Fondamenti di Informatica (Elettronici)

THINK JULIA – Chapter 14

30 novembre 2020

14. Files[1]

1. Persistenza
2. Leggere e scrivere
3. Formattazione
4. Nomi di file e percorsi
5. Catturare le eccezioni
6. Basi dati
7. Serializzazione
8. Oggetti comando
9. Moduli
10. Debug
11. Glossario
12. Esercizi

[1] From https://benlauwens.github.io/ThinkJulia.jl/latest/book.html. It is available under the Creative Commons Attribution-NonCommercial 3.0 Unported License.

Questo capitolo introduce l'idea di programmi "persistenti" che conservano i dati in una memoria permanente e mostra come utilizzare diversi tipi di memoria permanente, come file e database.

Section 1

Persistenza

Programmi persistanti

La maggior parte dei programmi che abbiamo visto finora sono transitori, nel senso che funzionano per un breve periodo e producono un output, ma quando terminano, i loro dati scompaiono.

Se esegui di nuovo il programma, inizia con uno stato vuoto.

Altri programmi sono persistenti: funzionano a lungo (o tutto il tempo); conservano almeno alcuni dei loro dati in una memoria permanente (un disco rigido, ad esempio); e se si spengono e si riavviano, riprendono da dove si erano interrotti.

Esempi di programmi persistenti sono i sistemi operativi, che vengono eseguiti praticamente ogni volta che un computer è acceso, e i server web, che girano sempre, in attesa che arrivino richieste sulla rete.

Uno dei modi più semplici con cui i programmi conservano i propri dati è leggere e scrivere file di testo.

Abbiamo già visto programmi che leggono file di testo; in questo capitolo vedremo i programmi che li scrivono.

Un'alternativa è memorizzare lo stato del programma in un database.

In questo capitolo presenterò anche come utilizzare un semplice database.

Section 2

Leggere e scrivere

Scrivere file

Un file di testo è una sequenza di caratteri memorizzata su un supporto permanente come un disco rigido o una memoria flash. Abbiamo visto come "aprire" e "leggere" un file in Lettura di elenchi di parole.

Per `scrivere` un file, devi `aprirlo` con la *:

```
julia> fout = open("output.txt", "w")
IOStream(<file output.txt>)
```

Scrivere file

Un file di testo è una sequenza di caratteri memorizzata su un supporto permanente come un disco rigido o una memoria flash. Abbiamo visto come "aprire" e "leggere" un file in Lettura di elenchi di parole.

Per `scrivere` un file, devi `aprirlo` con la *:

```
julia> fout = open("output.txt", "w")
IOStream(<file output.txt>)
```

Se il file esiste già, aprirlo in modalità `write` cancella i vecchi dati e ricomincia da capo, quindi fai attenzione! Se il file non esiste, ne viene creato uno nuovo.

"open" restituisce un oggetto file e la funzione "write" inserisce i dati nel file.

```
julia> line1 = "This here's the wattle,\n"; julia> write(fout,
24
```

Oggetto file

Il valore di "return" è il numero di caratteri che sono stati scritti.

L'oggetto file tiene traccia di dove si trova, quindi se chiami di nuovo `write`, aggiunge i nuovi dati alla fine del file.

```
julia> line2 = "the emblem of our land.\n"; julia> write(fout,
24
```

Oggetto file

Il valore di "return" è il numero di caratteri che sono stati scritti.

L'oggetto file tiene traccia di dove si trova, quindi se chiami di nuovo `write`, aggiunge i nuovi dati alla fine del file.

```
julia> line2 = "the emblem of our land.\n"; julia> write(fout,
24
```

Quando hai finito di scrivere, dovresti `chiudere` il file.

```
julia> close(fout)
```

Se non chiudi il file, verrà chiuso al termine del programma.

Section 3

Formattazione

L'argomento di "write" deve essere una "stringa"

L'argomento di `write` deve essere unastring, quindi se vogliamo mettere altri valori in un file, dobbiamo convertirli in stringhe.

Il modo più semplice per farlo è con `string` o "interpolation string":

```
julia> fout = open("output.txt", "w")
IOStream(<file output.txt>)
julia> write(fout, string(150))
3
```

Un'alternativa è usare la famiglia di funzioni`print(ln)`.

```
julia> camels = 42
42
julia> println(fout, "I have spotted $camels camels.")
```

MANCIA

Un'alternativa più potente è la macro `@printf` che stampa usando una stringa di specifica del formato in stile `C`, di cui puoi leggere ahttps://docs.julialang.org/en/v1/stdlib/Printf/

Section 4

Nomi di file e percorsi

pwd stampa la directory corrente

I file sono organizzati in directory (chiamate anche "cartelle").

