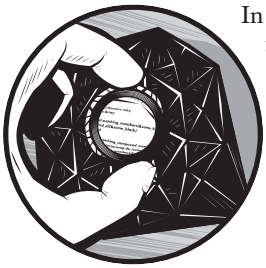
1

Interactive Ruby e l'ambiente



In Ruby, come nella maggior parte dei linguaggi di programmazione, generalmente si memorizzano i programmi in file esterni e li si utilizza come unità. Ruby però offre la possibilità di digitare le linee di un programma una alla volta e di visualizzare i risultati mentre si procede, utilizzando Interactive Ruby (irb); questo è una shell, simile a bash dei sistemi Unix o tipo Unix o alla riga dei comandi di Windows. Utilizzare irb permette di avere una buona percezione di come Ruby elabora le informazioni e dovrebbe anche aiutare a capire la basi ancor prima di iniziare a programmare.

Chi dovrebbe leggere questo capitolo? Il lettore che già ha utilizzato Ruby e conosce i termini espressione, irb, controllo di flusso, variabile, funzione, metodo *e* costante, *probabilmente potete semplicemente saltare questo capitolo (se si dovesse incontrare qualcosa di non familiare in seguito, è possibile sempre tornare indietro). Se non si ha mai programmato prima si dovrebbe leggere questo capitolo con attenzione. Se si ha già utilizzato un linguaggio con ambiente interattivo, come Lisp o Python, è probabilmente possibile semplicemente guardare le sessioni con irb per vedere come si differenzia Ruby rispetto ai linguaggi già noti. È facile che si riveli seguire concetti standard.*

Il programma irb è un esempio di ambiente REPL (read-eval-print-loop, leggi-valuta-stampa-cicla). Questa è un'idea presa da un antenato di Ruby, Lisp. Questo significa proprio quello che dice il nome: legge una linea, la valuta, stampa i risultati della valutazione e cicla, attendendo una nuova riga. La shell fornisce un riscontro per ciascuna linea si introduce, un modo ideale per imparare la sintassi di un linguaggio.

Avviare irb

L'avvio di irb è semplicissimo. Su una macchina Unix o tipo Unix (come GNU/Linux o Mac OS X) è possibile semplicemente digitare `irb` al prompt della shell. Questo dovrebbe restituire il seguente risultato:

```
$ irb
irb(main):001:0>
```

Su macchina Windows si sceglie Start | Run, si digita `irb` e poi si fa clic su OK.

Utilizzare irb

Ora che è stato avviato, irb è in attesa che si digiti la prima linea. Le linee sono composte da una o più espressioni.

Espressioni

Per come è concepito Ruby, una espressione è semplicemente un pezzo di codice che ha un valore. Vediamo come irb reagisce all'espressione `"Hello, world!"` (nella più bella tradizione della programmazione dei computer).

```
irb(main):001:0> "Hello, world!"
=> "Hello, world!"
```

Il listato mostra la linea che è necessario digitare, oltre alla risposta irb. Si noti inoltre che irb mostra i numeri di linea all'inizio di ciascuna riga. Ogni tanto si farà riferimento a questi numeri.

Che cosa è successo? Digitando `"Hello, world!"` irb lo ha semplicemente ripetuto. La parte interessante di questo comportamento è quella non esplicita. L'espressione inserita ha un valore in Ruby, e quindi in irb. `"Hello, world!"` è una `String`, che è una sequenza di caratteri, solitamente inclusi in apici singole o doppie.

Qualsiasi cosa è un oggetto

In Ruby, come nel suo antenato Smalltalk, tutto è un *oggetto*, che è semplicemente una istanza di una classe. `"Hello, world!"` è una istanza della classe `String`. Verifichiamo questa cosa in irb:

```
irb(main):002:0> "Hello, world!".class
=> String
```

Gli oggetti hanno i *metodi* (chiamati su un oggetto come in `un_oggetto.un_metodo`), che sono semplicemente azioni che l'oggetto può eseguire. Il metodo chiamato `class` non fa altro che indicare a quale classe appartiene; in altre parole, il tipo di cosa che contiene. Dato che `"Hello, world!"` è una `String`, questo è esattamente quello che riporta il metodo `class` una volta chiamato. Ci sono ovviamente altri oggetti, oltre le stringhe.

