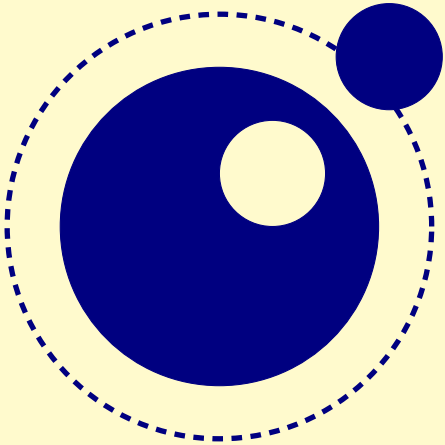
QUADERNI

di Informatica & Matematica

QIM
1

Programmare in LUA

Guida Pratica



Jilani KHALDI - www.khaldi.it - jilani@khaldi.it

Prima Edizione
Febbraio 2018

Sommario

Lua è un piccolo linguaggio di scripting con paradigma imperativo funzionale, molto facile da imparare ed estremamente flessibile per vari utilizzi. Lua è normalmente usato come linguaggio *embedded* (incorporato) per estendere le applicazioni a run-time. In effetti è molto usato in ambito PSP (*PlayStation Portable*) ed in tantissimi programmi di gioco e di grafica 3D.

Lua è anche usato come linguaggio autonomo in modo del tutto simile a PHP, Python e Ruby, ma è molto più semplice, leggero ed immediato da usare. Lua è forse l'unico linguaggio completo che si può imparare in una sola settimana. Quindi è ottimo per chi vuole imparare a programmare senza molto dispendio di tempo e di energie.

Questa libro è dedicato sia a chi vuole imparare Lua come primo linguaggio, sia a chi già conosce altri linguaggi.

- Formato pagine: A5 (148 x 210 mm)
- Numero pagine: 95
- Livello: introduttivo, medio.

Programmare in LUA # Guida Pratica # Jilani KHALDI # Prima Edizione - Febbraio 2018

PREFAZIONE

Ci sono tantissimi linguaggi di programmazione, forse un migliaio! In realtà quelli più usati sono solo una ventina e molti di loro si assomigliano. Ma se andassimo a vedere i paradigmi di programmazione invece, troveremmo che sono solo quattro, cioè:

1. Linguaggi imperativi con capostipite l'Algol.
2. Linguaggi funzionali con capostipite il LISP.
3. Linguaggi ad oggetti con capostipite Smalltalk.
4. Linguaggi dichiarativi con capostipite il Prolog.

Poi abbiamo i linguaggi ibridi, cioè che combinano diversi paradigmi di programmazione tipo il C++, Object Pascal, Ada, Python, Lua, Go, Dart, Julia e molti altri.

I linguaggi di programmazione si dividono poi in tre categorie, la prima è quella dei compilati, tipo C/C++, Object Pascal, Ada, Go, ecc, mentre la seconda è quella degli interpretati, tipo PHP, Python, PERL, Ruby, Lua, Javascript, Dart, ecc. La terza invece è quella dei

linguaggi compilati per una macchina virtuale tipo Java, Erlang, ecc. Ogni linguaggio è stato creato per sviluppare un genere di software specifico, esattamente come accade con gli attrezzi di lavoro. Quindi tocca al programmatore scegliere il linguaggio più adatto per il genere di software che intende sviluppare.

La scelta di imparare un linguaggio particolare dipende dal contesto e qui possiamo distinguere i quattro classici casi. Il primo è quello degli studenti che sono costretti ad imparare il linguaggio, oppure i linguaggi, imposti dai loro docenti. Il secondo caso è quello del programmatore che lavora in una azienda che produce software e quindi deve attenersi alle scelte imposte dai responsabili dello sviluppo. Il terzo caso è quello del programmatore freelance, oppure del libero professionista, che in generale sviluppa il software per il mercato oppure su specifiche richieste. Infine il quarto ed ultimo caso che è quello di chi programma per hobby. Quest'ultimo gode di estrema libertà e può scegliere il linguaggio che gli piace senza nessuna costrizione.

In generale, un programmatore professionista conosce diversi linguaggi e tutti i paradigmi di programmazione nonché è sempre pronto ad imparare nuovi linguaggi. Fortunatamente, la Rete offre di tutto e di più per rimanere sempre aggiornati ed attivi in quella che viene spesso chiamata l'arte magica, o ancora l'arte delle arti che è la programmazione dei computer.

Lua[1)] è veramente un ottimo linguaggio sia per il neofita, sia per l'esperto programmatore perché con solo una ventina di parole chiavi si riesce a scrivere piccole applicazioni di quasi ogni genere. Lua è anche un eccellente linguaggio da usare per descrivere dati dinamici. Infatti questa era la ragione che ha spinto il suo autore Roberto Ierusalimschy a crearlo nel lontano 1993.

[1)] Lua significa Luna in lingua portoghese.

Lua è ottimo anche per scrivere prototipi, applicazioni scientifiche e tecniche non troppo impegnative, per generare pagine dinamiche web e soprattutto per estendere applicazioni scritte in C/C++, Object Pascal, Ada, Go ed altri.

Per chi conosce qualsiasi linguaggio di programmazione, l'apprendimento di Lua risulta immediato, mentre per chi non ha mai programmato in precedenza risulterà semplice e comprensibile, più di qualsiasi altro linguaggio.

Una importante caratteristica di Lua è che l'intero codice è scritto in ANSI C. Questo significa che Lua è portabile ed è possibile usarlo con qualsiasi linguaggio che dialoghi con il C. Praticamente quasi tutti i linguaggi. Perciò Lua è molto usato soprattutto nelle industrie che producono videogame. In effetti, Lua permette di personalizzare alcune strategie e modificare il codice dell'intelligenza artificiale senza conoscere il codice sorgente del programma e senza ricompilazione. La lista di giochi che utilizzano Lua è ampissima. Ecco solo alcuni: Neverwinter Night, Dragon Age e World of Warcraft che è considerato il più grande gioco online con oltre dieci milioni di iscritti. Anche Corona sdk (www.coronalabs.com) il famoso framework di sviluppo per applicazioni mobile cross platform ha scelto come linguaggio base Lua.

Il fatto che Lua sia anche *freeware e open source* e che abbia una comunità vibrante di utenti e sostenitori garantisce al linguaggio un futuro molto roseo. In più abbiamo moltissime librerie scritte in Lua già pronte per l'uso e diversi siti web dedicati al linguaggi. Quindi vale proprio la pena imparare a programmare in Lua.

Ovviamente ci vuole tempo e motivazione per padroneggiare un nuovo linguaggio di programmazione anche quando si tratta di un

linguaggio così semplice come Lua.

L'obiettivo di pubblicare gratuitamente questo "quaderno" è quello di diffondere la cultura scientifica, soprattutto tra i giovani. Questo quaderno ed i suoi contenuti potrebbero essere utilizzati per qualsiasi scopo senza il consenso dell'autore.

Jilani KHALDI
San Salvo (CH) - ITALIA
Febbraio 2018

Contatti:
E. Mail: jilani@khaldi.it
Web: www.khaldi.it

Indice

Capitolo 1

INTRODUZIONE

> Bisogna rendere ogni cosa il più semplice possibile, ma non più semplice di quanto sia possibile.
>
> Albert Einstein

A differenza di molti altri linguaggi, Lua è nato molto piccolo, semplice e portabile e quindi veloce e facile da imparare. Anche durante tutta la sua evoluzione, raggiungendo oggi la versione 5.3, è rimasto tale conservando la stessa filosofia che l'ha fatto nascere.

1.1 Definizione del Linguaggio

Lua è un linguaggio procedurale con l'attitudine di descrivere i dati, nato per estendere applicazioni scritte in altri linguaggi compilati. Lua permette diversi paradigmi di programmazione, come quello funzionale, ad oggetti ed orientato alla manipolazione dei dati. Lua è

interamente scritto in ANSI C ed il codice è molto pulito ed ordinato.

Le due caratteristiche più importante di Lua sono: un linguaggio dinamico e non tipizzato. Un linguaggio dinamico significa che il codice sorgente viene direttamente eseguito dall'interprete senza compilazione; non tipizzato invece significa che l'interprete non effettua nessun controllo sul tipo di una variabile prima dell'esecuzione del codice. Nei linguaggi statici invece, tipo il C/C++, Object Pascal, Ada[1)]... il codice sorgente è prima compilato e poi eseguito, in più tutte le variabili, rigorosamente con nome e tipo, debbono essere dichiarati prima dell'uso. Durante la fase di compilazione, in generale, i linguaggi statici effettuano dei controlli sui tipi dei dati ed il loro uso ed in caso di errori oppure di incongruenza segnalano gli errori. Questo fatto evita a priori una vastissima serie di errori.

Ci sono diverse scuole di pensiero al riguardo dei linguaggi statici e dinamici, personalmente preferisco i linguaggi statici per grandi progetti, mentre per piccoli progetti e prototipazione potrebbero andar bene anche i linguaggi dinamici.

NOTA BENE

I linguaggi dinamici, Lua in particolare, diventano la scelta obbligata quando si tratta di estendere applicazioni compilate. ♦

1.2 Parole Chiavi

Il set di parole chiavi di Lua è molto ridotto:

[1)] Ada è proverbiale per il suo controllo sui tipi dei dati e la loro congruenza.

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until     while
```

Ed ecco il set dei simboli:

```
+     -     *     /     %     ^     #
==    ~=    <=    >=    <     >     =
(     )     {     }     [     ]
;     :     ,     .     ..    ...
```

Precedenza degli operatori:

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
*     /     %
not   #     - (unary)
^
```

1.3 Caratteristiche di Lua

Trovo molto interessanti due caratteristiche di Lua. La prima consiste nell'uso delle tabelle come strutture dati generiche che permette con estrema flessibilità la gestione e la manipolazione di qualsiasi tipo di dati. La seconda caratteristica è quella di poter interpretare le funzioni come oggetti di primo livello ed assegnarli a variabili. In questo modo Lua ci offre qualche chicca per scrivere codice quasi orientato agli oggetti senza essere realmente un OOP.[2)].

[2)] Object Oriented Programming, ossia Linguaggio Orientato agli Oggetti.

NOTA BENE

Le tabelle sono l'unica struttura dati definita dal linguaggio.

Lua, essendo un linguaggio procedurale e funzionale, si presta molto bene per scrivere piccole applicazioni scientifiche et tecniche.
Perciò nella nostra trattazione, saranno molto frequenti esempi di calcolo numerico.

PROSSIMO CAPITOLO

Installazione

Capitolo 2

INSTALLAZIONE

In questo capitolo vediamo come installare l'interprete Lua e fare il primo passo.

2.1 Dove Trovare Lua

Lua è scaricabile in diversi formati dal sito ufficiale del linguaggio (http://www.lua.org/download.html). Abbiamo due possibilità: la prima è quella di scaricare Lua già compilato in formato binario e di scompattare il file in una nuova directory, oppure in una esistente ed inserire il suo percorso nelle variabili d'ambiente. In questo modo possiamo usare Lua da qualsiasi applicazione installata nel nostro computer.