Ogni programma in esecuzione ha una "directory corrente", che è la directory predefinita per la maggior parte delle operazioni.

Ad esempio, quando "apri" un file da leggere, Julia lo cerca nella directory corrente.

La funzione pwd restituisce il nome della directory corrente:

```
julia> cwd = pwd()
"/home/ben"
```

cwd sta per "current working directory".

Il risultato in questo esempio è /home/ben, che è la directory "home" di un utente chiamato ben.

Una stringa come "/home/ben" che identifica un file o una directory è chiamata path.

Percorsi assoluti e relativi

Anche un semplice nome di file, come `memo.txt`, è considerato un percorso, ma è un percorso relativo perché si riferisce alla directory corrente.

Se la directory corrente è `/home/ben`, il nome del file `memo.txt` farebbe riferimento a `/home/ben/memo.txt`.

Un percorso che inizia con / non dipende dalla directory corrente; è chiamato un percorso assoluto.

Per trovare il percorso assoluto di un file, puoi usare `abspath`:

```
julia> abspath("memo.txt")
"/home/ben/memo.txt"
```

ispath, isdir, isfile, e readdir

Julia fornisce altre funzioni per lavorare con nomi di file e percorsi.

Ad esempio, `ispath` controlla se esiste un file o una directory:

```
julia> ispath("memo.txt")
true
```

Se esiste, `isdir` controlla se si tratta di una directory:

```
julia> isdir("memo.txt")
false
julia> isdir("/home/ben")
true
```

Allo stesso modo, "`isfile`" controlla se si tratta di un file. `"readdir"` restituisce un array di nomi di file (e altre directory) nella directory data:

```
julia> readdir(cwd)
3-element Array{String,1}:
    "memo.txt"
    "music"
    "photos"
```

Attraversare una directory

Per dimostrare queste funzioni, il seguente esempio "cammina" attraverso una directory, stampa i nomi di tutti i file e chiama se stesso ricorsivamente su tutte le directory.

```
function walk(dirname)
    for name in readdir(dirname)
    path = joinpath(dirname, name)
        if isfile(path)
            println(path)
        else
            walk(path)
        end
    end
end
```

`joinpath` prende una directory e un nome di file e li unisce in un percorso completo.

MANCIA

Julia fornisce una funzione chiamata `walkdir` (vedere [* https://docs.julialang.org/en/v1/base/file/ # Base.Filesystem.walkdir *] (https://docs.julialang.org/en/v1/base/file/#Base.Filesystem.walkdir)) simile a questo ma più versatile.
Come esercizio, leggi la documentazione e usala per stampare i nomi dei file in una data directory e nelle sue sottodirectory.

Section 5

Catturare le eccezioni

Getting SystemError

Molte cose possono andare storte quando provi a leggere e scrivere file.

Se provi ad aprire un file che non esiste, ottieni un "SystemError":

```
julia> fin = open("bad_file")
ERROR: SystemError: opening file "bad_file": No such file or d
```

Getting SystemError

Molte cose possono andare storte quando provi a leggere e scrivere file.

Se provi ad aprire un file che non esiste, ottieni un "SystemError":

```
julia> fin = open("bad_file")
ERROR: SystemError: opening file "bad_file": No such file or d
```

Se non disponi dell'autorizzazione per accedere a un file:

```
julia> fout = open("/etc/passwd", "w")
ERROR: SystemError: opening file "/etc/passwd": Permission den
```

Per evitare questi errori, potresti usare funzioni come `ispath` `eisfile`, ma ci vorrebbe molto tempo e codice per verificare tutte le possibilità.

try <code> catch <exception> end

È più facile andare avanti e provare - e affrontare i problemi se si verificano - che è esattamente ciò che fa l'istruzione `try`.

La sintassi è simile a un'istruzione `if`:

```
try
    fin = open("bad_file.txt")
catch exc
    println("Something went wrong: $exc")
end
```

try <code> catch <exception> end

È più facile andare avanti e provare - e affrontare i problemi se si verificano - che è esattamente ciò che fa l'istruzione `try`.

