Capitolo 1

Imparare a programmare

L'obiettivo di questo libro è insegnarti a pensare da informatico. Questo modo di pensare combina alcune delle migliori caratteristiche della matematica, dell'ingegneria e delle scienze naturali. Come i matematici, gli informatici usano linguaggi formali per denotare idee (nella fattispecie elaborazioni). Come gli ingegneri progettano cose, assemblano componenti in sistemi e cercano compromessi tra le varie alternative. Come gli scienziati osservano il comportamento di sistemi complessi, formulano ipotesi e verificano previsioni.

La più importante capacità di un informatico è quella di **risolvere problemi**. Risolvere problemi significa avere l'abilità di schematizzarli, pensare creativamente alle possibili soluzioni ed esprimerle in modo chiaro ed accurato. Da ciò emerge che il processo di imparare a programmare è un'eccellente opportunità di mettere in pratica l'abilità di risolvere problemi.

Da una parte ti sarà insegnato a programmare, già di per sé un'utile capacità. Dall'altra userai la programmazione come un mezzo rivolto ad un fine. Mentre procederemo quel fine ti diverrà più chiaro.

1.1 Il linguaggio di programmazione Python

Il linguaggio di programmazione che imparerai è il Python. Python è un esempio di **linguaggio di alto livello**; altri linguaggi di alto livello di cui puoi aver sentito parlare sono il C, il C++, il Perl ed il Java.

Come puoi immaginare sentendo la definizione "linguaggio di alto livello" esistono anche **linguaggi di basso livello**, talvolta chiamati "linguaggi macchina" o "linguaggi assembly". In modo non del tutto corretto si può affermare che i computer possono eseguire soltanto programmi scritti in linguaggi di basso livello: i programmi scritti in un linguaggio di alto livello devono essere elaborati prima di poter essere eseguiti. Questo processo di elaborazione impiega del tempo e rappresenta un piccolo svantaggio dei linguaggi di alto livello.

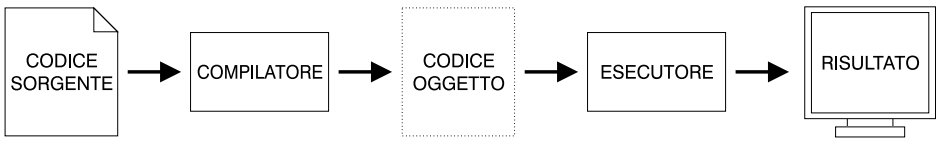
I vantaggi sono d'altra parte enormi. In primo luogo è molto più facile programmare in un linguaggio ad alto livello: questi tipi di programmi sono più

veloci da scrivere, più corti e facilmente leggibili, ed è più probabile che siano corretti. In secondo luogo i linguaggi di alto livello sono portabili: **portabilità** significa che essi possono essere eseguiti su tipi di computer diversi con poche o addirittura nessuna modifica. I programmi scritti in linguaggi di basso livello possono essere eseguiti solo su un tipo di computer e devono essere riscritti per essere trasportati su un altro sistema.

Questi vantaggi sono così evidenti che quasi tutti i programmi sono scritti in linguaggi di alto livello, lasciando spazio ai linguaggi di basso livello solo in poche applicazioni specializzate.

I programmi di alto livello vengono trasformati in programmi di basso livello eseguibili dal computer tramite due tipi di elaborazione: l'**interpretazione** e la **compilazione**. Un interprete legge il programma di alto livello e lo esegue, trasformando ogni riga di istruzioni in un'azione. L'interprete elabora il programma un po' alla volta, alternando la lettura delle istruzioni all'esecuzione dei comandi che le istruzioni descrivono:

Un compilatore legge il programma di alto livello e lo traduce completamente in basso livello, prima che il programma possa essere eseguito. In questo caso il programma di alto livello viene chiamato **codice sorgente**, ed il programma tradotto **codice oggetto** o **eseguibile**. Dopo che un programma è stato compilato può essere eseguito ripetutamente senza che si rendano necessarie ulteriori compilazioni finché non ne viene modificato il codice.