ATTENZIONE

*Per la compilazione di Lua da codice sorgente bisogna aver già installati **mingw** e **msys** per l'emulazione dell'ambiente UNIX sotto Windows.* ♠

La seconda possibilità è quella di compilare Lua da codice sorgente. In questo caso sarà necessario scaricare il file "lua-5.x.x.tar.gz", ma prima della compilazione da codice sorgente, però, assicuratevi di aver già installati **mingw** e **msys**. Questi due programmi emulano l'ambiente UNIX sotto Windows. La compilazione del codice Lua è semplicissima basta seguire i seguenti quattro passi procedere nel modo seguente:
1) Scompattare il file contenente il codice sorgente in una directory.
2) Aprire un *Prompt dei comandi*.
3) Accedere alla sottodirectory dove si trova il codice sorgente di Lua.
4) Digitare dalla riga di comando *make mingw* seguito da invio ed aspettare che finisca la compilazione.

In assenza di errore il programma compilato (*lua.exe*), la corrispondente DLL (**lua5x.dll**) ed il compilatore *luac.exe* si troveranno nella stessa sottodirectory del codice sorgente.

NOTA BENE

Per un normale utilizzo di Lua conviene sempre utilizzare la versione compilata disponibile sul sito www.lua.org

2.2 File Binari

Una volta che il codice sorgente di Lua è stato compilato, ci troviamo con i seguenti tre file binari:

- **lua.exe** è l'interprete che esegue un programma scritto in Lua.
- **lua5x.dll** è la libreria dinamica du supporto all'interprete.

- **luac.exe** è il compilatore che traduce un programma Lua in un formato binario eseguibile con l'interprete Lua.

A questo punto Conviene spostare questi tre file in una directory che si trova nel percorso (PATH) del vostro sistema in modo che si possano usare da qualsiasi applicazione installa sul vostro computer.

NOTA BENE

*Il programma **luac.exe** serve a tradurre un programma Lua da codice sorgente (leggibile) a codice binario (illeggibile) ma che rimane sempre eseguibile dall'interprete **lua.exe***

2.3 I Primi Passi

Per assicurarsi che Lua è stato installato correttamente nel nostro sistema, basta aprire un prompt dei comandi e digitare "Lua" seguito dal tasto invio e subito dovrebbe apparire una schermata simile alla seguente:

```
Microsoft Windows [Versione 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.
Tutti i diritti riservati.

C:\Users\jilani>cd\lua (+invio)

C:\lua>lua (+invio)
Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio
```

2.4 Ciao Mondo!

Ora proviamo a scrivere il classico "Ciao Mondo!" in Lua.

```
C:\luaexp>lua
Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio

> print "Ciao Mondo!" (+invio)
Ciao Mondo!
>
```

Ora che il primo passo è stato fatto, procediamo con la scelta dell'editor.

2.5 Editor per Lua

Ora che abbiamo l'interprete Lua installato e funzionante, dobbiamo procurare un editor di testo per cominciare a scrivere le nostre applicazioni.

La scelta è vastissima, ma rimanendo nel mondo del freeware, personalmente preferisco l'ottimo l'editor SciTE (http://www.scintilla.org).

Questo editor, oltre ad essere gratuito ed open source, è utilizzabile anche per codificare in tanti altri linguaggi di programmazione nonché usa proprio il linguaggio Lua per essere esteso e personalizzato.

Altri due ottimi editor per Lua molto simile a **SciTE** sono:

- **Notepad++** (https://notepad-plus-plus.org)
- **PSP** (http://www.pspad.com/en/).

Il mio consiglio è quello di provarli tutti i tre, e magari anche altri editor, poi decidere quello che va per voi. Personalmente trovo il grande vantaggio che ha **SciTE** rispetto agli altri due è la possibilità

di estenderlo utilizzando proprio il linguaggio Lua.

Ora che avete scelto il vostro editor possiamo iniziare a programmare in Lua.

■

PROSSIMO CAPITOLO

Primi Passi ★

Capitolo 3

PRIMI PASSI

In questo capitolo vediamo come lavorare con i numeri e le stringhe con le operazioni di base.

3.1 Numeri

Ora vediamo come lavorare con i numeri ed eseguire alcune operazioni aritmetiche.

```
> print(1+1)
2
> print(2*3)
6
> print(2/3)
0.66666666666667
> print(2-3)
-1
> print(2.71828+3.14159)
5.85987
> print(2.71828*3.14159)
8.5397212652
```

```
> print(2^16)
65536.0
> print(2e16)
2e+016
> print(2.71828^3.14159)
23.140582326247
>
```

Possiamo anche usare numeri espressi nel sistema esadecimale:

```
> print(0x10)
16
> print(0xA)
10
> print(0xF)
15
> print(0xFF)
255
> print(0xFFED)
65517
> print(0xFFED*0xEDA)
249095634
> print(0xFFED+0xEDA)
69319
> print(0xFFED/0xEDA)
17.232246186218
> print(0xFFED-0xEDA)
61715
> print(0xFF^0xE)
4.9154414350646e+033
>
```

3.1.1 Numeri Infiniti

Il numero massimo da non superare programmando in Lua è 2^{1024}. Questo limite è dovuto all'uso da parte di Lua di numeri a virgola

mobile definiti come IEEE 64-bit. Quindi quando il risultato di un'operazione supera tale valore Lua restituisce **inf**, che sta per "infinito", come nel caso della divisione di un numero per zero. Il risultato dell'operazione 0/0 è invece **NaN**[1)], cioè indeterminato.

```
> print(0xFFED^0xEDA)
inf
> print(1/0)
inf
> print(0/0)
nan
>
```

3.2 Assengnazione di Variabili

In Lua l'assegnazione di una variabile, ossia l'attribuzione di un valore ad una variabile, avviene mediante il segno "=".

```
> x=3.14159 -- pi greco
> y=2.71828 -- e
> z=1.61803 -- rapporto d'oro
> t=x+y+z
> v=x*y/z
> w=(x-y)/z
> print(x)
3.14159
> print(y)
2.71828
> print(z)
1.61803
> print(t)
7.4779
> print(v)
5.2778510072125
```

1) Not a Number.

```
> print(w)
0.26162061272041
>
```

ATTENZIONE

*Il linguaggio Lua è di tipo case sensitive, cioè fa distinzione tra lettere minuscole e maiuscole. In altri termini, le variabili **a** e **A** sono considerate due variabili differenti, mentre nel Pascal, per esempio, si tratta della stessa variabile.*

L'assegnazione multipla delle variabili può avvenire sulla stessa linea:

```
> somma, prodotto, inverso = x+y+z, x*y*z, 1/(x+y+z)
> print(somma, prodotto, inverso)
7.4779  13.817525198732 0.13372738335629
>
```

Assegnazione con scambio delle variabili:

```
12      15
> x, y = y, x
> print(x, y)
15      12
>
```

L'assegnazione delle variabili di tipo stringa avvengono allo stesso modo:

```
> nome="Jilani"
> cognome="KHALDI"
> print(nome..cognome)
JilaniKHALDI
> print(nome.." "..cognome)
```

```
Jilani KHALDI
> print(nome,cognome)
Jilani  KHALDI
> print(nome + cognome)
stdin:1: attempt to perform arithmetic on a string value
(global 'nome') stack traceback:
stdin:1: in main chunk
[C]: in ?
> nc=nome.." "..cognome
> print(nc)
Jilani KHALDI
>
```

ATTENZIONE

In Lua la somma di due stringhe non si ottiene mediante il segno '+' come se fossero dei numeri ma con '..'. Non tutti i linguaggio però si comportano in questo modo.

Quotare una stringa all'interno di un'altre:

```
print('Il mio nome è "nessuno" ed il tuo?')
=> Il mio nome è "nessuno" ed il tuo?
```

3.3 Sequenze *Esacpe*

A volte l'uso delle sequenze di *escape* diventa necessario per ottenere certi risultati con le stringhe:

```
print("Ciao \"Mamma\"!")
=> Ciao "Mamma"!
print('All\'ombra dell\'ultimo Sole...')
=> All'ombra dell'ultimo Sole...
```

Ecco le sequenze di *escape* più usate:

Tabella 3.1: Sequenze Escape più usate

Sequenza	**Significato**
\n	Nuova linea
\r	Tasto 'Invio'
\t	Tasto 'Tab orizzontale'
\v	Carattere 'Tab verticale'
\	Tasto 'invio'
\\r	backslash
\"	Doppia quota
\'	Singola quota
\numero	Caratteri ASCII corrispondente

Esempio:

```
print(\123\125)
=> {}
```

3.4 Confronto tra Numeri

Ci sono 6 operatori per confrontare due numeri:

```
A < B (A è minore di B?)
A > B (A è maggiore di B?)
A <= B (A é minore o uguale di B?)
A >= B (A é maggiore o uguale di B?)
A == B (A é uguale B?)
A ~= B Is (A non è uguale a B?)
```

Esempi:

```
print(3 < 5) => true
print(3 > 5) => false
```

```
print(3 <= 5) => true
print(3 >= 5) => false
print(3 > 5) => false
print(3 == 5) => false
print(3 ~= 5) => true
```

3.5 Confronto tra Stringhe

Si usano gli stessi 6 operatori per confrontare i numeri per confrontare le stringhe che debbono essere quotate:

```
print("abc" < "bcd") => true
print("abc" > "bcd") => false
print("abc" <= "bcd") => true
print("abc" >= "bcd") => false
print("abc" == "bcd") => false
print("abc" ~= "bcd") => true
```

3.6 Il Valore Nil

Lua attribuisce il valore **nil** ad una variabile inesistente oppure non inizializzata. Esempi:

```
print(x) => nil
x=3.14
-- ora che x è stata inizializzata
print(x) => 3.14
A, B, C = 3.14159, 2.71828
=> 3.14159 2.71828 nil
```

NOTA BENE

*Il valore zero non è considerato un valore **nil**, ma è un valore numerico. Il valore 'nil' potrebbe essere definito come 'inesistente'.*

3.7 Operatori Booleani

Gli operatori booleani **and** (e), **or** (o) e **not** (non) si possono usare sia con le variabili booleane (vero e falso) sia con altri valori. Esempi:

```
-- and
print(true and true)
=> true
print(true and false)
=> false
print(false and true)
=> false
print(false and false)
=> false
-- or
print(true or true)
=> true
print(true or false)
=> true
print(false or true)
=> true
print(false or false)
=> false

-- and
print(true and nil)
=> nil
print(nil and true)
=> nil
print(false and nil)
=> false
print(nil and false)
=> nil
-- or
print(true or nil)
```

```
=> true
print(nil or true)
=> true
print(false or nil)
=> false
print(nil or false)
=> nil
```

Gli operatori booleani servono per il confronto tra due valori ed anche controllare se una condizione è verificata o meno. Esempio:

```
pi = 3.14
e = 2.71
x = 3
print (x > e and x < pi)
=> true
x = 2
print (x > e and x < pi)
=> false
```

3.8 Concatenamento

Il concatenamento tra due stringhe avviene mediante due punti consecutivi. Ecco un esempio:

```
nome = "Mario"
cogome = "Rossi"
print(nome..cogome)
=> MarioRossi
Per avere il nome ed il cognome separati:
nome = "Mario "
cogome = "Rossi"
print(nome..cogome)
=> Mario Rossi
```

```
oppure:
nome = "Mario"
cogome = " Rossi"
print(nome..cogome)
=> Mario Rossi
oppure:
nome = "Mario"
cogome = "Rossi"
print(nome..' '..cogome)
=> Mario Rossi
```

3.9 Operatore

L'operatore # misura la lunghezza di una stringa. Ecco la sua sintassi con alcuni esempi:

```
print(#"") => 0
print(#"#") => 1
print(#"5") => 1
print(#"\n") => 1
print(#"1/6") => 3
print(#"Roma") => 4
print(#"11/12/2004") => 10
print(#"3.14159") => 7
```

NOTA

L'uso di "\n" è il seguente: posizionato all'inizio di una frase significa "salta una linea prima di scrivere questa frase"; mentre posizionato alla fine significa "salta una linea dopo aver scritto la frase".