La sintassi è simile a un'istruzione `if`:

```
try
    fin = open("bad_file.txt")
catch exc
    println("Something went wrong: $exc")
end
```

Julia inizia eseguendo la clausola `try`. Se tutto va bene, salta la clausola di cattura e procede. Se si verifica un'eccezione, salta fuori dalla clausola `try` ed esegue la clausola `catch`.

La gestione di un'eccezione con un'istruzione `try` viene chiamata cattura di un'eccezione.

In questo esempio, la clausola stampa un messaggio di errore che non è molto utile. In generale, catturare un'eccezione ti dà la possibilità di risolvere il problema, o riprovare, o almeno terminare il programma con grazia.

try <code> finally <code> end

Nel codice che esegue modifiche di stato o utilizza risorse come i file, in genere è necessario eseguire un lavoro di pulizia (come la chiusura dei file) al termine del codice.

Le eccezioni complicano potenzialmente questa attività, poiché possono causare l'uscita di un blocco di codice prima del raggiungimento della sua fine normale.

La parola chiave `finally` fornisce un modo per eseguire del codice quando un determinato blocco di codice esce, indipendentemente da come esce:

```
f = open("output.txt")
try
    line = readline(f)
    println(line)
finally
    close(f)
end
```

La funzione "close" sarà sempre eseguita.

Section 6

Basi dati

GDBM (GNU dbm) database object creation

Un database è un file organizzato per la memorizzazione dei dati.

Molti database sono organizzati come un dizionario nel senso che mappano dalle chiavi ai valori.

La più grande differenza tra un database e un dizionario è che il database è su disco (o altro archivio permanente), quindi persiste anche dopo la fine del programma.

`ThinkJulia` fornisce un'interfaccia a `GDBM` (GNU dbm) per la creazione e l'aggiornamento dei file di database.

Ad esempio, creerò un database che contiene didascalie per i file di immagine. L'apertura di un database è simile all'apertura di altri file:

```
julia> using ThinkJulia
julia> db = DBM("captions", "c")
DBM(<captions>)
```

La modalità "c" significa che il database dovrebbe essere creato se non esiste già.

Il risultato è un oggetto database che può essere utilizzato (per la maggior parte delle operazioni) come un dizionario.

Lettura e accesso al database

Quando crei un nuovo elemento, GDBM aggiorna il file del database:

```
julia> db["cleese.png"] = "Photo of John Cleese."
"Photo of John Cleese."
```

Lettura e accesso al database

Quando crei un nuovo elemento, GDBM aggiorna il file del database:

```
julia> db["cleese.png"] = "Photo of John Cleese."
"Photo of John Cleese."
```

Quando accedi a uno degli elementi, GDBM legge il file:

```
julia> db ["cleese.png"]
```

```
julia> db["cleese.png"]
"Photo of John Cleese."
```

Se esegui un'altra assegnazione a una chiave esistente, GDBM sostituisce il vecchio valore:

```
julia> db["cleese.png"] = "Photo of John Cleese doing a silly walk."
"Photo of John Cleese doing a silly walk."
julia> db["cleese.png"]
"Photo of John Cleese doing a silly walk."
```

Database vs dizionario in Julia

Alcune funzioni che hanno un dizionario come argomento, ovvero come "chiavi" e/o "valori", non funzionano con gli oggetti database.

Ma l'iterazione con un ciclo `for` funziona:

```
for (key, value) in db
    println(key, ": ", value)
end
```

Database vs dizionario in Julia

Alcune funzioni che hanno un dizionario come argomento, ovvero come "chiavi" e/o "valori", non funzionano con gli oggetti database.

Ma l'iterazione con un ciclo `for` funziona:

```
for (key, value) in db
    println(key, ": ", value)
end
```

Come con altri file, dovresti `chiudere` il database quando hai finito:

```
julia> close(db)
```

Section 7

Serializzazione

Funzioni serialize e deserialize 1/3

Una limitazione di "GDBM" è che le "chiavi" e i "valori" devono essere stringhe o array di byte. Se provi a utilizzare qualsiasi altro tipo, ottieni un errore.

Le funzioni `serialize` `edeserialize` possono aiutare.

