Fondamenti di Informatica (Elettronici)

THINK JULIA – Capitolo 7

4 novembre 2020

7. Iterazione[1]

[1] Tratto da https://benlauwens.github.io/ThinkJulia.jl/latest/book.html, disponibile sotto Licenza 'Creative Commons Attribution-NonCommercial 3.0 Unported'.

Questo capitolo riguarda l'iterazione, ovvero la capacità di eseguire ripetutamente un blocco di istruzioni.

Abbiamo visto una sorta di iterazione, usando la ricorsione, in Recursion.
Abbiamo visto un altro tipo, usando un ciclo `for`, in Simple Repetition.

In questo capitolo vedremo ancora un altro tipo, usando un'istruzione `while`.

Ma prima voglio dire qualcosa di più sull'assegnazione delle variabili.

Section 1

Riassegnazione

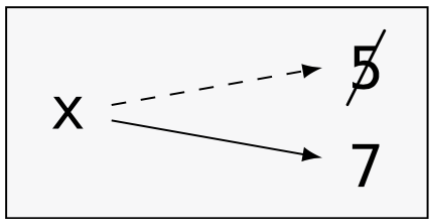
In Julia è legale riassegnare una variabile

Come forse avrete scoperto, è legale fare più di un'assegnazione alla stessa variabile.

Una nuova assegnazione fa in modo che una variabile esistente faccia riferimento a un nuovo valore (e smetta di fare riferimento al vecchio valore).

```
julia> x = 5
5
julia> x = 7
7
```

La prima volta che visualizziamo "`x`", il suo valore è 5; la seconda volta, il suo valore è 7.



La figura mostra l'aspetto della riassegnazione in un diagramma di stato.

Assegnazione versus uguaglianza

A questo punto consideriamo una comune fonte di confusione. Poiché Julia usa il segno di uguale (=) per l'assegnazione, si è tentati di interpretare un'affermazione come `a = b` come una proposizione matematica di uguaglianza; cioè l'affermazione che "a" e "b" sono uguali.

Ma questa interpretazione è sbagliata.

Primo, l'uguaglianza è una relazione simmetrica e l'assegnazione non lo è.

- Ad esempio, in matematica, se $a = 7$ allora $7 = a$. Ma in Julia, l'affermazione "`a = 7`" è legale e "`7 = a`" non lo è (PERCHÉ?)

Assegnazione versus uguaglianza

A questo punto consideriamo una comune fonte di confusione. Poiché Julia usa il segno di uguale (=) per l'assegnazione, si è tentati di interpretare un'affermazione come `a = b` come una proposizione matematica di uguaglianza; cioè l'affermazione che "a" e "b" sono uguali.

Ma questa interpretazione è sbagliata.

Primo, l'uguaglianza è una relazione simmetrica e l'assegnazione non lo è.

- Ad esempio, in matematica, se $a = 7$ allora $7 = a$. Ma in Julia, l'affermazione "`a = 7`" è legale e "`7 = a`" non lo è (PERCHÉ?)

Assegnazione versus uguaglianza

A questo punto consideriamo una comune fonte di confusione. Poiché Julia usa il segno di uguale (=) per l'assegnazione, si è tentati di interpretare un'affermazione come `a = b` come una proposizione matematica di uguaglianza; cioè l'affermazione che "a" e "b" sono uguali.

Ma questa interpretazione è sbagliata.

Primo, l'uguaglianza è una relazione simmetrica e l'assegnazione non lo è.

- Ad esempio, in matematica, se $a = 7$ allora $7 = a$. Ma in Julia, l'affermazione "`a = 7`" è legale e "`7 = a`" non lo è (PERCHÉ?)

Inoltre, in matematica, una proposizione di uguaglianza è o "vera" o "falsa" per sempre.

Fai attenzione con la riassegnazione di variabile

Se $a = b$ adesso, a sarà sempre uguale a b. In Julia, una dichiarazione di assegnazione può rendere due variabili uguali, ma non devono rimanere tali:

```
julia> a = 5
5
julia> b = a     # a and b are now equal
5
julia> a = 3 # a and b are no longer equal
3
julia> b
5
```

La terza riga cambia il valore di a ma non cambia il valore di b, quindi non sono più uguali.