Questo libro assume che il lettore abbia familiarità con la programmazione orientata agli oggetti. Se non l'avete, ecco una descrizione lampo. Un oggetto *è una cosa. Potrebbe essere qualsiasi tipo di cosa. Ogni oggetto è una istanza di una classe; per esempio, gli oggetti* Glasgow, Cairo *e* Buffalo *sono tutte istanze della classe* City. *Gli oggetti sono uno distinto dall'altro, ma sono sempre lo stesso tipo di cosa.* Monty Python *e* The Kids in the Hall *sarebbero entrambe istanze di* Comedy Troupe *e così via. In Ruby, le istanze hanno tradizionalmente nomi che iniziano per minuscola e utilizzano underscore al posto di spazi; i nomi delle classi invece sono in CamelCase. Nel vero codice Ruby, la classe* `ComedyTroup` *avrebbe istanze (oggetti) chiamati* `monty_python` *e* `kids_in_the_hall`.

Interi, FixNum e BigNum

Un altro tipo di oggetto (o *classe*) è `Integer`, che è un qualsiasi numero divisibile per uno. Questi dovrebbero essere familiari: 0, 1, -5, 27 e così via. Digitiamo un intero in irb.

```
irb(main):003:0> 100
=> 100
```

Se si invoca il metodo `class` su un `Integer`, ritornerà `Fixnum` o `Bignum`, non `Integer`. Questo ha a che fare con il modo in cui Ruby memorizza i numeri internamente. I computer possono operare più velocemente se non sprecano spazio, quindi si devono preoccupare di quanto spazio effettivamente i numeri prendono. Però, i computer devono anche essere in grado di gestire numeri molto grandi. Quindi, il compromesso è di memorizzare i piccoli numeri in poco spazio, e dedicare più spazio a quelli grandi. I linguaggi sofisticati e ad alto livello come Ruby traducono tra questi diversi tipi di numeri in modo automatico, quindi è sufficiente trattarli come numeri senza preoccuparsi di dettagli specifici. Non è comodo? Per esempio, `100.class` ritorna `Fixnum`, perché `100` è abbastanza piccolo per stare in un `Fixnum`, ma il valore di `(100 * 100)` ci starà solo in un `Bignum`. È troppo grande per un `Fixnum`.

Si è visto che il numero 100 ha il valore 100 in irb, come ci si potrebbe aspettare. Ma si desidera essere in grado di fare di più che vedere semplicemente quanto digitato, quindi facciamo qualcosa con il numero 100. Aggiungiamogli 100.

```
irb(main):004:0> 100 + 100
=> 200
```

Si può vedere che irb ha sommato questi numeri in modo corretto ed ha mostrato il risultato. In Ruby, `100+100` è solo un'espressione, proprio come `"Hello, world!"` e `100`. L'espressione `100+100` vale, naturalmente, 200. I numeri hanno metodi chiamati +, che servono a sommarli ad altri numeri. È proprio quello che abbiamo appena fatto.

Aggiunta, concatenazione ed eccezioni

Il segno + può fare molto di più che aggiungere solo numeri. Proviamo a fare la somma di altre due espressioni:

```
irb(main):005:0> "Hello, " + "world!"
=> "Hello, world!"
```

Aggiungendo la stringa "Hello," alla stringa "world!" è stata creata una nuova stringa, più lunga, che vale "Hello, world!". Le stringhe, per l'esattezza, non possono essere propriamente "sommate". Utilizzano il simbolo + per eseguire un'operazione chiamata *concatenazione*, che è semplicemente l'aggiunta di una cosa alla fine di un'altra. In Ruby, il segno + significa, *esegui una qualsiasi operazione tipo aggiunta che abbia senso per questo tipo d'oggetto.* Questo consente di utilizzare il solo simbolo + e assumere che gli interi verranno sommati in modo "numerico", le stringhe in modo "stringa" e così via.
Cosa succede se si tenta di aggiungere due tipi diversi di oggetti? Scopriamolo con irb.

```
irb(main):006:0> "Hello, world!" + 100
TypeError: failed to convert Fixnum into String
        from (irb):6:in '+'
        from (irb):6
```

Questa espressione non funziona bene come le altre. `TypeError` è un esempio di quello che Ruby (e altri linguaggi), chiamano *eccezione*, una segnalazione dal linguaggio di programmazione che si è verificato un errore. In questo caso signficia che Ruby non è contento se gli chiedete di sommare stringhe e numeri. Le stringhe sanno come aggiungersi l'una all'altra, e lo stesso i numeri, ma non possono incrociare i tipi. Quando si calcola una somma, entrambi gli operandi devono essere dello stesso tipo.