Traducono quasi ogni tipo di oggetto in un array di byte (un `iobuffer`) adatto per l'archiviazione in un database, quindi traduce nuovamente gli array di byte in oggetti:

```
julia> using Serialization

julia> io = IOBuffer();

julia> t = [1, 2, 3];

julia> serialize(io, t)
24
julia> print(take!(io))
UInt8[0x37, 0x4a, 0x4c, 0x07, 0x04, 0x00, 0x00, 0x00, 0x15, 0x00, 0x08,
0xe2, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0
```

Funzioni serialize e deserialize 2/3

Il formato non è ovvio per i lettori umani; è pensato per essere facile da interpretare per Julia.

`deserialize` ricostituisce l'oggetto:

```
julia> io = IOBuffer();
julia> t1 = [1, 2, 3];
julia> serialize(io, t1)
24
julia> s = take!(io);
julia> t2 = deserialize(IOBuffer(s));
julia> print(t2)
[1, 2, 3]
```

`serialize` e`deserialize` scrivono su e leggono da un oggetto `iobuffer` che rappresenta uno `stream` di I/O in memoria. La funzione `take!` recupera il contenuto di `iobuffer` come un array di byte e ripristina `iobuffer` al suo stato iniziale.

Funzioni `serialize` e `deserialize` 3/3

Sebbene il nuovo oggetto abbia lo stesso valore del vecchio, non è (in generale) lo stesso oggetto:

```
julia> t1 == t2
true
julia> t1 === t2
false
```

In altre parole, la serializzazione e quindi la deserializzazione hanno lo stesso effetto della copia dell'oggetto.

Puoi usarlo per memorizzare le non-stringhe in un database.

MANCIA

In effetti, l'archiviazione di elementi non di stringa in un database è così comune che è stata incapsulata in un package chiamato JLD2 (vedere https://github.com/JuliaIO/JLD2.jl).

Section 8

Oggetti comando

Interfaccia della riga di comando (shell)

La maggior parte dei sistemi operativi fornisce un'interfaccia a riga di comando, nota anche come `shell`.

Le `shell` di solito forniscono comandi per navigare nel file system e avviare le applicazioni.

Ad esempio, in `Unix` puoi cambiare directory con `cd`, visualizzare il contenuto di una directory con `ls` e avviare un browser web digitando (ad esempio) `firefox`.

Qualsiasi programma che puoi avviare dalla shell può anche essere avviato da Julia utilizzando un oggetto comando:

```
julia> cmd = `echo ciao`
`echo ciao`
```

I `backtick` vengono utilizzati per delimitare il comando.

Eseguire `run` un oggetto comando

La funzione `run` esegue il comando:

```
julia> run(cmd);
Ciao
```

"«" è l'output del comando "`echo`", inviato a "`STDOUT`".

La stessa funzione `run` restituisce un oggetto di tipo processo e lancia un'eccezione `ErrorException` se il comando esterno non viene eseguito correttamente.

Se vuoi leggere l'output del comando esterno, puoi invece usare `read`:

```
julia> a = read(cmd, String)
"Ciao\n"
```

Esempio `md5sum`

Ad esempio, la maggior parte dei sistemi Unix fornisce un comando chiamato "`md5sum`" o "`md5`" che legge il contenuto di un file e calcola un "`checksum`".

Puoi leggere informazioni su `MD5` su * https: //en.wikipedia.org/wiki/Md5*.

Questo comando fornisce un modo efficiente per verificare se due file hanno lo stesso contenuto.

La probabilità che contenuti diversi producano lo stesso checksum è molto piccola (ovvero, è improbabile che accada prima che l'universo collassi :-).

Puoi usare un oggetto comando per eseguire `md5` da Julia e ottenere il risultato:

```
julia> filename = "output.txt"
"output.txt"
julia> cmd = `md5 $filename`
`md5 output.txt`
julia> res = read(cmd, String)
"MD5 (output.txt) = d41d8cd98f00b204e9800998ecf8427e\n"
```

Section 9

Moduli

File include nel REPL

Supponiamo di avere un file denominato "`wc.jl`" con il seguente codice:

```
function linecount(filename)
    count = 0
    for line in eachline(filename)
        count += 1
    end
    count
end

print(linecount("wc.jl"))
```

Se esegui questo programma, si legge da solo (sic!) e stampa il numero di righe nel file, che è 9. Puoi anche `includerlo` nella REPL in questo modo:

```
julia> include("wc.jl")
9
```

File include nel REPL

Supponiamo di avere un file denominato "wc.jl" con il seguente codice:

```
function linecount(filename)
    count = 0
    for line in eachline(filename)
        count += 1
    end
    count
end

print(linecount("wc.jl"))
```

Se esegui questo programma, si legge da solo (sic!) e stampa il numero di righe nel file, che è 9. Puoi anche includerlo nella REPL in questo modo:

```
julia> include("wc.jl")
9
```

NOTA

Julia introduce moduli per creare separati workspace delle variabili, ovvero nuovi ambiti globali.

import edexport da module

Un modulo inizia con la parola chiave `module` e termina con `end`.