AVVERTIMENTO

La riassegnazione delle variabili è spesso utile, ma andrebbe usata con cautela. Se i valori delle variabili cambiano frequentemente, può rendere difficile la lettura e il debug del codice.
Non è consentito definire una funzione con lo stesso nome di una variabile definita in precedenza.

Section 2

Aggiornamento delle variabili

Spesso la riassegnazione è un aggiornamento

Un tipo comune di riassegnazione è l'aggiornamento, in cui il nuovo valore della variabile dipende dal vecchio.

```
julia> x = x + 1
8
```

Ciò significa "ottenere il valore corrente di `x`, sommare 1, e quindi aggiornare `x` con il nuovo valore." Se proviamo ad aggiornare una variabile che non esiste, riceviamo un errore, perché Julia valuta il lato destro prima di assegnare un valore a `x`:

```
julia> y = y + 1
ERROR: UndefVarError: y not defined
```

Prima di poter aggiornare una variabile, si deve inizializzarla, di solito con un semplice assegnamento:

```
julia> y = 0
0
julia> y = y + 1
1
```

L'aggiornamento di una variabile aggiungendo 1 è chiamato incremento; sottrarre 1 è chiamato decremento.

Section 3

L'istruzione `while`

Ripetizione

I computer vengono spesso utilizzati per automatizzare attività ripetitive. Ripetere attività identiche o simili senza commettere errori è qualcosa che i computer fanno bene e le persone fanno male. In un programma per computer, ripetizione è anche chiamata iterazione.

Abbiamo già visto due funzioni, `countdown` e `printn`, che iterano usando la ricorsione.

Poiché l'iterazione è così comune, Julia fornisce funzionalità del linguaggio per renderla più semplice. Uno è l'istruzione `for` che abbiamo visto in Simple Repetition. Ci torneremo più tardi.

Un altro è l'istruzione `while`. Ecco una versione di `countdown` che usa l'istruzione `while`:

```
function countdown(n)
    while n > 0
        print(n, " ")
        n = n - 1
    end
    println("Blastoff!")
end
```

Ciclo

Puoi quasi leggere la dichiarazione `while` come se fosse inglese.

Significa, "Mentre `n` è maggiore di 0, visualizza il valore di `n` e decrementa `n`. Quando arrivi a 0, visualizza la parola `Blastoff!`"

Più formalmente, ecco il flusso di esecuzione per un'istruzione `while`:

1. Determina se la condizione è "true" o "false".

Questo tipo di flusso è chiamato `loop` perché il terzo passaggio ritorna all'inizio.

Ciclo

Puoi quasi leggere la dichiarazione `while` come se fosse inglese.

Significa, "Mentre `n` è maggiore di 0, visualizza il valore di `n` e decrementa `n`. Quando arrivi a 0, visualizza la parola `Blastoff!`"

Più formalmente, ecco il flusso di esecuzione per un'istruzione `while`:

1. Determina se la condizione è "true" o "false".
2. Se `false`, uscire dall'istruzione `while` e continuare l'esecuzione dall'istruzione successiva.

Questo tipo di flusso è chiamato `loop` perché il terzo passaggio ritorna all'inizio.

Ciclo

Puoi quasi leggere la dichiarazione `while` come se fosse inglese.

Significa, "Mentre `n` è maggiore di 0, visualizza il valore di `n` e decrementa `n`. Quando arrivi a 0, visualizza la parola `Blastoff!`"

Più formalmente, ecco il flusso di esecuzione per un'istruzione `while`:

1. Determina se la condizione è "true" o "false".
2. Se `false`, uscire dall'istruzione `while` e continuare l'esecuzione dall'istruzione successiva.
3. Se la condizione è `true`, eseguire il body e poi tornare al passaggio 1.

Questo tipo di flusso è chiamato `loop` perché il terzo passaggio ritorna all'inizio.

Fai attenzione con la variabile del ciclo

Il corpo del ciclo dovrebbe modificare il valore di una o più variabili in modo che la condizione alla fine diventi falsa e il ciclo termini.

Altrimenti il ciclo si ripeterà per sempre, ed è chiamato ciclo infinito.

Una fonte di divertimento per gli informatici è l'osservazione che le indicazioni sui flaconi di shampoo, "Insaponare, sciacquare, ripetere", sono un ciclo infinito.