3.10 if...else

L'istruzione **if...else** (se...altrimenti) permette di eseguire uno specifico codice in base alla verità o meno di una certa. La sua sintassi è la seguente:

```
if condizione then
  codice1
else
  codice2
end
end
```

Oppure nel caso in cui ci solo più di due condizioni allora l'istruzione diventa:

```
if condizione then
  codice1
else if
  codice2
end
else if
  ...
else
 ...
end
end
```

Esempio:

```
x = 3.14
y = 2.718
if x == y then
print("x è uguale a y")
else if x > y then
print("x è maggiore di y")
else
```

```
print("y è maggiore di x")
end
end
=> x è maggiore di y
```

3.11 Ciclo *while…do*

L'istruzione *while* permette di eseguire un ciclo di istruzione finché una certa condizione rimanga vera. La sua sintassi è la seguente:

```
while condizione do
  esegue il codice
end
```

Esempio:

```
n=10
while n > 0 do
  print(n)
  n = n-1
end
=> stampa i numeri da 10 a 1.
```

Ecco un altro esempio un po' più complicato del precedente ma più interessante:

```
N, F = 1, 1
while F < 130 do
  print(N .."! = ".. F)
  N = N + 1
  -- Calcola il fattoriale di N
  F = F * N
end
```

3.12 Ciclo *for...do*

L'istruzione *for* (fino a) permette di eseguire un numero di cicli prestabili. La sua sintassi è la seguente:

```
for condizione do
  esegue il codice
end
```

Il seguente esempio calcolo i quadrati da 1 a N:

```
N=5
for i = 1,N do
  print(i.."^2 = "..i*i)
end
=>
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
5^2 = 25
```

NOTA BENE

*Un errore comune è quello di dimenticare di scrivere l'istruzione **do** dopo **for**. Quindi Lua è simile al Pascal da questo punto di vista.*

3.13 Ciclo *repeat...until*

L'istruzione *repeat...until* (ripeti...finché) permette di eseguire un cicli finché una condizione rimanga vera. La sua sintassi è la seguente:

```
N=5
i=1
repeat
  print(i.."^2 = "..i*i)
  i = i+1
until i == N
=>
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
```

NOTA BENE

La differenza tra i cicli repeat...until e for...to è che nel primo la condizione è controllata all'inizio del ciclo, mentre nel secondo il controllo della condizione avviene alla fine del ciclo.

Quindi, l'ultimo valore è 16, mentre nell'esempio precedente è 25.

3.14 Istruzione *break*

L'istruzione **break** interrompe un ciclo alla verifica di una certa condizione. La sua sintassi è la seguente:

```
inizia un ciclo
  esegue il codice
    se si verifica una condizione allora
      break (esci dal ciclo)
  finisci
finisci
```

Ed ecco un esempio:

```
N=100
for i = 1,N do
  print(i.."^2 = "..i*i)
  if i == 5 then
    break
  end
end
=>
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
5^2 = 25
```

Normalmente il ciclo dovrebbe continuare fino a N=100, ma la presenza dell'istruzione '*if i == 5 then*' lo interrompe quando N=5.

NOTA BENE

L'istruzione break deve essere piazzata nel punto giusto di un ciclo visto che provoca la brusca interruzione di un ciclo con una parte del codice che non sarà eseguita. ♦

Le istruzioni "*if...then...else*" e quelle dell'esecuzione dei cicli "*while...do*", "*for...end*" e "*repeat...until*" sono alla base di qualsiasi linguaggio di programmazione e sono sempre presenti.

NOTA

Anche i linguaggi dichiarativi come il Prolog, oppure quelli funzionali come Erlang che per loro natura non hanno le istruzioni "while...do", "for...to" e "repeat...until", sono comunque in grado di eseguire cicli mediante ricorsione. ♣

Dopo questa breve introduzione, il modo miglior di imparare a programmare in Lua è quello di iniziare a sperimentare modificando il codice degli esempi di questo capitolo ed osservare i risultati.

■

PROSSIMO CAPITOLO

Funzioni ★

Capitolo 4

FUNZIONI

In questo capitolo vediamo come lavorare con le funzioni che rappresentano l'essenza della programmazione in Lua.

Le funzioni sono presenti in tutti i linguaggi di programmazione. Una semplice definizione di una funzione potrebbe essere la seguente: una funzione è un frammento di codice che ha un nome, che prende nessuno, uno oppure più variabili in ingresso, esegue una o più operazioni e restituisce uno o più valore in uscita.

NOTA

*Possiamo definire una **funzione** come un unità di elaborazione all'interno di un programma.* ♣

Alcuni programmi come Lua, permettono di annidare le funzioni, cioè permettono ad una funzione di contenere al suo interno una o più funzioni. Il linguaggio **C**, per esempio, non permette una cosa del genere, mentre il **Pascal** lo permette.

Il modo migliore per capire l'uso delle funzioni in Lua è quello di dare una serie di esempi semplici per passare in seguito a quelli più complessi.

4.1 Esempi Pratici

Il primo esempio è quello di una funzione che non prende nessun valore:

```
function CiaoMamma()
  print("Ciao mamma!")
end
CiaoMamma() => Ciao mamma!
```

In questo secondo esempio la funzione **saluta** prende una stringa come variabile, le aggiunge la stringa "Ciao" ed il punto esclamativo alla fine, poi la stampa.

```
function saluta(soggetto)
st = "Ciao "..soggetto.."!"
print(st)
end
saluta("mamma")
=> Ciao mamma!
```

NOTA BENE

Essendo Lua un linguaggio dinamico, le variabili possono essere creati al volo, senza tipo e praticamente in qualsiasi punto del programma. Questa peculiarità costituisce un punto di forza del linguaggio, ma anche tanti punti di debolezza.

Ecco un altro esempio:

```
function SommaEProdotto(x, y)
  s = x+y
  p = x*y
  print("Somma = "..s)
  print("Prodotto = "..p)
end
SommaEProdotto(3.14159,2.71828)
Risultato:
Somma = 5.85987
Prodotto = 8.539721265199999
```

Il risultato dell'esempio precedente potrebbe essere scritto nel modo seguente:

```
function SommaEProdotto(x, y)
  s = x+y
  p = x*y
  print(x.." + "..y.." = "..s)
  print(x.." x "..y.." = "..p)
end
SommaEProdotto(3.14159,2.71828)
Risultato:
3.14159 + 2.71828 = 5.85987
3.14159 x 2.71828 = 8.539721265199999
```

NOTA BENE

Bisogna sempre cercare di dare nomi significativi e contestuali alle variabili ed alle funzioni per rendere il codice più chiaro e leggibile.

4.2 Istruzione *return*

L'istruzione **return** restituisce il risultato di una funzione. Questo risultato potrebbe essere composto da un singolo o più valori.

Ora scriviamo gli esempi precedenti usando l'istruzione **return**:

```
1)
function saluta(soggetto)
  st = "Ciao "..soggetto.."!"
  -- restiuisci Ciao mamma!
  return st
end
saluta("mamma")
=> Ciao mamma!

2)
function saluta(soggetto)
  st = "Ciao "..soggetto.."!"
  -- restiuisci Ciao mamma!
  return st
end
genitori = "mamma e papà"
viSaluto = saluta(genitori)
print(viSaluto)
=> Ciao mamma e papà!

3)
function SommaEProdotto(x, y)
  s = x+y
  p = x*y
  return s, p
end
a, b = 3.14159, 2.71828
somma, prodotto = SommaEProdotto(a,b)
print(a.." + "..b.." = "..somma)
print(a.." x "..b.." = "..prodotto)

Risultato:
3.14159 + 2.71828 = 5.85987
```

```
3.14159 x 2.71828 = 8.539721265199999
```

L'istruzione **return** è sempre presente in una funzione anche quando non è dichiarata in modo esplicito. Ecco un piccolo esempio:

```
function somma(x,y)
  print(x+y)
end
t=somma(2,3)
print(t)
Risultato:
5
nil

invece:

function somma(x,y)
  print(x+y)
  return x+y
end
t=somma(2,3)
print(t)
Risultato:
5
5
```

NOTA BENE

*Nel caso in cui l'istruzione **return** non è presente all'interno di una funzione, il suo valore è **nil**, cioè nullo. In altri termini, la funzione restituisce un valore nullo.*

ATTENZIONE

*Il valore **nil** restituito da una funzione oppure da una variabile non corrisponde affatto al valore **zero**, ma semplicemnete ad un valore indeterminato. Il valore **zero**, invece, è un valore numerico ben definito che è quello matematico.*

Ora vediamo questo l'esempio precedente leggermente modificato, cioè l'istruzione **return** è stata sposta prima di **print**:

```
function somma(x,y)
  return x+y
  print(x+y)
end
t=somma(2,3)
print(t)
Risultato:
line:3(column:7) near 'print': syntax error
```

L'interprete di Lua ci restituisce un errore perché l'istruzione **return** deve sempre concludere una funzione.

Ora vediamo un altro esempio più interessante:

```
-- risolvere l'equazione di primo grado ax+b=0
function EquazioneDiPrimoGrado(a,b)
  if a == 0 then
    return 0, 0
  else
    return 1, -b/a
  end
end
a1, b1 = 0,2
soluzione, valore = EquazioneDiPrimoGrado(a1,b1)
```

```
if soluzione == 0 then
  print("a = 0, non è un'equazione di primo grado.")
else
  print("x = "..valore)
end
```

Il codice è molto semplice e non necessita particolari commenti.

4.3 Esercizi Proposti

A questo punto il lettore, dovrebbe essere in grado a scrivere delle piccole ma significative applicazioni per applicare le conoscenze acquisite fin qui. Si suppone che il lettore non abbia conoscenza di altri linguaggi di programmazione, ma ovviamente, saper programmare a qualsiasi livello in qualsiasi altro linguaggio, soprattutto se di tipo procedurale, rende Lua molto familiare.

Ecco alcuni esercizi che il lettore potrebbe tentare di risolvere sviluppando le soluzioni in Lua:

- Risolvere un'equazione di secondo grado.
- Trovare le intersezioni tra una retta ed una parabola.
- Trovare il MCD (Massimo Comune Divisore) ed il mcm (minimo comune multiplo) di due numeri.

Comunque questi esercizi saranno risolti nel capitolo 11 "Programmazione Scientifica".