Si evitano conflitti di denominazione tra le proprie definizioni di primo livello e quelle che si trovano nel codice di qualcun altro.

`import` permette di controllare quali nomi da altri moduli sono visibili e `export` specifica quali dei tuoi nomi sono `pubblici`, ad es. può essere utilizzato al di fuori del modulo senza essere preceduto dal "nome" del modulo.

```
module LineCount
export linecount

function linecount(filename)
    count = 0
    for line in eachline(filename)
        count += 1
    end
    count
end
end #module
```

using <module-name>

L'oggetto modulo `LineCount` fornisce la funzione `linecount`:

```
julia> using LineCount
julia> linecount("wc.jl")
11
```

Esercizio 14-1

Digita questo esempio in un file chiamato "`wc.jl`", includilo in "`REPL`" e inserisci "`using LineCount`".

AVVERTIMENTO

- Se `importi` un modulo che è già stato importato, Julia non fa nulla.

using <module-name>

L'oggetto modulo `LineCount` fornisce la funzione `linecount`:

```
julia> using LineCount
julia> linecount("wc.jl")
11
```

Esercizio 14-1

Digita questo esempio in un file chiamato "`wc.jl`", includilo in "`REPL`" e inserisci "`using LineCount`".

AVVERTIMENTO

- Se `importi` un modulo che è già stato importato, Julia non fa nulla.
- Non rilegge il file, anche se è cambiato.

using <module-name>

L'oggetto modulo `LineCount` fornisce la funzione `linecount`:

```
julia> using LineCount
julia> linecount("wc.jl")
11
```

Esercizio 14-1

Digita questo esempio in un file chiamato "`wc.jl`", includilo in "`REPL`" e inserisci "`using LineCount`".

AVVERTIMENTO

- Se `importi` un modulo che è già stato importato, Julia non fa nulla.
- Non rilegge il file, anche se è cambiato.
- Se vuoi `ricaricare` un modulo, devi riavviare`REPL`.

using <module-name>

L'oggetto modulo `LineCount` fornisce la funzione `linecount`:

```
julia> using LineCount
julia> linecount("wc.jl")
11
```

Esercizio 14-1

Digita questo esempio in un file chiamato "`wc.jl`", includilo in "`REPL`" e inserisci "`using LineCount`".

AVVERTIMENTO

- Se `importi` un modulo che è già stato importato, Julia non fa nulla.
- Non rilegge il file, anche se è cambiato.
- Se vuoi `ricaricare` un modulo, devi riavviare`REPL`.
- Esiste un pacchetto `Revise` che può mantenere le sessioni in esecuzione più a lungo (vedere https: //github.com/timholy/Revise.jl).

Section 10

Debug

Functions repr and dump

Durante la lettura e la scrittura di file, potrebbero verificarsi problemi con gli spazi. Questi errori possono essere difficili da "debuggare" perché gli spazi, le tabulazioni e le nuove righe sono normalmente non visibili:

```
julia> s = "1 2\t 3\n 4"
"1 2\t 3\n 4"
julia> println(s)
1 2  3
 4
```

La funzione predefinita `repr` o `dump` può aiutare. Accetta qualsiasi oggetto come argomento e restituisce una rappresentazione di stringa dell'oggetto.

```
julia> repr(s)
"\"1 2\\t 3\\n 4\""
julia> dump(s)
String "1 2\t 3\n 4"
```

Questo può essere utile per il debug.

Caratteri diversi per "fine riga" (`end-of-line`) 1/2

Un altro problema che potresti incontrare è che sistemi diversi utilizzano caratteri diversi per indicare la fine di una riga.

1. Alcuni sistemi (Uni-like) creano una nuova riga (LF = `\n`), rappresentata da "`\n`".

Caratteri diversi per "fine riga" (`end-of-line`) 1/2

Un altro problema che potresti incontrare è che sistemi diversi utilizzano caratteri diversi per indicare la fine di una riga.