Nel caso di "countdown", possiamo provare che il ciclo termina: se "`n`" è zero o negativo, il ciclo non viene mai eseguito.

Altrimenti, "`n`" si rimpicciolisce ogni volta nel ciclo, quindi alla fine dobbiamo arrivare a 0.

Fai attenzione con la variabile del ciclo

Per alcuni altri loop, non è così facile da dire. Per esempio:

```
function seq(n)
    while n != 1
        println(n)
        if n % 2 == 0 # n is even
            n = n/2
        else # n is odd
            n = n*3 + 1
        end
    end
end
```

La condizione per questo ciclo è "`n! = 1`", quindi il ciclo continuerà fino a quando `n` è 1, il che rende la condizione `false`.

Section 4

Istruzione break

Istruzione "break" per uscire dal ciclo

A volte non sappiamo che è ora di terminare un ciclo finché non arriviamo a metà del corpo. In questo caso possiamo usare l'istruzione "break" per uscire dal ciclo.

Ad esempio, supponiamo di voler ricevere input dall'utente finché non digita "done":

```
while true
    print("> ")
    line = readline()
    if line == "done"
        break
    end
    println(line)
end
println("Done!")
```

Controlla la condizione in qualsiasi punto del ciclo

La condizione del ciclo è "true", che resta sempre "true", quindi il ciclo viene eseguito finché non raggiunge l'istruzione "`break`".

Ogni volta viene richiesto input all'utente con una parentesi angolare. Se l'utente digita "done", l'istruzione `break` esce dal ciclo. Altrimenti il programma riproduce qualunque cosa l'utente digiti e torna all'inizio del ciclo. Ecco un esempio di esecuzione:

```
> not done
not done
> done
Done!
```

Questo modo di scrivere i cicli `while` è comune, perché si può controllare la condizione ovunque nel ciclo (non solo all'inizio) ed esprimere la condizione di arresto affermativamente ("fermati quando questo accade") piuttosto che negativamente ("continua finché capita").

Section 5

Istruzione continue

break and continue statements

L'istruzione `break` esce dal ciclo. Quando si incontra un'istruzione `continue` all'interno di un ciclo, il controllo salta all'inizio del ciclo per l'iterazione successiva, saltando l'esecuzione delle istruzioni all'interno del corpo del ciclo per l'iterazione corrente. Per esempio:

```
for i in 1:10
    if i % 3 == 0
        continue
    end
    print(i, " ")
end
```

Output:

```
1 2 4 5 7 8 10
```

Se `i` è divisibile per 3, l'istruzione `continue` arresta l'iterazione corrente e inizia l'iterazione successiva. Vengono stampati solo i numeri compresi tra 1 e 10 non divisibili per 3.

Section 6

Radici quadrate

Metodo di Newton per le radici quadrate

I cicli vengono spesso utilizzati nei programmi che calcolano i risultati numerici iniziando con una risposta approssimativa e migliorandola iterativamente.

Ad esempio, un modo per calcolare le radici quadrate è il metodo di Newton.

Supponi di voler conoscere la radice quadrata di `a`. Se inizi con una stima, `x <= a`, puoi calcolare una stima migliore con la seguente formula:

$$y = \frac{1}{2}\left(x + \frac{a}{x}\right).$$

Per esempio, se `a` è 4 e `x` è 3:

```
julia> a = 4
4
julia> x = 3
3
julia> y = (x + a/x) / 2
2.1666666666666665
```

Aggiornamento del ciclo

Il risultato è più vicino alla risposta corretta ($\sqrt{4} = 2$). Se ripetiamo il processo con la nuova stima, si avvicina ancora di più:

```
julia> x = y
2.1666666666666665
julia> y = (x + a/x) / 2
2.0064102564102564
```

Dopo qualche altro aggiornamento, la stima è quasi esatta:

```
julia> x = y
2.0064102564102564
julia> y = (x + a/x) / 2
2.0000102400262145
julia> x = y
2.0000102400262145
julia> y = (x + a/x) / 2
2.0000000000262146
```

In generale non sappiamo in anticipo quanti cicli occorrono per arrivare alla risposta giusta, ma lo sappiamo quando ci arriviamo perché la stima smette di cambiare:

Condizione di arresto

```
julia> x = y
2.0000000000262146
julia> y = (x + a/x) / 2
2.0
julia> x = y
2.0
julia> y = (x + a/x) / 2
2.0
```

Quando "y == x", possiamo fermarci. Ecco un ciclo che inizia con una stima iniziale, "x", e la migliora finché non smette di cambiare:

```
while true
    println(x)
    y = (x + a/x) / 2
    if y == x
        break
    end
    x=y
end
```

Testare l'uguaglianza del float è pericoloso

Per la maggior parte dei valori di a questo funziona bene, ma in generale è pericoloso testare l'uguaglianza dei float.