■

PROSSIMO CAPITOLO

Tabelle ★

Capitolo 5

Tabelle

In questo capitolo vediamo come lavorare con le tabelle che sono l'unica struttura dati in Lua. Quindi, questo capitolo è molto importante visto che tutta la manipolazione e gestione dei dati in Lua avviene mediante le tabelle.

In questo capitolo vediamo come:

- Creare e modificare una tabella.
- Scorrere gli elementi di una tabella.
- Usare la libreria di funzioni dedicate alle tabelle.

5.1 Creare una Tabella

Una tabella in Lua è un array associativo, cioè un array che può essere indicizzato sia con numeri, sia con lettere, sia con stringhe, **nil** escluso.

Il modo più semplice per creare una tabella vuota è il seguente:

```
a = {} -- a referenzia la tabella vuota
```

Si può popolare la tabella appena creata nel modo seguente:

```
a[1] = 3.14
a[2] = "Ciao Lua!"
a["nome"] = "Mario"
a["cognome"] = "Rossi"
a[12] = "Vedrai, vedrai che cambierà..."
print(a[1]) => 3.14
print(a[2]) => Ciao Lua!
print(a["nome"]) => Mario
print(a["cognome"]) => Rossi
print(a[12]) => Vedrai, vedrai che cambierà...
print(a[3]) => nil

Ecco un altro esempio:
giorni = {"Lunedì", "Martedì", "Mercoledì", "Giovedì",
 "Venerdì", "Sabato", "Domenica"}
for i=1,7 do
  print(giorni[1])
end
Risultati:
Lunedi
Martedi
Mercoledi
Giovedi
Venerdi
Sabato
Domenica
```

Possiamo referenziare la tabella "a"con una nuova semplice assegnazione, cioè:

```
b=a -- ora b referenzia a
print(b[1]) => 3.14
print(b[2]) => Ciao Lua!
```

```
print(b["nome"]) => Mario
print(b["cognome"]) => Rossi
print(b[12]) => Vedrai, vedrai che cambierà...
print(b[3]) => nil
```

Ora se assegniamo alla tabella "a" il valore **nil**, solo "b" continua a referenziale la tabella.

NOTA BENE

Una tabella è sempre anonima. Cioè non esiste una relazione fissa tra la variabile tabella e la tabella stessa. ♦

NOTA

Per avere una tabella convenzionale, cioè una matrice, basta che suoi indici siano solo degli interi. ♣

Un altro modo di creare ed inizializzare una tabella in Lua è il seguente:

Linea = ID=0, x1=0.0, y1=0.0, x2=0.0, y2=0.0

In questo modo abbiamo definito il record **Linea** con i parametri inizializzati.

È possibile popolare una tabella in modo automatico. Ecco un esempio che crea un vettore di 100 numeri interi (da uno a cento) creato mediante un ciclo **for** e che contiene i quadrati degli indici:

```
v={} -- crea una tabella vuota
for i=1,100 do
  v[i] = i*i
```

```
  print("n = "..i.." => n*n = "..i*i)
end
Risultato:
n = 1 => n*n = 1
n = 2 => n*n = 4
n = 3 => n*n = 9
...
n = 100 => n*n = 10000
```

Le tabelle in Lua sono strutture dinamiche molto flessibili e permettono di manipolare vari tipi di dati con molta facilità.

5.2 Operazioni su Tabelle

Ora vediamo come manipolare i dati all'interno delle tabelle.

5.2.1 Dimensioni Tabella

Sapere le dimensioni di una tabella a priori rende le operazioni più veloci soprattutto quando si tratta di eseguire dei cicli, ma a volte le dimensioni non sono noti oppure sono modificate dinamicamente. In questo caso bisogna sapere le dimensioni di una tabella nel momento opportuno. Ci sono due modi per sapere le dimensioni di una tabella.

```
t={1,2,3,4}
print(table.getn(t)) --> 4
-- oppure
print(#t) --> 4
```

5.2.2 Concatenazione

per concatenare gli elementi di una tabella si usa il comando *concat*. Ecco degli esempi:

```
giorni = {"Lunedì", "Martedì", "Mercoledì", "Giovedì",
"Venerdì", "Sabato", "Domenica"}

-- Concatena tutti gli elementi
print(table.concat(giorni))

-- Concatena con un carattere
print(table.concat(giorni,", "))

-- Concatena in base ad un indice
print(table.concat(giorni,", ", 1,3))

-- Concatena in base ad un indice
print(table.concat(giorni,", ", 4,6))

Risultati:
LunediMartediMercoledìGiovediVenerdìSabatoDomenica
Lunedì, Martedì, Mercoledì, Giovedì, Venerdì, Sabato,
Domenica
Lunedì, Martedì, Mercoledì
Giovedì, Venerdì, Sabato
```

5.2.3 Inserimento e Rimozione

L'inserimento e la rimozione di elementi sono operazioni molto comuni nelle manipolazioni delle tabelle. I rispettivi comandi sono *insert* e *remove*. Ecco alcuni esempi:

```
giorni = {"Lunedì", "Martedì", "Giovedì",
"Venerdì", "Sabato"}

-- inserisci la domenica
table.insert(giorni,"Domenica")
print("Il giorno inserito alla fine è ",giorni[6])

-- inserisci al terzo giorno Mercoledì
```

```
table.insert(giorni,3,"Mercoledì")
print("Il giorno inserito all'indice 3 è ",giorni[3])

print("Numero massimo = ",table.maxn(giorni))

print("L'ultimo giorno è ",giorni[7])

-- Elimina l'ultimo giorno
n=table.maxn(giorni)
table.remove(giorni)
print("L'ultimo giorno ora è ",giorni[n-1])

-- Elimina il Mercoledì
table.remove(giorni,3)
print("Ora all'indice 3 abbiamo ",giorni[3])

Risultati:
Il giorno inserito alla fine è Domenica
Il giorno inserito all'indice 3 è Mercoledi
Numero massimo =  7
L'ultimo giorno è Domenica
L'ultimo giorno ora è Sabato
Ora all'indice 3 abbiamo Giovedi
```

5.2.4 Ordinamento di una Tabella

A volte è necessario ordinare una tabella secondo un particolare ordine. La funzione *sort* permette di ordinare una tabella in ordine alfabetico. Ecco come funziona:

```
giorni = {"Lunedì", "Martedì", "Mercoledì", "Giovedì",
"Venerdì", "Sabato", "Domenica"}

-- Stampa l'elenco dei giorni senza ordinamento
for k,v in ipairs(giorni) do
  print(k..": ",v)
```

```
end

Risultato:
1: Lunedì
2: Martedì
3: Mercoledì
4: Giovedì
5: Venerdì
6: Sabato
7: Domenica

-- Ordina la tabella
table.sort(giorni)

-- Ed ora stampala
print("sorted table")
for k,v in ipairs(giorni) do
  print(k..": ",v)
end
Tabella ordinata
1: Domenica
2: Giovedi
3: Lunedi
4: Martedi
5: Mercoledi
6: Sabato
7: Venerdi
```

È possibile creare una tabella all'interno ad un'altra:

```
a={{}}
```

Ecco come si crea una matrice (un vettore di vettori) rettangolare in Lua:

```
mat = {}          -- crea la matrica
for i=1,N do
```

```
  mat[i] = {}      -- crea una nuova linea
  for j=1,M do
    mat[i][j] = 0.0
  end
end
```

Questo codice invece crea una matrice triangolare

```
mat = {}           -- crea la matrica
for i=1,N do
  mat[i] = {}      -- crea una nuova linea
  for j=1,i do
    mat[i][j] = 0.0
  end
end
```

Uno degli aspetti più interessanti della tabelle in Lua è la facilità con la quale possiamo creare vettori e matrici. Questo fatto rende Lua un linguaggio adatto allo sviluppo di piccole applicazioni scientifiche e tecniche molto utili ed in poco tempo.

■

PROSSIMO CAPITOLO

Funzioni ★

Capitolo 6

STRINGHE

La libreria ‘‘string’’ offre un insieme di funzioni molto versatili per la gestione delle stringhe. Queste funzioni sono esportabili come modulo chiamato appunto ‘‘string’’.

6.1 Funzioni basilari

Come vederemo, si tratta di funzioni semplici e di immediato utilizzo. Ecco quelle più comunemente usate:

- string.len(s) restituisce la lunghezza di una stringa. Questa funzione è simile a #s.
- string.rep(s,n) restituisce una stringa contenente la stringa s ripetuta n volte. Questa funzione è simile a s:rep(n).
- string.upper(s) restituisce la stringa s in lettere maiuscole. Questa funzione è equivalente a s:upper().
- string.lower(s) restituisce la stringa s in lettere minuscole. Questa funzione è equivalente a s:lower().

- string.sub(s,i,j) restituisce una stringa iniziando con il carattere di ordine i e finendo con il carattere di ordine j. Questa funzione è simile a s:sub(i,j)).
- string.sub(s,2,-2) restituisce la stringa s con il primo ed ultimo carattere rimossi. Questa funzione è simile a s:sub(2, -2).
- string.char(n1,n2...) converte i numeri n1, n2... nei corrispondenti carattere ASCII.
- string.byte(s,i) restituisce il valore in byte del carattere d'ordine i nella stringa s. Il secondo argomento i è facoltativo e quindi se viene emesso la funzione restituisce il primo carattere della stringa s.
- string.byte(s,i,j) restituisce valori multipli dei caratteri tra gli indici i e j.
- s:byte(1,-1) restituisce una tabella composta da tutti i valori dei caratteri contenuti nella stringa s.
- string.format è una funzione per la formattazione delle stringhe ed è molto simile alla funzione *printf* del linguaggio C, oppure a *format* dell'Object Pascal.

C'è qualcosa di molto importante da ricordare quando si tratta di lavorare con le stringhe in Lua. Queste rimangono immutabili. In altri termini, le funzioni citate, come string.upper e le altre, non modificano il valore originale della stringa ma restituiscono una nuova stringa modificata. Perciò se, per esempio, vogliamo trasformare la stringa "ciao mamma!" in "CIAO MAMMA!" dobbiamo scrivere qualcosa del genere:

```
s = "ciao mamma!"
s = s:upper()
Ora s = "CIAO MAMMA!".
```

NOTA BENE

Le stringhe in Lua sono immutabili. ♦

6.2 Esempi Pratici

Ecco una serie di brevi esempi che illustrino il funzionamento delle funzioni citate nella precedente sezione.

s=”Jilani KHALDI”
#s => 13

NOTA

Lo spazio vuoto è contato come un singolo carattere. ♣

string.rep(’Ciao! ’, 2) => Ciao! Ciao!

s:upper => JILANI KHALDI

s:lower => jilani khaldi

t=”(Ciao Mamma!)” t:sub(2, -2) => Ciao Mamma!

string.char(97,98,99) => abc

string.byte(”abc”) => 97

string.byte(”abc”, 1) => 97

string.byte(”abc”, 2) => 98

string.byte(”abc”, 3) => 99

string.byte(”abc”, -1) => 99

string.byte(”abc”, 1, 3) => 97 98 99

s:byte(1,-1) => 67 73 65 79 32 77 65 77 77 65 33

string.format(”pi = %.5f”, math.pi) => pi = 3.14159

string.format(”e = %.5f”, math.exp(1)) => e = 2.71828

A questo punto, vista l’importanza della gestione delle stringhe nella programmazione in Lua, il lettore è invitato a provare con altri esempi, magari con funzioni combinate.