1. Alcuni sistemi (Uni-like) creano una nuova riga (LF = `\n`), rappresentata da "`\n`".
2. Altri (MacOS) usano un carattere di ritorno carrello (CR = `\r`), rappresentato da "`\r`".

Caratteri diversi per "fine riga" (`end-of-line`) 1/2

Un altro problema che potresti incontrare è che sistemi diversi utilizzano caratteri diversi per indicare la fine di una riga.

1. Alcuni sistemi (Uni-like) creano una nuova riga (LF = `\n`), rappresentata da "`\n`".
2. Altri (MacOS) usano un carattere di ritorno carrello (CR = `\r`), rappresentato da "`\r`".
3. Alcuni (Windows) usano entrambi (`CR LF = \r\n`) rappresentato da "`\r\n`".

Caratteri diversi per "fine riga" (`end-of-line`) 1/2

Un altro problema che potresti incontrare è che sistemi diversi utilizzano caratteri diversi per indicare la fine di una riga.

1. Alcuni sistemi (Uni-like) creano una nuova riga (LF = `\n`), rappresentata da "`\n`".
2. Altri (MacOS) usano un carattere di ritorno carrello (CR = `\r`), rappresentato da "`\r`".
3. Alcuni (Windows) usano entrambi (`CR LF = \r\n`) rappresentato da "`\r\n`".

Caratteri diversi per "fine riga" (`end-of-line`) 1/2

Un altro problema che potresti incontrare è che sistemi diversi utilizzano caratteri diversi per indicare la fine di una riga.

1. Alcuni sistemi (Uni-like) creano una nuova riga (LF = `\n`), rappresentata da "`\n`".
2. Altri (MacOS) usano un carattere di ritorno carrello (CR = `\r`), rappresentato da "`\r`".
3. Alcuni (Windows) usano entrambi (`CR LF = \r\n`) rappresentato da "`\r\n`".

Se sposti file tra sistemi diversi, queste incongruenze possono causare problemi.

Per la maggior parte dei sistemi, esistono applicazioni per la conversione da un formato all'altro.

Puoi trovarli (e leggere di più su questo problema) su https://en.wikipedia.org/wiki/Newline.

O, naturalmente, potresti scriverne uno tu stesso.

Caratteri diversi per “fine riga” 2/2

Newline (spesso chiamato `end-of-line` (`EOL`), avanzamento riga o interruzione di riga) è un carattere di controllo o una sequenza di caratteri di controllo in una codifica dei caratteri.

Il termine CRLF si riferisce a `Carriage Return` (ASCII 13, `\r`) `Line Feed` (ASCII 10, `\n`).

Per esempio: * in Windows sia una CR che una LF sono necessarie per annotare la fine di una riga, mentre

- in Linux / UNIX è richiesto solo un LF.

Caratteri diversi per “fine riga” 2/2

Newline (spesso chiamato `end-of-line` (`EOL`), avanzamento riga o interruzione di riga) è un carattere di controllo o una sequenza di caratteri di controllo in una codifica dei caratteri.

Il termine CRLF si riferisce a `Carriage Return` (ASCII 13, `\r`) `Line Feed` (ASCII 10, `\n`).

Per esempio: * in Windows sia una CR che una LF sono necessarie per annotare la fine di una riga, mentre

- in Linux / UNIX è richiesto solo un LF.
- In HTTP, la sequenza `CR-LF` viene sempre utilizzata per terminare una riga.

Section 11

Glossario

Glossario

persistente Pertinente a un programma che viene eseguito a tempo indeterminato e conserva almeno alcuni dei suoi dati in una memoria permanente.

Glossario

persistente Pertinente a un programma che viene eseguito a tempo indeterminato e conserva almeno alcuni dei suoi dati in una memoria permanente.

file di testo Una sequenza di caratteri archiviata in una memoria permanente come un disco rigido.

Glossario

persistente Pertinente a un programma che viene eseguito a tempo indeterminato e conserva almeno alcuni dei suoi dati in una memoria permanente.

file di testo Una sequenza di caratteri archiviata in una memoria permanente come un disco rigido.

directory Una raccolta di file denominata, chiamata anche cartella.