Conversione di tipo

La soluzione a questo problema è un'operazione chiamata conversione di tipo (casting), che è un'operazione che trasforma il tipo di una variabile in un altro. Ecco un esempio si conversione di tipo in irb:

```
irb(main):007:0> "Hello, world!" + 100.to_s
=> "Hello, world!100"
```

Qui viene chiamato il metodo `to_s` su `100` prima di cercare di aggiungerlo a `"Hello, world!"`. Questo metodo significa `to String` e, come si può ipotizzare dal nome, converte in una stringa l'oggetto su cui viene chiamato. Quindi, quando si opera la somma, i due operandi sono entrambi stringhe, e Ruby li concatena correttamente (in realtà, tecnicamente parlando, non è stato fatto un casting, ma è stato creato un oggetto completamente nuovo che è l'equivalente stringa per il numero 100). Verifichiamo quindi che `100.to_s` sia una stringa:

```
irb(main):008:0> 100.to_s
=> "100"
```

Quindi è vero. Ma cosa succede quando si vuole convertire qualcosa a un intero? Esiste un metodo `to_i` che potrebbe essere invocato sulla stringa "100"? Scopriamolo.

La conversione di tipo è comune nei linguaggi fortemente tipizzati come Ruby. È meno comune in quelli debolmente tipizzati, sebbene si possa comunque presentare. Entrambi gli approcci hanno i propri sostenitori.

```
irb(main):009:0> "100".to_i
=> 100
```

È infatti possibile. Quindi ora si sa come convertire stringhe in interi e viceversa, utilizzando i metodi `to_s` e `to_i`. Sarebbe carino se si potesse vedere un elenco di tutti i metodi che è possibile chiamare su un dato oggetto. Si può fare, con un metodo dal nome molto opportuno: `methods`. Chiamiamolo sull'intero 100.

```
irb(main):010:0> 100.methods
=> ["<=", "to_f", "abs", "-", "upto", "succ", "|", "/", "type", "ti-
mes", "%", "-@", "&", "~", "<", "**", "zero?", "^", "<=>", "to_s",
"step", "[]", ">", "==", "modulo", "next", "id2name", "size", "<<",
"*", "downto", ">>", ">=", "divmod", "+", "floor", "to_int", "to_i",
"chr", "truncate", "round", "ceil", "integer?", "prec_f", "prec_i",
"prec", "coerce", "nonzero?", "+@", "remainder", "eql?", "===", "clo-
ne", "between?", "is_a?", "equal?", "singleton_methods", "freeze",
"instance_of?", "send", "methods", "tainted?", "id", "instance_varia-
bles", "extend", "dup", "protected_methods", "=~", "frozen?", "kind_
of?", "respond_to?", "class", "nil?", "instance_eval", "public_metho-
ds", "__send__", "untaint", "__id__", "inspect", "display", "taint",
"method", "private_methods", "hash", "to_a"]
```

Si può notare che sia + che `to_s` *sono nell'elenco dei nomi dei metodi.*

Si noti che è possibile concatenare i metodi, e quindi scrivere qualcosa come `100.methods.sort`*. Se si prova questa espressione in irb si ottiene l'elenco dei metodi ottenuto con* `100.methods`*, ma elencati in ordine alfabetico.*

Vettori

Si noti che l'output prodotto da `methods` è racchiuso tra parentesi quadre ([]). Queste indicano che gli elementi inclusi sono membri di `Array` (vettore), che è un elenco di oggetti. I vettori sono semplicemente un'ulteriore classe presente in Ruby, come `String` o `Integer` e (diversamente da altri linguaggi), non c'è il bisogno che tutti gli elementi del vettore siano di una stessa classe. Un modo facile per convertire un singolo elemento in un vettore è di racchiuderlo tra parentesi, come nella riga seguente:

```
irb(main):011:0> [100].class
=> Array
```

Anche i vettori sanno come sommare loro stessi, come qui:

```
irb(main):012:0> [100] + ["Hello, world!"]
=> [100, "Hello, world!"]
```

Il risultato è semplicemente un altro vettore, che contiene tutti gli elementi dei vettori sommati.