6.3 Funzioni di Pattern-Matching

Il *pattern matching*, ossia la corrispondenza tra gli elementi di due parti, è un meccanismo estremamente potente ed è alla base della programmazione funzionale. Ecco un piccolo esempio per rendere chiaro il concetto. Considerando la seguente espressione:
x, y, z = 3.14, ”Che bella giornata!”, [1,2,3,4,5,2.718, ”cane”, ”gatto”]

Secondo il meccanismo del pattern matching abbiamo:

```
x = 3.14
y = "Che bella giornata!"
z = [1,2,3,4,5,2.718, "cane", "gatto"]
Cioè, x prende il primo valore che è un reale,
y prende il secondo valore che è una stringa,
mentre z prende il valore della lista.
```

Il pattern matching offre molta flessibilità per strutturare il codice rendendolo più leggibile, chiaro, efficiente e breve.

Per il pattern matching Lua non usa la libreria di espressioni regolari del linguaggio Perl oppure quella standard POSIX, ma ha implementato la sua propria libreria per ridurre le dimensioni dell'interprete.

NOTA

La libreria di espressioni regolari di POSIX ha più di 4000 linee di codice, mentre quella di Lua ha meno di 600. ♣

Tra le funzioni più potenti della libreria che gestisce le stringhe troviamo *find* (trova), *match* (corrispondenza), *gsub* (sostituzione globale) e *gmatch* (corrispondenza globale). Tutte queste funzioni sono bastate sull'uso dei pattern.

6.3.1 string.find

La funzione *string.find* cerca un pattern (corrispondenza) all'interno di una stringa. Ecco alcuni esempi:

```
s = "Paolo Rossi, Mondiali 1980"
i, j = string.find(s, "Rossi")
print(i, j) => 7 11

print(string.sub(s, 7, 11)) => Rossi

Trova la virgola:
print(string.find(s, ",")) => 12 12

i, j = string.find(s, "Pirlo") => nil
```

La funzione *string.find* ha un terzo parametro opzionale che è l'indice del carattere da dove iniziare la ricerca. Questo parametro diventa molto utile nel caso in cui cerchiamo un termine presente più volte nel nostro testo.

Il seguente esempio conta il numero delle parole in una frase separate da una virgola:

```
s = "arance, banane, mele, kiwi, kaki, nespole, fichi"
k = 1 -- inizia dal primo carattere
count = 1 -- parte con la prima frutta
while true do
k = string.find(s, ",", k+1) -- trova la prossima
if k == nil
then break
end
count = count +1
end
print(count)
```

6.3.2 string.match

La funzione *string.match* è simile a *string.find* ma restituisce il termine corrispondente al pattern richiesto. Ecco un esempio:

print(string.match("Ciao mamma!", "Ciao")) => Ciao

Ecco un altro esempio più complesso:

```
data = "Il giorno del D-day è stato il 06/06/1944"
dd = string.match(data, "%d+/%d+/%d+")
print(dd) => 06/06/1944
```

6.3.3 string.gsub

La funzione *string.gsub* sostituisce una parte di una stringa con un'altra. Ecco un esempio:

```
s = string.gsub("Paolo Rossi", "Rossi", "Conte")
print(s) => Paolo Conte

s = string.gsub("3/3/1954", "3", "6")
print(s) => 6/6/1954

s = string.gsub("3/3/1954", "3", "14", 1)
print(s) => 14/3/1954
```

Un quarto parametro opzionale potrebbe essere aggiunto alla funzione per limitare il numero delle sostituzioni.

La funzione string.gsub inoltre restituisce come secondo risultato che rappresenta il numero delle sostituzioni eseguite. Ecco un esempio che restituisce il numero delle volte che è stata ripetuta la parola "Rossi":

```
s = "Paolo Rossi, Aldo Rossi, Alberto Rossi, Gino Rossi"
n = select(2, string.gsub(s, "Rossi", "Rossi"))
print(n) => 4
```

6.3.4 string.gmatch

la funzione *string.gmatch* restituisce una funzione che itera alla ricerca delle ricorrenze di un termine in una stringa. L'esempio seguente restituisce il tutte le parole contenute in una stringa:

```
s = "Paolo Rossi, Aldo Rossi, Alberto Rossi, Gino Rossi"
parole = {}
for k in string.gmatch(s, "%a+") do
```

```
  parole[#parole + 1] = k
end
for k = 1,#parole do
  print(parole[k])
end
```

Come appare nella tabella 6.1, il pattern "%a+" corrisponde ad una sequenza di uno o più caratteri alfabetici.

6.4 Pattern

Per rendere i pattern più utili e flessibili è raccomandabile usarli in combinazione con dei particolari caratteri. Per esempio "%d" significa un carattere numerico, cioè un carattere compreso tra 0 e 9. Ecco un esempio pratico:

```
s = "Il D-Day è stato il 06/06/1944 in Normadia."
data =  "%d%d/%d%d/%d%d%d%d"
t,u = string.find(s, data)
print(string.sub(s, t, u))
```

L'esempio precedente cerca la data del D-Day nel formato "gg/mm/aaaa" all'interno della stringa s.

La tabella 6.1 illustra i caratteri particolari da combinare con i pattern di ricerca.

Tabella 6.1: Simboli di pattern

Simbolo	Significato
.	Tutti i caratteri
%a	caratteri alfabetici
%c	Caratteri di controllo
%d	Numeri digitali
%g	Caratteri stampabili
%l	Lettere minuscole
%p	Caratteri di puntualizzazione
%s	Spazi vuoti
%u	Lettere maiuscole
%w	Caratteri alfanumerici
%x	Numeri in esadecimali

NOTA BENE

Sostituendo il carattere minuscolo con quello maiuscolo nella tabella 6.1 otteniamo l'insieme dei caratteri complementari.

Se, per esempio, vogliamo sostituire i non caratteri con il simbolo "-" all'interno di una stringa, allora usiamo "%A".

```
s = "Paolo Rossi, un ragazzo come noi"
print(string.gsub(s, "%A", "_"))
=> Paolo_Rossi__un_ragazzo_come_noi 6
```

NOTA BENE

*I seguenti caratteri "( ) [ ] . ? * + - % $, detti caratteri magici, hanno dei significati particolari nell'uso dei pattern.*

Tabella 6.2: Simboli di pattern

Simbolo	Significato
+	1 o più ripetizioni
*	0 o più ripetizioni
-	simile a "*" ma segue la sequenza più breve
?	0 oppure 1 occorrenza

Il modificatore "+" corrisponde ad uno o più caratteri simili a quello da modificare. Ecco un esempio:

```
s="Maria, Anna, Paola"
print(string.gsub(s, "%a+", "Gabriella"))
=> Gabriella, Gabriella, Gabriella 3
```

Il pattern "%d+" corrisponde ad un numero intero. Ecco un esempio:

```
s="L'anno del cavallo è il 1954 secondo l'oroscopo cinese."
print(string.match(s, "%d+")) => 1954
```

Il modificatore "*" è simile a quello "+", l'unica differenza è che accetti zero occorrenza; mentre il modificatore "-" è simile a "*" e l'unica differenza è che segua la sequenza di caratteri più breve. Il modificatore "?" corrisponde ad un carattere opzionale. Quando il pattern inizia "^" allora indica il primo carattere della stringa, mentre "$" indica l'ultimo.

C'è un altro carattere molto usato nei pattern che è "%b", "%b", oppure "%b<>". Questo carattere serve nelle stringhe bilanciate, cioè, tipo quelle che iniziano e finiscono con una parentesi.

Ora vediamo alcuni esempi che illustrino l'uso dei pattern.

Il seguente esempio ci permette di trovare un numero all'interno di una stringa:

```
s="Nei mondiali del 1982 Rossi segnò 3 gol al Brasile"
t=string.match(s, ?[+-]?%d+?)
print(t) => 1982
```

Il seguente esempio scandisce la stringa e sostituisce la parola ”cagnolino” con ”gattino”.

```
s = "Un (cagnolino) dentro la casetta."
print(string.gsub(s, "%b()", "gattino"))
=> Un gattino dentro la casetta. 1
```

Il seguente esempio sostituisce la parola ”due” con la parola ”tre” usando il pattern frontiera ”%f[insieme di caratteri]”.

```
s = "abbiamo due cani e due gatti."
u,v = s:gsub("%f[%w]due%f[%W]", "tre")
print(u)=> abbiamo tre cani e tre gatti.
print(v) => 2
```

6.5 Cattura

La cattura è un’azione che permette ad un pattern d’estrarre una parte di una stringa secondo un preciso criterio usando la funzione *string.match*.

Il seguente esempio cattura la targa di una macchina.

```
s = "Targa = AB123JK"
targa, valore = string.match(s, "(%w+)%s*=%s*(%w+)")
print(targa, valore) => Targa AB123JK
```

Questo altro esempio cattura e scandisce una data all’interno di una stringa.

```
dataN = "Data di nascita 11/12/2004"
g, m, a = string.match(dataN, "(%d+)/(%d+)/(%d+)")
print(d, m, y) => 11 12 2004
```

A questo punto il lettore è pregato di esercitarsi di esplorare questi meccanismi di pattern che sono utilissimi per la gestione e manipolazione delle stringhe che è considerato uno dei punti forti di Lua.

NOTA BENE

Spesso ci sono diverse soluzioni di pattern per raggiungere lo stesso risultato, il lettore deve sempre misurare le prestazioni di ciascuna soluzione per la ricerca di quella più efficiente.

PROSSIMO CAPITOLO

Input Output

Capitolo 7

INPUT OUTPUT

La libreria Input Output (I/O) di Lua offre una serie di funzioni per la lettura e scrittura dei file sul dispositivo di default. Ci sono due modelli per la manipolazione dei file. Il primo è molto semplice e opera due file, cioè uno per la lettura dei dati e l'altro per la scrittura. Il secondo modello è invece più completo e permette di operare su più file contemporaneamente.

7.1 Modello Semplice di I/O

Il primo file di input sul quale si opera in questo caso è associato al modello standard di input, cioè *stdin* mentre il secondo file per l'output è associato al modello standard di output, ossia *stdout*.

La funzione *io.input(NomeFile)* apri il file in modalità di lettura e diventa il file di riferimento sul quale operare. La funzione *io.output(NomeFile)* si comporta allo stesso modo della precedente ma per la scrittura. L'errore in lettura oppure in scrittura è automaticamente segnalato. Nel caso in cui l'utente desidera gestire gli errori deve utilizzare il modello completo.

NOTA BENE

Nel caso in cui l'utente desidera gestire gli errori deve utilizzare il modello completo.

7.1.1 io.write

La funzione *io.write* legge un numero arbitrario di dati in formato stringa e li scrive nell'attuale file di output. I numero sono convertiti in stringhe e per aver maggior controllo sull'output si utilizza la funzione *string.format*.