I valori in virgola mobile sono approssimativamente corretti: la maggior parte dei numeri razionali, come 1, e i numeri irrazionali, come $\sqrt{4} = 2$, non possono essere rappresentati esattamente con un "Float64".

Piuttosto che controllare se "`x`" e "`y`" sono esattamente uguali, è più sicuro utilizzare la funzione predefinita "`abs`" per calcolare il valore assoluto, o grandezza, della differenza tra loro:

```
if abs(y-x) < epsilon
    break
end
```

Dove `epsilon` (`\varepsilon TAB`) ha un valore come `0.0000001` che determina quando sono abbastanza vicini.

Section 7

Algoritmi

First steps . . .

Il metodo di Newton è un esempio di algoritmo: è un processo meccanico per risolvere una categoria di problemi (in questo caso, calcolare le radici quadrate).

Per capire cos'è un algoritmo, potrebbe essere utile iniziare con qualcosa che non è un algoritmo.

Quando hai imparato a moltiplicare numeri a una cifra, probabilmente hai memorizzato la tabella della moltiplicazione di 10 numeri.

In effetti, hai memorizzato 100 soluzioni specifiche. Questo tipo di conoscenza non è algoritmica.

Ma se tu fossi "pigro" (`lazy`), potresti aver imparato alcuni trucchi.

Ad esempio, per trovare il prodotto di `n` e 9, puoi scrivere come prima cifra "`n-1`" e "`10 - n`" come seconda cifra (es: `5 x 9 = 45`)

Questo trucco è una soluzione generale per moltiplicare qualsiasi numero a una cifra per 9. Questo è un algoritmo!

Pensiero algoritmico

Allo stesso modo, le tecniche apprese per l'addizione con il trasporto, la sottrazione con il prestito e la divisione lunga sono tutti algoritmi.

Una delle caratteristiche degli algoritmi è che** non richiedono** alcuna intelligenza per essere eseguiti. Sono processi meccanici dove ogni passaggio segue dall'ultimo secondo un semplice insieme di regole.

L'esecuzione degli algoritmi è noiosa, ma progettarli è interessante, intellettualmente impegnativo e una parte centrale dell'informatica.

Alcune delle cose che le persone fanno naturalmente, senza difficoltà o pensiero cosciente, sono le più difficili da esprimere algoritmicamente.

La comprensione del linguaggio naturale è un buon esempio.

Lo facciamo tutti, ma finora nessuno è stato in grado di spiegare come lo facciamo, almeno non sotto forma di algoritmo.

Section 8

Debug

Debug per bisezione

Quando inizi a scrivere programmi più grandi, potresti ritrovarti a dedicare più tempo al debug.

Più codice significa più possibilità di commettere un errore e più posti per nascondere i bug.

Un modo per ridurre i tempi di debug è il "debug per bisezione".

Ad esempio, se ci sono 100 righe nel tuo programma e le controlli una alla volta, ci vorranno 100 passaggi. Invece, prova a dividere il problema a** metà**.

Guarda al centro del programma, o vicino ad esso, per un valore intermedio che puoi controllare.

Aggiungi un'istruzione `print` (o qualcos'altro che abbia un effetto verificabile) ed esegui il programma.

Se il controllo del punto intermedio non è corretto, deve esserci un problema nella prima metà del programma. Se è corretto, il problema è nella seconda metà.

Divide et impera

Ogni volta che si esegue un controllo come questo, si dimezza il numero di righe da controllare.

Dopo sei passaggi (che sono meno di 100), dovresti essere ridotto a una o due righe di codice, almeno in teoria.

In pratica non è sempre chiaro quale sia la "metà del programma" e non sempre è possibile verificarlo.

Non ha senso contare le linee e trovare il punto medio esatto.