Booleani

Oltre a classi per stringhe, interi e vettori, Ruby ha anche una classe chiamata `Boolean`. Le stringhe sono sequenze di caratteri, gli interi numeri divisibili per uno e i vettori sono elenchi di oggetti. I valori boolean possono invece assumere solo i valori vero o falso. I booleani hanno diversi usi, ma quelli più comuni sono legati a valutazioni per determinare se eseguire una azione o la sua alternativa. Queste operazioni sono chiamate *controllo di flusso.*

Il nome booleano deriva dal matematico George Boole, che svolse i primi lavori di formalizzazione di questi elementi.

Controllo di flusso

Una delle operazioni di controllo di flusso più utilizzate è `if`. Valuta l'espressione che segue come vera o falsa. Vediamo di dimostrare il suo funzionamento:

```
irb(main):013:0> 100 if true
=> 100
```

Abbiamo appena chiesto se l'espressione `100 if true` è vera. Dato che l'espressione `true` viene valutata a vero, è stato ottenuto il valore 100. Cosa succede quando l'espressione valutata da `if` non è vera?

```
irb(main):014:0> 100 if false
=> nil
```

Qui c'è qualcosa di nuovo. L'espressione `false` non è vera, quindi non si è ottenuto il valore 100. Infatti, non è stata ottenuta alcuna espressione. irb ha comunicato che non ha valore da riportare. Ruby dispone di un valore specifico che rappresenta l'assenza di valore (oppure un valore senza senso), che è `nil`.

Il valore potrebbe essere assente per diverse ragioni. Potrebbe essere un concetto che non può essere espresso, oppure potrebbe riferirsi a dati mancanti, che è proprio quello che è successo nell'esempio. Infatti, non abbiamo mai indicato a irb cosa riportare quando l'espressione da valutare è falsa, quindi il valore manca. Qualsiasi valore che potrebbe essere rappresentato come N/A è un buon candidato per il valore `nil`. Questa situazione si presenta spesso quando si interagisce con un database. Non tutti i linguaggi hanno `nil`; alcuni lo hanno, ma danno per scontato che sia successo un errore. Ruby gestisce bene i valori `nil` e li usa quando è appropriato.

Il valore `nil` è distinto dagli altri valori. Però, quando si forza Ruby a valutare nil come booleano, si ottiene il valore `false`, come mostrato:

```
irb(main):015:0> "It's true!" if nil
=> nil
```

Gli unici valori che hanno valore booleano false sono `nil` e `false`. In molti altri lunguaggi anche 0 o "" (stringa con zero caratteri) vengono valutati come falsi, ma non è il caso di Ruby. Tutto quanto non è `nil` o `false`, se forzato a booleano, ritorna `true`.

> *È necessario convertire esplicitamente stringhe e interi attraverso i metodi `to_s` e `to_i`, ma si noti che questo non è necessario per trattare con i booleani. Questa conversione è infatti implicita quando si utilizza `if`. Se si volesse attuare una conversione esplicita a `Boolean`, ci si potrebbe aspettare un metodo simile a `to_s` e `to_i`, chiamato `to_b`. In Ruby questo metodo ancora non esiste, ma ci scriveremo il nostro nel Capitolo 3.*

Ipotizziamo di volere un determinato valore, nel caso una data espressione sia vera (come abbiamo già fatto prima con la `if`), ma anche un qualche valore non-`nil` se l'espressione risulta false. Come si può fare? Ecco un esempio in irb:

```
irb(main):016:0> if true
irb(main):017:1> 100
irb(main):018:1> else
irb(main):019:1* 50
irb(main):020:1> end
=> 100
```

Questa è la nostra prima espressione multi linea in irb. Dovrebbe essere abbastanza diretta, ritornando 100, perché `true` valuta a vero. Proviamo ancora, con qualche differenza:

```
irb(main):021:0> if false
irb(main):022:1> 100
irb(main):023:1> else
irb(main):024:1* 50
irb(main):025:1> end
=> 50
```

Questa volta, dato che `false` valuta come falso, il valore dell'espressione su più linee è il valore di `else`, che è 50. Questo formato è un po' più prolisso dei test precedenti che utilizzavano solo `if`. È inoltre necessario introdurre la parola chiave `end` per indicare a irb quando termina l'espressione iniziata con `if`. Se si volesse fare test come queste espressioni multi linea spesso, la ridigitazione continua di più espressioni, diverse solo per piccole differenze, potrebbe divenire tedioso. Ecco quindi che entrano in gioco i metodi.