L'esempio seguente illustra come utilizzare le funzioni *io.write* e *string.format*.

```
io.write("sin (30°) = ", math.sin(math.pi/6), "\n")
=> sin (30°) = 0.5
> io.write("sin (60°) = ", math.sin(math.pi/3), "\n")
sin (60°) = 0.86602540378444

>  io.write(string.format("sin (60°) = %.5f\n",
math.sin(math.pi/3)))
sin (60°) = 0.86603
```

NOTA BENE

Per un miglior controllo su output di tipo numerico usare la funzione string.format.

Per concatenare più elementi si possono usare i seguenti metodi meccanismo:

io.write(a..b..c) oppure io.write(a,b,c). Il secondo metodo usa meno

risorse del primo.

Esempio:

```
io.write("ciao".." Lua!\n")
=> Ciao Lua!
io.write("ciao", " Lua!\n")
=> Ciao Lua!
```

7.1.2 print

L'utilizzo della funziona *print* è consigliato per un output veloce e per il debug, cioè la ricerca di errori nel programma.

Esempio:

```
print("ciao".." Lua!\n")
=> Ciao Lua!
print("ciao", " Lua!\n")
=> Ciao   Lua!
```

7.1.3 io.read

La funzione *io.read* legge le stringhe contenute in un file aperto. Questa funzione usa i seguenti controlli per la lettura dei dati:

Tabella 7.1: Simboli di pattern

Simbolo	**Significato**
*a	legge l'intero file
%l	legge la linea successiva senza carattere di fine linea
*L	legge la linea successiva con carattere di fine linea
*n	legge un numero
num	legge num caratteri di una stringa

Una chiamata alla funzione ‘‘io.read(”*a”)’’ legge l’intero file corrente che sarà poi trasformato in un’unica stringa. Grazie a questo meccanismo, tra l’altro molto efficiente, la gestione dei file in formato testo diventa molto semplice e flessibile in Lua. Ecco un semplice esempio:

```
t = io.read("*a") -- legge l'intero file
t = string.gsub(t, ...) -- trasformalo in una string
io.write(t) -- scrivi il file
```

La funzione ‘‘io.read(”*l”)’’ restituisce la linea seguente del file di input, ma segna il carattere di fine linea. La funzione ‘‘io.read(”*L”)’’ è simile al precedente, in più conserva il carattere di fine linea se è presente nel file. A fine file, le due funzioni restituis ‘‘nil’’. Il pattern‘‘*l? è quello di default.

7.2 Modello Completo di I/O

Per un maggior controllo sul I/O, è più conveniente utilizzare le funzioni avanzate offerte dal modello completo. Questo modello si basa concettualmente sul riferimento ad un file generico, cioè qualcosa di molto simile al puntatore ad un file del linguaggio C (FILE*). In altri termini, un file aperto con una posizione sul suo primo elemento.

Per aprire un file utilizziamo la funzione ‘‘io.open’’ che prende come argomenti il nome del file compreso di percorso seguito dal carattere ‘r’ per la lettura oppure ‘w’ per la scrittura. C’è un ulteriore carattere opzionale che è ‘a’ per aggiungere qualcosa al file. Queste opzioni sono applicabili a file di testo. Quando si tratta invece di un file in formato binario allora si aggiunge il carattere ‘b’.

NOTA BENE

In caso di errore la funzione "io.open" restituisce 'nil' ed un messaggio di errore con il relativo codice.

semplice esempio:

```
print(io.open("myfile.txt", "r"))
-- apri il file ?myfile.txt?
> nil myfile.txt: No such file or directory 2
-- file o directrory inesistente

print(io.open("/root/myfile.txt", "w"))
-- crea il file ?myfile.txt?
> nil /root/myfile.txt: Permission denied 13
-- accesso negato
```

Il valore del codice d'errore e la sua interpretazione dipendono dal sistema operativo in questione.

Dopo l'apertura del file, è possibile leggere i contenuti di un file oppure crearne uno nuovo. Si possono usare le funzioni "leggi/scrivi" (read/write) in modo simile a quello precedente. Per aprire un file e leggerlo per intero, per esempio, possiamo farlo in questo modo:

```
local f = assert(io.open(NomeFile, "r"))
local t = f:read("*a")
f:close()
```

La libreria standard di I/O di Lua offre tre meccanismi per il controllo del flusso dei dati contenuti in un file: io.stdin, io.stdout, and io.stderr. Il primo per l'input, il secondo per l'output, mentre il terzo per gli errori. In caso di errore, per esempio, si può inviare un messaggio di errore tipo:

```
io.stderr:write(messaggio)
```

È comunque possibile combinare il modello completo con quello semplice, come per esempio chiamare la funzione 'io.input()' senza argomenti. Possiamo scrivere codice del genere:

```
local temp = io.input() -- salva il file attuale
io.input("NuovoFile") -- crea un nuovo file vuoto
<esegui alcune operazioni sul nuovo file>
io.input():close() -- chiudilo
io.input(temp) -- ritorno al file precedente
```

Invece di usare 'io.read', posiamo usare 'io.lines' per leggere i dati da un file come abbiamo già visto in precedenza. Il primo argomento può essere il nome oppure un riferimento (handle) di un file.

7.3 Altre Operazioni su File

La funzione **tmpfile** restituisce il riferimento al file temporaneo, aperto per la lettura/scrittura. Normalmente questo file viene eliminato a fine esecuzione programma. La funzione **flush** esegue in un'unica operazione la scrittura di tutti i dati pendenti. Tale funzione viene chiamata in questo modo:

```
io.flush() -- scrivi tutti i dati pendenti nel file attuale
f:flush() -- scrivi tutti i dati nel file f
```

Il metodo **seek** può stabilire, oppure ottenere, la posizione all'interno di un file e viene usato nel seguente modo:

```
f:seek(DaDove,offset)
```

DaDove può avere uno dei seguenti valori: 'set' per indicare l'inizio del file, 'cur' per indicare l'attuale posizione ed in fine 'end' per indicare la fine del file. Indipendentemente dal valore di DaDove, la funzione di ritorno restituisce la nuova posizione nel file misurata in byte a partire dall'inizio del file.

NOTA BENE

*Il valore di default di **set** e **cur** è zero.*

L'esempio seguente ci permette di trovare le dimensioni del file senza modificare la sua attuale posizione. function fDimensioni (file) local attuale = file:seek() – ottiene l'attuale posizione local dimensioni = file:seek("end") – ottiene le dimensioni del file file:seek("set", attuale) – torna alla posizione iniziale return dimensioni end In caso di errori, la funzione restituisce **nil**.

PROSSIMO CAPITOLO

Funzioni del Sistema Operativo

Capitolo 8

FUNZIONI DEL SISTEMA OPERATIVO

Tutti i sistemi operativi offrono molte funzioni per la gestione del tempo e delle date. Essendo scritto in ANSI C, Lua sfrutta solo alcune di queste funzioni per ragioni di portabilità ignorando tutte quelle funzioni strettamente legate al sistema operativo. Comunque Lua offre il necessario per la gestire e la manipolazione del tempo e delle date.

8.1 Data e Ora

Ci sono due funzioni *time* (tempo) e *date* (data) per la gestione del tempo e delle date in Lua.

8.1.1 date

Questa funzione priva di argomenti restituisce la data e l'ora attuali fornite dal sistema operativo.

Eseguendo questa funzione con il sistema operativo Windows otteniamo:

```
> os.date()
=> 06/25/17 19:40:51
```

8.1.2 time

Questa funzione priva di argomenti restituisce il numero dei secondi trascorsi a partire da una certa data che dipende dal sistema operativo.

Eseguendo questa funzione otteniamo:

```
> os.time()
=> 1498427488
```

La seguente tabella illustra tutte le possibilità per una completa gestione della data e dell'ora.

Tabella 8.1: Simboli di pattern

Simbolo	Significato
%a	nome giorno abbreviato (esempio: Lun)
%A	nome giorno completo (es. Lunedì)
%b	nome mese abbreviato (es. Ago)
%B	nome mese completo (es. Marzo)
%c	data ed ora (es. 12/31/2018 23:59:59)
%d	giorno del mese [01..31]
%H	ora, considerando un giorno di 24 ore [00..23]
%I	ora, considerando un giorno di 12 ore [01..12]
%j	giorno dell'anno [001..366]
%M	minuti [00..59]
%m	mese [01..12]
%p	am (antimeridiane) o pm (post meridiane)
%S	secondi [00..60]
%w	giorno della settimana [0..6 = Domenica..Sabato]
%x	data (es. 03/27/92)
%X	ora (es. 22:40:15)
%y	anno con due cifre (es. 18) [00..99]
%Y	anno con tutte le cifre (es. 2018)
%%	il carattere '%'

8.2 Timing

Il timing è il lasso tempo tra due eventi e serve spesso per monitorare le prestazioni di un programma e scoprire i punti di minor efficienza di esecuzione. Ecco un semplice esempio:

```
-- fissa il momento
local inizio = os.clock()
local somma = 0.0
-- calcola somma un milione di volte pi greco
for i = 1, 1000000 do somma = somma + 3.14 end
-- stampa il tempo impiegato
```

```
print(string.format("Tempo impiegato: %.3f\n", os.clock() - iniz
```

Calcolare il timing di una funzione, soprattutto quando si tratta di operazioni impegnative di calcolo numerico oppure di lettura/scrittura di dati, ci aiuta spesso a scoprire i colli di bottiglia nel programma e rendere il codice più veloce ed efficiente.

8.3 Altre Chiamate al Sistema

Ci sono diverse utili chiamate al sistema operativo che possono essere eseguite direttamente da Lua.

8.3.1 os.getenv

La chiamata 'os.getenv' restituisce il percorso dell'ambiente di lavoro.

```
print(os.getenv("HOME")) --> C:\Users\jilani
```

8.3.2 os.execute

La chiamata 'os.execute' è estremamente potente perché ci permette di eseguire comandi del sistema operativo direttamente da codice Lua. Il seguente codice, per esempio, crea una nuova directory. Questo comando, come tutti gli altri, dipende dal sistema operativo utilizzato.

```
-- Windows
function createDir (NomeDir)
os.execute("mkdir " .. NomeDir)
end

-- Unix
function createDir (NomeDir)
```

```
os.execute("md " .. NomeDir)
end
```

ATTENZIONE

Prima di eseguire comandi del sistema operativo da codice Lua bisogna essere consapevoli dell'esito di tali comandi perché nella maggior parte dei casi, soprattutto quando si tratta di Unix, le operazioni sono irreversibili.

Come abbiamo avuto modo di vedere, Lua possiede un insieme di funzioni semplici, intuitive e molto flessibili per la gestione delle date e del tempo nonché la possibilità di eseguire direttamente comandi del sistema operativo. A questo punto il lettore è chiamate a esercitarsi per prendere confidenza con queste funzioni.