Invece, pensa ai punti del programma in cui potrebbero esserci errori e ai luoghi in cui è facile mettere una stampa.

Quindi scegli un punto in cui pensi che le probabilità siano le stesse che il bug si trovi prima o dopo il controllo.

Section 9

Glossario

Glossario I

riassegnazione Assegnazione di un nuovo valore a una variabile già esistente.

Glossario I

riassegnazione Assegnazione di un nuovo valore a una variabile già esistente.

aggiornamento Un'assegnazione in cui il nuovo valore della variabile dipende dal vecchio.

Glossario I

riassegnazione Assegnazione di un nuovo valore a una variabile già esistente.

aggiornamento Un'assegnazione in cui il nuovo valore della variabile dipende dal vecchio.

inizializzazione Assegnazione che fornisce un valore iniziale a una variabile che verrà aggiornata.

Glossario I

riassegnazione Assegnazione di un nuovo valore a una variabile già esistente.

aggiornamento Un'assegnazione in cui il nuovo valore della variabile dipende dal vecchio.

inizializzazione Assegnazione che fornisce un valore iniziale a una variabile che verrà aggiornata.

incremento Un aggiornamento che aumenta il valore di una variabile (spesso di uno).

Glossario I

riassegnazione Assegnazione di un nuovo valore a una variabile già esistente.

aggiornamento Un'assegnazione in cui il nuovo valore della variabile dipende dal vecchio.

inizializzazione Assegnazione che fornisce un valore iniziale a una variabile che verrà aggiornata.

incremento Un aggiornamento che aumenta il valore di una variabile (spesso di uno).

decremento Un aggiornamento che riduce il valore di una variabile.

Glossario I

riassegnazione Assegnazione di un nuovo valore a una variabile già esistente.

aggiornamento Un'assegnazione in cui il nuovo valore della variabile dipende dal vecchio.

inizializzazione Assegnazione che fornisce un valore iniziale a una variabile che verrà aggiornata.

incremento Un aggiornamento che aumenta il valore di una variabile (spesso di uno).

decremento Un aggiornamento che riduce il valore di una variabile.

iterazione Esecuzione ripetuta di una serie di istruzioni utilizzando una chiamata di funzione ricorsiva o un ciclo.

Glossario I

riassegnazione Assegnazione di un nuovo valore a una variabile già esistente.

aggiornamento Un'assegnazione in cui il nuovo valore della variabile dipende dal vecchio.

inizializzazione Assegnazione che fornisce un valore iniziale a una variabile che verrà aggiornata.

incremento Un aggiornamento che aumenta il valore di una variabile (spesso di uno).

decremento Un aggiornamento che riduce il valore di una variabile.

iterazione Esecuzione ripetuta di una serie di istruzioni utilizzando una chiamata di funzione ricorsiva o un ciclo.

istruzione `while` Istruzione che consente iterazioni controllate da una condizione.

Glossario I

riassegnazione Assegnazione di un nuovo valore a una variabile già esistente.

aggiornamento Un'assegnazione in cui il nuovo valore della variabile dipende dal vecchio.

inizializzazione Assegnazione che fornisce un valore iniziale a una variabile che verrà aggiornata.

incremento Un aggiornamento che aumenta il valore di una variabile (spesso di uno).

decremento Un aggiornamento che riduce il valore di una variabile.

iterazione Esecuzione ripetuta di una serie di istruzioni utilizzando una chiamata di funzione ricorsiva o un ciclo.

istruzione `while` Istruzione che consente iterazioni controllate da una condizione.

dichiarazione di interruzione Istruzione che consente di uscire da un ciclo.

Glossario II

Istruzione "continue" Istruzione all'interno di un ciclo che salta all'inizio del ciclo per l'iterazione successiva.

Glossario II

Istruzione "continue" Istruzione all'interno di un ciclo che salta all'inizio del ciclo per l'iterazione successiva.

ciclo infinito Un ciclo in cui la condizione di terminazione non è mai soddisfatta.

Glossario II

Istruzione "continue" Istruzione all'interno di un ciclo che salta all'inizio del ciclo per l'iterazione successiva.

ciclo infinito Un ciclo in cui la condizione di terminazione non è mai soddisfatta.

algoritmo Un processo generale per risolvere una categoria di problemi.

Section 10

Esercizi

aaaa

aaaa