Python è considerato un linguaggio interpretato perché i programmi Python sono eseguiti da un interprete. Ci sono due modi di usare l'interprete: a linea di comando o in modo script. In modo "linea di comando" si scrivono i programmi Python una riga alla volta: dopo avere scritto una riga di codice alla pressione di Invio (o Enter, a seconda della tastiera) l'interprete la analizza subito ed elabora immediatamente il risultato, eventualmente stampandolo a video:

```
$ python
Python 1.5.2 (#1, Feb 1 2000, 16:32:16)
Copyright 1991-1995 Stichting Mathematish Centrum, Amsterdam
>>> print 1 + 1
2
```

La prima linea di questo esempio è il comando che fa partire l'interprete Python in ambiente Linux e può cambiare leggermente a seconda del sistema operativo utilizzato. Le due righe successive sono semplici informazioni di copyright del programma.

La terza riga inizia con `>>>`: questa è l'indicazione (chiamata "prompt") che l'interprete usa per indicare la sua disponibilità ad accettare comandi. Noi

abbiamo inserito `print 1 + 1` e l'interprete ha risposto con `2`.

In alternativa alla riga di comando si può scrivere un programma in un file (detto **script**) ed usare l'interprete per eseguire il contenuto del file. Nell'esempio seguente abbiamo usato un editor di testi per creare un file chiamato `pippo.py`:

```
print 1 + 1
```

Per convenzione, i file contenenti programmi Python hanno nomi che terminano con `.py`.

Per eseguire il programma dobbiamo dire all'interprete il nome dello script:

```
$ python pippo.py
2
```

In altri ambienti di sviluppo i dettagli dell'esecuzione dei programmi possono essere diversi.

La gran parte degli esempi di questo libro sono eseguiti da linea di comando: lavorare da linea di comando è conveniente per lo sviluppo e per il test del programma perché si possono inserire ed eseguire immediatamente singole righe di codice. Quando si ha un programma funzionante lo si dovrebbe salvare in uno script per poterlo eseguire o modificare in futuro senza doverlo riscrivere da capo ogni volta. Tutto ciò che viene scritto in modo "linea di comando" è irrimediabilmente perso nel momento in cui usciamo dall'ambiente Python.

1.2 Cos'è un programma?

Un **programma** è una sequenza di istruzioni che specificano come effettuare una elaborazione. L'elaborazione può essere sia di tipo matematico (per esempio la soluzione di un sistema di equazioni o il calcolo delle radici di un polinomio) che simbolico (per esempio la ricerca e sostituzione di un testo in un documento).

I dettagli sono diversi per ciascun linguaggio di programmazione, ma un piccolo gruppo di istruzioni è praticamente comune a tutti:

- **input:** ricezione di dati da tastiera, da file o da altro dispositivo.
- **output:** scrittura di dati su video, su file o trasmissione ad altro dispositivo.
- **matematiche:** esecuzione di semplici operazioni matematiche, quali l'addizione e la sottrazione.
- **condizionali:** controllo di alcune condizioni ed esecuzione della sequenza di istruzioni appropriata.
- **ripetizione:** ripetizione di un'azione, di solito con qualche variazione.

Che ci si creda o meno, questo è più o meno tutto quello che c'è. Ogni programma che hai usato per quanto complesso possa sembrare (anche il tuo videogioco preferito) è costituito da istruzioni che assomigliano a queste. Possiamo affermare che la programmazione altro non è che la suddivisione di un compito grande e complesso in una serie di sotto-compiti via via più piccoli, finché questi sono sufficientemente semplici da essere eseguiti da una di queste istruzioni fondamentali.

Questo concetto può sembrare un po' vago, ma lo riprenderemo quando parleremo di **algoritmi**.

1.3 Cos'è il debug?

La programmazione è un processo complesso e dato che esso è fatto da esseri umani spesso comporta errori. Per ragioni bizzarre gli errori di programmazione sono chiamati **bug** ed il processo della loro ricerca e correzione è chiamato **debug**.

Sono tre i tipi di errore nei quali si incorre durante la programmazione: gli errori di sintassi, gli errori in esecuzione e gli errori di semantica. È utile distinguerli per poterli individuare più velocemente.