Si noti che irb offre alcune informazioni utili al prompt. Questo termina spesso con un simbolo >, che solitamente è preceduto da un numero. Questo indica il livello di annidamento, comunicando il numero di istruzioni end necessarie per ritornare al primo livello. Si noterò anche che qualche volta, invece che terminare con il simbolo >, il prompt termina con asterisco (*). Questo significa che irb ha una istruzione incompleta ed è in attesa di altri elementi per completarla. Molto utile.

Metodi

Abbiamo incontrato i metodi prima, ma li approfondiremo solo ora. Un *metodo* è semplicemente un pezzo di codice attaccato a un oggetto; si aspetta uno o più valori di input e ritorna qualcosa come risultato.

Ruby è orientato agli oggetti, quindi utilizza il termine metodo. *I linguaggi con minor focalizzazione sugli oggetti chiama i metodi* funzioni. *Un metodo non è altro che una funzione attaccata a un'oggetto.*

Gli input a un metodo sono detti *argomenti* o *parametri*, ed il risultato della chiamata è detto *valore di ritorno.* I metodi sono definiti in Ruby tramite la parola chiave `def`:

```
irb(main):026:0> def first_if_true(first, second, to_be_tested)
irb(main):027:1> if to_be_tested
irb(main):028:2> first
irb(main):029:2> else
irb(main):030:2* second
irb(main):031:2> end
irb(main):032:1> end
=> nil
```

Abbiamo appena definito un metodo chiamato `first_if_true`, che richiede tre parametri (chiamati rispettivamente `first`, `second` e `to_be_tested`) e ritorna il valore di `first` se `to_be_tested` è `true`, o il valore di `second` se `to_be_tested` vale `false`.
Con questo abbiamo definito il test multilinea come qualcosa di astratto che potrà essere riutilizzato con diversi valori. Proviamolo.

Si noti che il nome `first_if_true` *dice quello che farà il metodo. Questa è una buona abitudine da prendere. I nomi dei metodi dovrebbero sempre dire quello che fanno. Nomi di metodi chiari e intuitivi sono una parte importante della documentazione. Lo stesso suggerimento vale per le variabili, descritte in seguito. Secondo questo criterio,* `result` *(come vedremo poi), non è un così buon nome. Va bene per un semplice esempio che non fa altro che introdurre il concetto di assegnazione a una variabile, ma è eccessivamente vago per codice reale di produzione.*

Si ricordi che `first_if_true` verifica il terzo valore e ritorna il primo o il secondo.

```
irb(main):033:0> first_if_true(1, 2, true)
=> 1
irb(main):034:0> first_if_true(1, 2, false)
=> 2
irb(main):035:0> first_if_true(1, 2, nil)
=> 2
irb(main):036:0> first_if_true(nil, "Hello, world!", true)
=> nil
irb(main):037:0> first_if_true(nil, "Hello, world!", false)
=> "Hello, world!"
```

Sentivevi liberi di provare il metodo `first_if_true` in irb con diversi parametri, ora o dopo. Dovrebbe fornire una buona idea di come Ruby elabora le espressioni.

Mentre i metodi ritornano i valori quando vengono utilizzati, il semplice fatto di definire un metodo ritorna nil, come si può vedere.