■

PROSSIMO CAPITOLO

Gestione degli Errori

Capitolo 9

GESTIONE DEGLI ERRORI

Scrivere applicazioni di una certa entità senza errori è quasi impossibili, perciò tutto quello che possiamo fare nello sviluppo del software è limitare il numero degli errori. La strategia di raggiungere tale obiettivo è semplice ed ormai collaudata. In effetti, bastano tre modi per farlo. La prima ed è la più importante è scrivere codice leggibile, modulare e testare ogni singola funzione. La seconda è monitorare il funzionamento del programma in alcuni punti particolari. La terza invece è l'utilizzo intensivo del programma per scoprire comportamenti anomali e colli di bottiglia.

Lua offre semplici ed efficienti meccanismi per sgamare e capire il tipo di errori indicando esattamente la sua posizione all'interno del codice.

9.1 Tipi di Errori

Molti errori vengono individuati e segnalati durante la compilazione del codice sorgente in bytecode eseguibile. Di solito questi errori sono dovuti ad errori di battitura e quindi facilmente correggibili. In questo caso si tratta spesso di scrivere male il nome di un comando Lua oppure quello di una variabile.

Gli errori più subdoli sono quelli contenuti in un codice sintatticamente corretto, quindi non viene mai segnalato dal compilatore, ma che producono un risultato non corretto. Questi errori si manifestano durante l'esecuzione del programma e per trovarli bisogna eseguire il debug, ossia, eseguire il programma passo-passo ed osservare il risultato di ogni singola istruzione. A volte, però, basta riesaminare il codice in alcuni punti critici per trovare l'errore e ricompilare il codice.

NOTA

Gli utenti finali dei programmi sono i miglior tester dei programmi e sono spesso in grado di individuare errori e mal funzionamenti. ♣

9.1.1 Errori di Sintassi

Sono gli errori più frequenti e si commettono durante la codifica del programma. La casistica è molto variegata e spazia da comandi e variabili scritte male, ad operazioni matematiche scorrette fino ad arrivare a combinazioni scorrette di comandi ed istruzioni. Comunque in tutti i casi un errore del genere è automaticamente individuato e segnalato in fase di compilazione del codice.

Ecco alcuni esempi:

```
a=1*+2 -- operazione illecita
st= 'Ciao ' + 'mondo!' -- errore di sintassi
for i=1..100 do -- errore di sintassi
a=a+1 -- a non è definita
```

Errori del genere si possono commettere praticamente in qualsiasi momento della codifica per differenti ragioni, ma per fortuna sono molto facili da individuare grazie al compilatore.

9.1.2 Errori Subdoli

Sono gli errori di codifica, di corretta sintassi ma che non producono i risultati attesi. Sono quasi sempre dovuti ad errori nell'interpretazione di un algoritmo, ad una esecuzione di un comando Lua in modo improprio, oppure di un'operazione sbagliata ma sintatticamente corretta.

A questo punto serve un esempio concreto per capire come si presentano questi errori ed in che modo vengono sgamati. Il codice seguente risolve un'equazione di secondo grado ma in modo sbagliato a causa di un errore nella formula risolutiva.

```
function equazione2grado(a,b,c)
  local delta = b*b-4*a*c
    if delta < 0 then
   return 0, 0, 0
    else if delta == 0.0 then
     return 1, -b/(2*a), -b/(2*a)
    else
      x1 = (-b-math.sqrt(delta))/2*a
      x2 = (-b+math.sqrt(delta))/2*a
      return 2, x1, x2
    end
    end
end
```

Ora chiamando questa funzione con questo codice:

```
sol, x1, x2 = equazione2grado(5,-8,3)
print(sol)
print('x1 = '..x1, ', x2 = '..x2)
```

il compilatore esegue il codice senza errori e ci fornisce due soluzioni $x_1 = 15.0$ e $x_2 = 25.0$ che sono sbagliate. Quelle corrette sono $x_1 = 1.0$ e $x_2 = 0.6$. Quindi l'errore c'è e lo dobbiamo trovare.

Questo esempio è molto facile e perciò è quasi immediato trovare l'errore che sta nell'espressione dell'equazione risolutiva nel caso in cui il discriminante dell'equazione è maggiore di zero. Quindi basta correggere le due linee (8 e 9) incriminate del codice, dopodiché avremo il risultato corretto codice corretto:

```
x1 = (-b-math.sqrt(delta))/(2*a)
x2 = (-b+math.sqrt(delta))/(2*a)
```

In molti casi, e vista la vastità delle casistiche e possibilità di commettere errori di ogni genere, soprattutto nella codifica di algoritmi complessi, trovare tutti gli errori potrebbe diventare un incubo per il programmatore. Molto spesso gli errori vengono segnalati dagli utenti anche dopo un lungo tempo di utilizzo del programma. In questo caso gli errori si manifestano solo in determinate circostanze e combinazioni di particolari condizioni che a volte rende estremamente difficile trovarli anche da parte di esperti programmatori.

NOTA BENE

Un software di una certa complessità senza errori è possibile solo in teoria. ♦

Perciò, quando si scrive del software complesso, chi lo sviluppo non ha nessuna responsabilità sulla presenza degli errori nella codifica. Diversamente nessun avrebbe scritto del software.

9.2 Limitare gli Errori

Visto che matematicamente è quasi impossibile scrivere software di una certa complessità privo di errori, però è possibile ridurre drasticamente il numero degli errori e renderli facilmente individuabili grazie all'utilizzo di alcuni stratagemmi che andremo ad illustrare.

9.2.1 Divisione in Moduli

Lua permette di dividere un programma in moduli (vedere il prossimo capitolo), in questo modo possiamo sviluppare ogni modulo come parte autonoma e testarlo separatamente fino ad assicurarci della sua validità.

9.2.2 Piccole Funzioni

Un'altra valida strategia è quella di scrivere piccole funzioni dove ogni funzione prende come input il minimo di variabili e restituire pochissimi valori di ritorno. Quando una funzione diventa lunga o complessa è meglio suddividerla in più funzioni.

Quando è il caso, prima di inviare degli argomenti ad una funzione è meglio controllare la loro validità. Se per esempio dobbiamo scrivere una funzione che risolve l'equazione di secondo grado, cioè $ax^2 + bx + c = 0$, dobbiamo prima controllare che il coefficiente a sia diverso da zero prima di chiamare la funzione risolutiva per evitare la divisione per zero. Questo errore potrebbe bloccare il programma o peggio ancora fornire dei risultati inattendibili.

9.2.3 Leggibilità

Scrivere codice leggibile e commentarlo nei punti particolari oltre a limitare il numero degli errori e renderli facilmente individuabili

rende soprattutto il codice comprensibile e facilmente estendibile in futuro senza grossi problemi. Un codice offuscato e poco commentato diventa incomprensibile dopo un po' di tempo anche per chi l'abbia scritto.

Bisogna sempre cercare di scrivere codice leggibile e commentarlo a meno che non si tratta di piccole funzioni auto-commentate oppure di breve codice che serve solo per sperimentare.

9.3 Debugger

Un debugger è un programma autonomo in grado di individuare gli errori all'interno del codice sorgente di un programma. In altri termini un debugger permette di eseguire una singola funzione o anche un'istruzione alla volta del programma visualizzando il valore delle variabili desiderate.

Personalmente trovo che la funzione 'print(..)' ben piazzata all'interno del codice con i giusti argomenti sia facile, pratica e sufficiente per sgamare gli errori più subdoli non solo con Lua, ma tutti i linguaggi che ho conosciuto finora. Perciò vi suggerisco di usare appena vi rendete conto che c'è qualcosa che non quadra nel vostro codice.

■

PROSSIMO CAPITOLO

Funzioni ★

Capitolo 10

MODULI

Uno dei modi migliori per strutturare un programma di una certa complessità è di suddividerlo in sub-programmi con interfacce ben definite ed implementazioni protette.

10.1 Interfacce ed Implementazioni

Una interfaccia è la parte di un sub-programma che è resa visibili per essere chiamata direttamente dallo stesso sub-programma. L'implementazione, invece, è la la parte implementativa dell'interfaccia. Quando una interfaccia è ben definita e la sua implementazione non dipende da variabili modificabili da altre parti del programma, allora diventa riutilizzabile. Questa caratteristica viene spesso chiamata modularizzazione oppure incapsulamento del codice.

Il fatto che una funzione Lua possa disporre di variabili locali utilizzabili solamente all'interno della stessa funzione è estendibile anche a file contenenti codice Lua dove un singolo file si comporta in modo simile ad una funzione Lua con gli argomenti come input ed i

risultati come output. Così un file file ha una propria interfaccia e la corrispondente implementazione.

NOTA BENE

Un modulo Lua è un file con una propria interfaccia visibile dall'esterno ma con una implementazione invisibile.

10.2 La Funzione Require

La funzione **require** carica un modulo in memoria e lo mette a disposizione del programma Lua chiamante. I moduli sono disponibili a run-time, ossia vengono eseguiti solo nel momento in cui sono chiamati, quindi sono diversi dalle direttive che fanno parte del programma principale. L'importante è che il modulo sia disponibile al momento di esecuzione del programma.

Ecco un esempio pratico per capire come funzionano i moduli. Scriviamo il codice seguente e lo salviamo in un file col nome 'modulo.lua'.

```
local modulo = {}

function modulo.MostraOggetto(Valore, Chiave)
print(Valore, Chiave)
end

return modulo
```

Ora creiamo un altro file contenente il seguente codice e lo salviamo col nome 'expmod.lua'.

```
m = require("modulo")
```

```
m.MostraOggetto("My table", {A = 1})
```

Ora eseguiamo questo esempio, cioè:

```
lua expmod.lua
```

Ed ecco il risultato:

```
My table table: 000000000054af50
```

NOTA

Ci sono altri modi di creare i moduli in Lua, ma quello appena descritto sembra essere il più semplice e leggibile. ♣

L'utilizzo dei moduli diventa necessario appena il nostro programma raggiunge una certa lunghezza e complessità. Suddividere un lungo programma in programmi più brevi e specializzati a svolgere un picco gruppo di funzioni è estremamente conveniente per ridurre gli errori ed aumentare la leggibilità e l'efficienza dei programmi.

10.3 Un Esempio Pratico

Questo modulo contiene una serie di semplici funzioni per il calcolo di numeri complessi.

```
local ComplexNum = {}

function ComplexNum.new(r, i)
  return {r=r, i=i}
end
```

```
-- il numero immaginario puro 'i'
ComplexNum.i = ComplexNum.new(0, 1)

function ComplexNum.add(c1, c2)
  return ComplexNum.new(c1.r + c2.r, c1.i + c2.i)
end

function ComplexNum.sub (c1, c2)
  return ComplexNum.new(c1.r - c2.r, c1.i - c2.i)
end

function ComplexNum.mul (c1, c2)
  return ComplexNum.new(c1.r*c2.r - c1.i*c2.i,
  c1.r*c2.i + c1.i*c2.r)
end

local function inv (c)
  local n = c.r^2 + c.i^2
  return ComplexNum.new(c.r/n, -c.i/n)
end

function ComplexNum.div (c1, c2)
  return ComplexNum.mul(c1, inv(c2))
end

function ComplexNum.tostring (c)
  return c.r .. "," .. c.i
end

return ComplexNum
```

Ed ecco un semplice esempio su come utilizzare questo modulo.

```
local cn = require "ComplexNum"
```

```
print(cn.tostring(cn.add(cn.new(1,3), cn.i)))
print(cn.tostring(cn.add(cn.new(1,3), cn.new(5,7))))
print(cn.tostring(cn.sub(cn.new(1,3), cn.new(5,7))))
print(cn.tostring(cn.mul(cn.new(1,3), cn.new(5,7))))
```

Il risultato sarà:

```
1,4 --(1 + 4i)
6,10 -- (6 + 10i)
-4,-4 -- (-4 - 4i)
-16,22 -- (-16 + 22i)
```

Questo semplice modulo illustra anche l'utilità di Lua per scrivere applicazioni scientifiche e tecniche di modeste entità.