1.3.1 Errori di sintassi

Python può eseguire un programma solo se il programma è sintatticamente corretto, altrimenti l'elaborazione fallisce e l'interprete ritorna un messaggio d'errore. La **sintassi** si riferisce alla struttura di un programma e alle regole concernenti la sua struttura. In italiano, per fare un esempio, una frase deve iniziare con una lettera maiuscola e terminare con un punto. *questa frase contiene un* **errore di sintassi**. *E anche questa*

Per la maggior parte dei lettori qualche errore di sintassi non è un problema significativo, tanto che possiamo leggere le poesie di E.E.Cummings (prive di punteggiatura) senza "messaggi d'errore". Python non è così permissivo: se c'è un singolo errore di sintassi da qualche parte nel programma Python stamperà un messaggio d'errore e ne interromperà l'esecuzione, rendendo impossibile proseguire. Durante le prime settimane della tua carriera di programmatore probabilmente passerai molto tempo a ricercare errori di sintassi. Via via che acquisirai esperienza questi si faranno meno numerosi e sarà sempre più facile rintracciarli.

1.3.2 Errori in esecuzione

Il secondo tipo di errore è l'**errore in esecuzione** (o "runtime"), così chiamato perché l'errore non appare finché il programma non è eseguito. Questi errori sono anche chiamati **eccezioni** perché indicano che è accaduto qualcosa di eccezionale nel corso dell'esecuzione (per esempio si è cercato di dividere un numero per zero).

Gli errori in esecuzione sono rari nei semplici programmi che vedrai nei primissimi capitoli, così potrebbe passare un po' di tempo prima che tu ne incontri uno.

1.3.3 Errori di semantica

Il terzo tipo di errore è l'**errore di semantica**. Se c'è un errore di semantica il programma verrà eseguito senza problemi nel senso che il computer non genererà messaggi d'errore durante l'esecuzione, ma il risultato non sarà ciò che ci si aspettava. Sarà qualcosa di diverso, e questo qualcosa è esattamente ciò che è stato detto di fare al computer.

Il problema sta nel fatto che il programma che è stato scritto non è quello che si desiderava scrivere: il significato del programma (la sua semantica) è sbagliato. L'identificazione degli errori di semantica è un processo complesso perché richiede di lavorare in modo inconsueto, guardando i risultati dell'esecuzione e cercando di capire cosa il programma ha fatto di sbagliato per ottenerli.

1.3.4 Debug sperimentale

Una delle più importanti abilità che acquisirai è la capacità di effettuare il debug (o "rimozione degli errori"). Sebbene questo possa essere un processo frustrante è anche una delle parti più intellettualmente vivaci, stimolanti ed interessanti della programmazione.

In un certo senso il debug può essere paragonato al lavoro investigativo. Sei messo di fronte agli indizi e devi ricostruire i processi e gli eventi che hanno portato ai risultati che hai ottenuto.

Il debug è una scienza sperimentale: dopo che hai avuto un'idea di ciò che può essere andato storto, modifichi il programma e lo provi ancora. Se la tua ipotesi era corretta allora puoi predire il risultato della modifica e puoi avvicinarti di un ulteriore passo all'avere un programma funzionante. Se la tua ipotesi era sbagliata devi ricercarne un'altra. Come disse Sherlock Holmes "Quando hai eliminato l'impossibile ciò che rimane, per quanto improbabile, deve essere la verità" (A.Conan Doyle, *Il segno dei quattro*)

Per qualcuno la programmazione e il debug sono la stessa cosa, intendendo con questo che la programmazione è un processo di rimozione di errori finché il programma fa ciò che ci si aspetta. L'idea è che si dovrebbe partire da un programma che fa *qualcosa* e facendo piccole modifiche ed eliminando gli errori man mano che si procede si dovrebbe avere in ogni momento un programma funzionante sempre più completo.

Linux, per fare un esempio, è un sistema operativo che contiene migliaia di righe di codice, ma esso è nato come un semplice programma che Linus Torvalds usò per esplorare il chip 80386 Intel. Secondo Larry Greenfields, "uno dei progetti iniziali di Linus era un programma che doveva cambiare una riga di AAAA in BBBB e viceversa. Questo in seguito diventò Linux." (*The Linux Users' Guide* Beta Version 1)

I capitoli successivi ti forniranno ulteriori suggerimenti sia per quanto riguarda il debug che per altre pratiche di programmazione.

