55-777"
```