■

PROSSIMO CAPITOLO

Funzioni ★

Capitolo 11

PROGRAMMAZIONE SCIENTIFICA

Lua sembra di aver tutte le carte in regola per scrivere piccole applicazioni scientifiche e tecniche. Grazie alla sua semplicità, al paradigma di programmazione imperativo e funzionale nonché l'uso dei moduli, Lua è molto indicato per scrivere software scientifico e tecnico non troppo impegnativo.

Personalmente uso Lua come linguaggio per scrivere prototipi di programmi e soprattutto pezzi di codice per la verifica di algoritmi di calcolo.

In questa edizione ci limitiamo solo alla risoluzione degli esercizi proposti nel capitolo 4 (FUNZIONI), cioè:

- Risolvere un'equazione di secondo grado.
- Trovare le intersezioni tra una retta ed una parabola.
- Trovare il MCD (Massimo Comune Divisore) ed il mcm (minimo comune multiplo) di due numeri.

11.1 Equazione di II Grado

Questo esempio di codice Lua risolve l'equazione di II grado applicando il noto algoritmo cercando di scrivere nel modo più semplice e didattico possibile.

Data l'equazione di secondo grado $ax^2 + bx + c = 0$, l'algoritmo di calcolo per risolverla è il seguente:

- si calcola il suo discriminante ($\Delta = b^2 - 4ac$)
- se $\Delta > 0$ allora abbiamo due soluzioni reali e distinte ($x = \frac{-b \pm \sqrt{\Delta}}{2a}$)
- se $\Delta = 0$ abbiamo una sola soluzione reale doppia ($x = \frac{-b}{2a}$)
- se se $\Delta < 0$ non ci sono soluzioni reali.

Ora, ecco il codice

```
function equazione2grado(a,b,c)
  print("a="..a, ", b="..b, ", c="..c)
  if a == 0.0 then
    return -1, 0, 0
  end
  local delta = b*b-4*a*c
  if delta < 0 then
    return 0, 0, 0
  else if delta == 0.0 then
    return 1, -b/(2*a), -b/(2*a)
  else
    x1 = (-b-math.sqrt(delta))/(2*a)
    x2 = (-b+math.sqrt(delta))/(2*a)
    return 2, x1, x2
  end
end
end
```

```
function StampaRisultati(num, x1, x2)
  if num < 0 then
    print("Errore! Il coeff.a deve essere divero da 0.")
  elseif num == 0 then
    print("Non ci sono sluzioni reali.")
  elseif num == 1 then
    print("Una sola soluzione reale doppia: "..x1)
  else
    print("Due soluzioni reali: x1 = "..x1, " x2 = "..x2)
  end
end
```

Ora possiamo chiamare le due funzioni in questo modo:

```
a=0; b=3; c=2;
sol, x1, x2 = equazione2grado(a,b,c)
StampaRisultati(sol,x1,x2)
print("----------")
a=1; b=1; c=2;
sol, x1, x2 = equazione2grado(a,b,c)
StampaRisultati(sol,x1,x2)
print("----------")
a=1; b=2; c=1;
sol, x1, x2 = equazione2grado(a,b,c)
StampaRisultati(sol,x1,x2)
print("----------")
a=1; b=3; c=2;
sol, x1, x2 = equazione2grado(a,b,c)
StampaRisultati(sol,x1,x2)
```

Il lettore può modificare a piacimento il codice ed osservare come cambia il suo funzionamento.

11.2 Intersezione Parabola con Retta

Data l'equazione generica di una parabola $ax^2 + bx + c = 0$, trovare i punti d'intersezione con una retta generica di equazione $y = mx + q$ al variare dei coefficienti delle due equazioni.

Per trovare i punti di intersezione tra la parabola e la retta basta risolvere l'equazione di secondo grado $ax^2 + bx + c = mx + q$ ossia $ax^2 + (b - m)x + c - q = 0$. Ecco il codice risolutivo:

```
function ParabolaRetta(a,b,c,m,q)
  print("a="..a, ", b="..b, ", c="..c.., m="..m.., q="..q)
  local a1, b1, c1
  local num, x1, x2, y1, y2
  a1=a; b1=b-m; c1=c-q
  num,x1,x2=equazione2grado(a1,b1,c1)
  if num == 0 then
    print("La retta è esterna alla parabola.")
  elseif num == 1 then
    print("La retta è tangente alla parabola")
    y1=m*x1+q
    print("T("..x1..","..y1..")")
  else
    print("La retta è secante la parabola")
    y1=m*x1+q
    y2=m*x2+q
    print("P1("..x1..","..y1.."), P2("..x2..","..y2..")")
  end
end
```

Se, per esempio, vogliamo trovare i punti d'intersezione tra la parabola $y = x^2 + 4x + 5$ e la retta $y = x + 3$, dobbiamo risolvere l'equazione di secondo grado $x^2 + 4x + 5 = x + 3$, ossia $x^2 + 3x + 2 = 0$.

Ecco come avvien la chiamata alla funzione *PrabolaRetta*:

```
a=1; b=4; c=5; m=1; q=3
ParabolaRetta(a,b,c,m,q)
Che ci fornirà come soluzioni:
P1(-2.0,1.0), P2(-1.0,2.0)
```

11.3 MCD e mcm

Ci sono diversi algoritmi per trovare Il **MCM** (Massimo Comune Divisore) tra due numeri interi, ma il più semplice da implementare è quello delle divisione successive. Cioè:

1. di divide il numero maggiore per il minore e quindi abbiamo un quoziente ed un resto;

2. se il resto è uguale a zero allora il **MCD** è il divisore;

3. altrimenti si continua con la divisione assumendo il divisore come dividendo ed il resto come divisore e si continua allo stesso modo fino a trovare un resto uguale a zero. Quindi il **MCD** è il divisore.

Ecco un esempio numerico. Supponiamo di trovare il **MCD** tra 27 e 12. Abbiamo $\frac{27}{12} = 2$ con il resto di 3. Quindi D=27, d=12, R=3. Dopo questa operazione visto che il resto (R) non è zero avremo D=12 e d=3 e procediamo con la divisione. Ora abbiamo $\frac{12}{3} = 4$ con resto 0. Quindi MCD=d, cioè 3.

Ecco invece il codice Lua:

```
function MCD(n1,n2)
local dividendo, divisore, quoziente, resto, d
-- scambia i numero se sono in ordine inverso
```

```
if n1 < n2 then
  d=n1
  n1=n2
  n2=d
end
dividendo=n1
divisore=n2
repeat
  quoziente,d=math.modf(dividendo/divisore)
  resto=dividendo-quoziente*divisore
  if resto == 0 then
    return divisore
  else
    dividendo=divisore
    divisore=resto
  end
until resto == 0
end
```

Ora possiamo chiamare la funzione MCD nel modo seguente:

```
N1=MCD(15,27)
print(N1)
--> 3

N1=MCD(15987,31)
print(N1)
--> 1
```

Il calcolo del **mcm** (massimo comune multiplo) tra due numeri è invece molto semplice una volta abbiamo il loro **MCD**. Cioè, dati due numeri interi N1 e N2, mcm(N1,N2)= N1.N2/MCD(N1,N2). In altri termini il loro **mcm** è ottenuto dividendo uno dei numeri per il loro MCD e moltiplicandolo con l'altro.

Esempio: abbiamo visto che il MCD tra 27 e 12 è 3. Quindi $\frac{27}{3} = 9$,

perciò il **mcm**(27,12)=9 x 12 = 108.

Ed ecco il codice Lua:

```
a=27
b=12
c=a*b/MCD(a,b)
print(c)
--> 108
```

Qui finisce la prima edizione di questo quaderno.

CONCLUSIONE

Lua è un ottimo linguaggio di scripting, soprattutto come primo linguaggio di programmazione. Come avete avuto modo di vedere, è molto semplice da imparare in poco tempo. Quindi è molto indicato per scrivere piccole applicazioni utili per poi passare ad un altro linguaggio più completo e quindi più complesso. Ma come tutti i linguaggi di programmazione è ideale per fare certe cose, mentre diventa la scelta sbagliata per fare altre. Tutto dipende dal contesto e dal reale utilizzo del linguaggio.

Imparare a padroneggiare un linguaggio di programmazione richiede molto tempo, impegno e soprattutto pazienza. L'investimento diventa notevole quando si tratta di usare un linguaggio per finalità professionali. Per fortuna Lua richiede molto meno sforzo e tempo di altri linguaggi di scripting simili per fare praticamente le stesse cose. Lua è il linguaggio ideale per il programmatore della domenica.

Un altro fattore molto importante prima di investire seriamente nell'apprendimento di un nuovo linguaggio di programmazione è

quello di prevedere il futuro di un linguaggio che è una cosa non certo facile. In generale, il futuro di un linguaggio non dipende solo dalla sua bontà o meno, ma da quante persone lo utilizzano. Da questo punto di vista Lua è veramente al sicuro anche a lungo termine, visto che è il linguaggio di scripting prediletto per estendere software di giochi e non solo. Il fatto che il notissimo sistema di pubblicazione **LaTeX** stia attualmente adottando Lua (**LuaTex**) in sostituzione allo storico **pdfLaTeX** è un ottimo indicatore dell'importanza di Lua anche in futuro. Anche Corona sdk (www.coronalabs.com) che è un framework di sviluppo per applicazioni mobile cross platform ha scelto come linguaggio base Lua, senza poi dimenticare che Lua ha una vastissima comunità di sviluppatori ed utenti sparsi per tutto il mondo.

Sul sito http://luaforge.net/projects/ c'è un lunghissimo elenco di progetti scritti in Lua, mentre sul sito http://lua-users.org/wiki/LibrariesAndBindings c'è un altro elenco delle varie librerie che si interfacciano con Lua, questo piccolo grande linguaggio.

Nella prossima edizione tratteremo nuovi argomenti avanzati in Lua, come la programmazione ad oggetti, le coroutine e come interfacciare Lua con il C, il Pascal e Go.

NOTA BENE

Segnalazione di errori e suggerimenti per la nuova edizione sono sempre molto graditi.

Per concludere, come è stato già ben precisato nella prefazione, alla fine, l'utilizzo di questo quaderno è gratuito per qualsiasi scopo e potrebbe anche essere copiato, modificato e redistribuito senza il consenso dell'autore. Comunque, la scrittura di questo quaderno ha

richiesto molte ore di lavoro. Perciò, se trovate questo quaderno utile ed interessante allora potreste dare qualsiasi contributo all'autore utilizzando questo indirizzo (/www.paypal.me/jilanik). Grazie.

Alla prossima edizione!
Jilani KHALDI