1.4 Linguaggi formali e naturali

I **linguaggi naturali** sono le lingue parlate, tipo l'inglese, l'italiano, lo spagnolo. Non sono stati "progettati" da qualcuno e anche se è stato imposto un certo ordine nel loro sviluppo si sono evoluti naturalmente.

I **linguaggi formali** sono linguaggi progettati per specifiche applicazioni.

Per fare qualche esempio, la notazione matematica è un linguaggio formale particolarmente indicato ad esprimere relazioni tra numeri e simboli; i chimici usano un linguaggio formale per rappresentare la struttura delle molecole; cosa più importante dal nostro punto di vista, *i linguaggi di programmazione sono linguaggi formali che sono stati progettati per esprimere elaborazioni.*

I linguaggi formali tendono ad essere piuttosto rigidi per quanto riguarda la sintassi: $3 + 3 = 6$ è una dichiarazione matematica sintatticamente corretta, mentre $3 = \div 6\$$ non lo è. H_2O è un simbolo chimico sintatticamente corretto contrariamente a ${}_2Zz$.

Le regole sintattiche si possono dividere in due categorie: la prima riguarda i **token**, la seconda la **struttura**. I token sono gli elementi di base del linguaggio (quali possono essere le parole in letteratura, i numeri in matematica e gli elementi chimici in chimica). Uno dei problemi con $3 = \div 6\$$ è che \$ non è un token valido in matematica; ${}_2Zz$ non è valido perché nessun elemento chimico è identificato dal simbolo Zz.

Il secondo tipo di regola riguarda la struttura della dichiarazione, cioè il modo in cui i token sono disposti. La dichiarazione $3 = \div 6\$$ è strutturalmente non valida perché un segno $\div$ non può essere posto immediatamente dopo un segno $=$. Allo stesso modo l'indice nelle formule chimiche deve essere indicato dopo il simbolo dell'elementi chimico, non prima, e quindi l'espressione ${}_2Zz$ non è valida.

> *Come esercizio crea quella che può sembrare una frase in italiano con dei token non riconoscibili. Poi scrivi un'altra frase con tutti i token validi ma con una struttura non valida.*

Quando leggi una frase in italiano o una dichiarazione in un linguaggio formale devi capire quale sia la struttura della dichiarazione. Questo processo (chiamato **parsing**) in un linguaggio naturale viene realizzato in modo inconscio e spesso non ci si rende conto della sua intrinseca complessità.

Per esempio, quando senti la frase "La scarpa è caduta", capisci che "la scarpa" è il soggetto e che "è caduta" è il verbo. Quando hai analizzato la frase puoi capire cosa essa significa (cioè la semantica della frase). Partendo dal presupposto che tu sappia cosa sia una "scarpa" e cosa significhi "cadere" riesci a comprendere il significato generale della frase.

Anche se i linguaggi formali e quelli naturali condividono molte caratteristiche (token, struttura, sintassi e semantica) ci sono tuttavia molte differenze:

Ambiguità: i linguaggi naturali ne sono pieni ed il significato viene ottenuto anche grazie ad indizi ricavati dal contesto. I linguaggi formali sono progettati per essere completamente non ambigui e ciò significa che ciascuna dichiarazione ha esattamente un significato, indipendente dal contesto.

Ridondanza: per evitare l'ambiguità e ridurre le incomprensioni i linguaggi naturali impiegano molta ridondanza. I linguaggi formali sono meno ridondanti e più concisi.

Letteralità: i linguaggi naturali fanno uso di paragoni e metafore, e possiamo parlare in termini astratti intuendo immediatamente che ciò che sentiamo ha un significato simbolico. I linguaggi formali invece esprimono esattamente ciò che dicono.

Anche se siamo cresciuti apprendendo un linguaggio naturale, la nostra lingua madre, spesso abbiamo difficoltà ad adattarci ai linguaggi formali. In un certo senso la differenza tra linguaggi naturali e formali è come quella esistente tra poesia e prosa, ma in misura decisamente più evidente:

Poesia: le parole sono usate tanto per il loro suono che per il loro significato, e la poesia nel suo complesso crea un effetto o una risposta emotiva. L'ambiguità è non solo frequente, ma spesso addirittura cercata.