Glossario

persistente Pertinente a un programma che viene eseguito a tempo indeterminato e conserva almeno alcuni dei suoi dati in una memoria permanente.

file di testo Una sequenza di caratteri archiviata in una memoria permanente come un disco rigido.

directory Una raccolta di file denominata, chiamata anche cartella.

sentiero (path, o percorso) Una stringa che identifica un file.

Glossario

persistente Pertinente a un programma che viene eseguito a tempo indeterminato e conserva almeno alcuni dei suoi dati in una memoria permanente.

file di testo Una sequenza di caratteri archiviata in una memoria permanente come un disco rigido.

directory Una raccolta di file denominata, chiamata anche cartella.

sentiero (path, o percorso) Una stringa che identifica un file.

percorso relativo Un percorso che inizia dalla directory corrente.

Glossario

persistente Pertinente a un programma che viene eseguito a tempo indeterminato e conserva almeno alcuni dei suoi dati in una memoria permanente.

file di testo Una sequenza di caratteri archiviata in una memoria permanente come un disco rigido.

directory Una raccolta di file denominata, chiamata anche cartella.

sentiero (path, o percorso) Una stringa che identifica un file.

percorso relativo Un percorso che inizia dalla directory corrente.

percorso assoluto Un percorso che inizia dalla directory più in alto nel file system.

Glossario

persistente Pertinente a un programma che viene eseguito a tempo indeterminato e conserva almeno alcuni dei suoi dati in una memoria permanente.

file di testo Una sequenza di caratteri archiviata in una memoria permanente come un disco rigido.

directory Una raccolta di file denominata, chiamata anche cartella.

sentiero (path, o percorso) Una stringa che identifica un file.

percorso relativo Un percorso che inizia dalla directory corrente.

percorso assoluto Un percorso che inizia dalla directory più in alto nel file system.

catturare Per evitare che un'eccezione termini un programma usando le istruzioni `try ... catch ... finally`.

Glossario

persistente Pertinente a un programma che viene eseguito a tempo indeterminato e conserva almeno alcuni dei suoi dati in una memoria permanente.

file di testo Una sequenza di caratteri archiviata in una memoria permanente come un disco rigido.

directory Una raccolta di file denominata, chiamata anche cartella.

sentiero (path, o percorso) Una stringa che identifica un file.

percorso relativo Un percorso che inizia dalla directory corrente.

percorso assoluto Un percorso che inizia dalla directory più in alto nel file system.

catturare Per evitare che un'eccezione termini un programma usando le istruzioni `try ... catch ... finally`.

Base dati Un file il cui contenuto è organizzato come un dizionario con chiavi che corrispondono a valori.

Glossario

persistente Pertinente a un programma che viene eseguito a tempo indeterminato e conserva almeno alcuni dei suoi dati in una memoria permanente.

file di testo Una sequenza di caratteri archiviata in una memoria permanente come un disco rigido.

directory Una raccolta di file denominata, chiamata anche cartella.

sentiero (path, o percorso) Una stringa che identifica un file.

percorso relativo Un percorso che inizia dalla directory corrente.

percorso assoluto Un percorso che inizia dalla directory più in alto nel file system.

catturare Per evitare che un'eccezione termini un programma usando le istruzioni `try ... catch ... finally`.

Base dati Un file il cui contenuto è organizzato come un dizionario con chiavi che corrispondono a valori.

shell Un programma che consente agli utenti di digitare comandi e quindi di eseguirli avviando altri programmi.

Glossario

persistente Pertinente a un programma che viene eseguito a tempo indeterminato e conserva almeno alcuni dei suoi dati in una memoria permanente.

file di testo Una sequenza di caratteri archiviata in una memoria permanente come un disco rigido.

directory Una raccolta di file denominata, chiamata anche cartella.

sentiero (path, o percorso) Una stringa che identifica un file.

percorso relativo Un percorso che inizia dalla directory corrente.

percorso assoluto Un percorso che inizia dalla directory più in alto nel file system.

catturare Per evitare che un'eccezione termini un programma usando le istruzioni `try ... catch ... finally`.

Base dati Un file il cui contenuto è organizzato come un dizionario con chiavi che corrispondono a valori.

shell Un programma che consente agli utenti di digitare comandi e quindi di eseguirli avviando altri programmi.

oggetto comando Un oggetto che rappresenta un comando della shell, che consente a un programma Julia di eseguire comandi e leggere i risultati.

Section 12

Esercizi

aaaa

aaaa