Variabili

Cosa succederebbe se si volesse utilizzare l'output di un metodo come input di un altro? Uno dei modi più convenienti per fare questo è di utilizzare le *variabili*. In modo similare all'algebra o alla fisica, si decide di riferirsi a valori attraverso un nome, come `m` per una massa specifica o `v` per una data velocità. Si assegna il valore nella variabile con il segno =, come mostrato:

```
irb(main):038:0> result = first_if_true(nil, "Hello, world!", false)
=> "Hello, world!"
irb(main):039:0> result
=> "Hello, world!"
```

Abbiamo assegnato il valore di `first_if_true(nil, "Hello, world!", false)` in una variabile chiamata `result`. Ora c'è il valore `"Hello, world!"` memorizzato con il nome di `result`, la cui valutazione avviene come previsto, come si può vedere dalla linea 39. Ora è possibile utilizzare `result` come un qualsiasi altro valore.

```
irb(main):040:0> first_if_true(result, 1, true)
=> "Hello, world!"
irb(main):041:0> first_if_true(result, 1, result)
=> "Hello, world!"
```

Si noti come sia possibile passare `result` a `first_if_true` e valutarlo (come `to_be_tested`) come valore booleano. È possibile utilizzarlo anche come parte di un'espressione più grande:

```
irb(main):042:0> first_if_true( result, 1, (not result) )
=> 1
```

Nell'esempio alla linea 42, il valore booleano contenuto in `result` è stato rovesciato utilizzando la

parola chiave `not`, prima che venisse passato a `first_if_true`. Non è stata fatta alcuna modifica al `result` di riga 42. È stata solo creata una nuova espressione (`not result`) che produce come risultato l'opposto del valore di `result`. La variabile `result` in sé non cambia valore.

> *Sono stati aggiunti degli spazi solo per rendere più facile capire quali parentesi racchiudono i parametri del metodo e quali delimitano l'espressione (`not result`). Ruby e irb non si preoccupano invece molto degli spazi vuoti.*

Costanti

Qualche volta si desidera riferirsi a un valore per nome, ma non si ha la necessità di cambiarlo. Infatti, qualche volta si è certi di non doverlo cambiare. Buoni esempi dalla fisica sono la velocità della luce o l'accelerazione dovuta alla gravità della Terra: questi valori non cambiano. In Ruby è possibile definire questi valori come *costanti*, e devono iniziare con lettera maiuscola (tradizionalmente, spesso sono completamente in maiuscolo). Definiamo una costante e utilizziamola:

```
irb(main):043:0> HUNDRED = 100
=> 100
irb(main):044:0> first_if_true( HUNDRED.to_s + ' is true', false, HUN-
DRED )
=> "100 is true"
```

Si vede che è possibile assegnare una costante proprio come è stato fatto con una variabile. È possibile quindi utilizzarla attraverso il nome, come un'espressione, oppure come parte di una espressione più grande.

Utilizzare l'interprete Ruby e l'ambiente

Se venite da un'esperienza Unix, probabilmente avete familiarità con il concetto di opzioni a linea di comando e variabili d'ambiente. Se non avete familiarità con questi termini, sappiate che sono solo modi che il computer utilizza per tenere traccia di dati esterni, solitamente opzioni di configurazione. Ruby utilizza le *opzioni a linea di comando* e le *variabili d'ambiente* per tracciare cose come quanto essere paranoico o rilassato in merito alla sicurezza o quanto prolisso relativamente ai messaggi d'avviso. Sono già stati descritti esempi di questo nelle istruzioni per installare Ruby attraverso il download dei sorgenti, quando è stato eseguito questo comando:

```
ruby --version
```

Come ci si può attendere, questo non fa altro che richiedere a Ruby di stampare la propria versione. È possibile scoprire di più sulle varie opzioni a linea di comando supportate da Ruby eseguendo questo comando:

```
ruby -h
```

Le variabili d'ambiente possono memorizzare queste opzioni come default; inoltre, possono contenere altre informazioni non specifiche di Ruby, ma che esso può trovare utile per eseguire determinate attività. Gli utenti dei sistemi tipo Unix memorizzano i propri file all'interno di quella che viene chiamata la directory HOME, che tiene i propri dati fuori dalla portata di altri utenti. La cartella Documenti di

Windows è simile. Un'altra variabile d'ambiente molto importante è ARGV, che è un vettore che tiene traccia di tutti i parametri passati a Ruby. Quando si esegue un programma Ruby esterno, come spesso farete utilizzando la sintassi seguente, il nome del programma stesso si troverà in ARGV.

```
ruby some_external_program.rb
```

Ma passiamo a qualche esempio specifico di programma. Tratteremo con grande dettaglio ed esempi appropriati molti degli argomenti che in questo capitolo abbiamo solo sfiorato.