Prosa: il significato delle parole è estremamente importante, con la struttura che contribuisce a fornire maggior significato. La prosa può essere soggetta ad analisi più facilmente della poesia, ma può risultare ancora ambigua.

Programmi: il significato di un programma per computer è non ambiguo e assolutamente letterale, può essere compreso nella sua interezza con l'analisi dei token e della struttura.

Qui sono esposti alcuni suggerimenti per la lettura di programmi e di altri linguaggi formali.

- Ricorda che i linguaggi formali sono molto più ricchi di significato dei linguaggi naturali, così è necessario più tempo per leggerli e comprenderli.

- La struttura dei linguaggi formali è molto importante e solitamente non è una buona idea leggerli dall'alto in basso, da sinistra a destra, come avviene per un testo letterario: impara ad analizzare il programma nella tua testa, identificandone i token ed interpretandone la struttura.

- I dettagli sono importanti: piccole cose come errori di ortografia e cattiva punteggiatura sono spesso trascurabili nei linguaggi naturali, ma possono fare una gran differenza in quelli formali.

1.5 Il primo programma

Per tradizione il primo programma scritto in un nuovo linguaggio è chiamato "Hello, World!" perché tutto ciò che fa è scrivere le parole `Hello, World!` a video e nient'altro. In Python questo programma è scritto così:

```
>>> print "Hello, World!"
```

Questo è un esempio di **istruzione di stampa**, che in effetti non stampa nulla su carta limitandosi invece a scrivere un valore sullo schermo. In questo caso ciò che viene "stampato" sono le parole

```
Hello, World!
```

Le virgolette segnano l'inizio e la fine del valore da stampare ed esse non appaiono nel risultato.

Alcune persone giudicano la qualità di un linguaggio di programmazione dalla semplicità del programma "Hello, World!": da questo punto di vista Python sembra essere quanto di meglio sia realizzabile.

1.6 Glossario

Soluzione di problemi: il processo di formulare un problema, trovare una soluzione ed esprimerla.

Linguaggio ad alto livello: un linguaggio di programmazione tipo Python che è progettato per essere facilmente leggibile e utilizzabile dagli esseri umani.

Linguaggio di basso livello: un linguaggio di programmazione che è progettato per essere facilmente eseguibile da un computer; è anche chiamato "linguaggio macchina" o "linguaggio assembly".

Portabilità: caratteristica di un programma di poter essere eseguito su computer di tipo diverso.

Interpretare: eseguire un programma scritto in un linguaggio di alto livello traducendolo ed eseguendolo immediatamente, una linea alla volta.

Compilare: tradurre un programma scritto in un linguaggio di alto livello in un programma di basso livello come preparazione alla successiva esecuzione.

Codice sorgente: un programma di alto livello prima di essere compilato.

Codice oggetto: il risultato ottenuto da un compilatore dopo aver tradotto il codice sorgente.

Eseguibile: altro nome per indicare il codice oggetto pronto per essere eseguito.

Script: programma memorizzato in un file, solitamente destinato ad essere interpretato.

Programma: serie di istruzioni che specificano come effettuare un'elaborazione.

Algoritmo: processo generale usato per risolvere una particolare categoria di problemi.

Bug: errore in un programma (detto anche "baco").

Debug: processo di ricerca e di rimozione di ciascuno dei tre tipi di errori di programmazione.

Sintassi: struttura di un programma.

Errore di sintassi: errore in un programma che rende impossibile la continuazione dell'analisi del codice (il programma non può quindi essere interpretato interamente o compilato).

Errore in esecuzione: errore che non è riconoscibile finché il programma non è stato eseguito e che impedisce la continuazione della sua esecuzione.

Eccezione, errore runtime: altri nomi per indicare un errore in esecuzione.

Errore di semantica: errore nel programma che fa ottenere risultati diversi da quanto ci si aspettava.

Semantica: significato di un programma.

Linguaggio naturale: ognuno dei linguaggi parlati evoluti nel tempo.

Linguaggio formale: ognuno dei linguaggi che sono stati progettati per scopi specifici, quali la rappresentazione di idee matematiche o programmi per computer (tutti i linguaggi per computer sono linguaggi formali).

Token: uno degli elementi di base della struttura sintattica di un programma analogo alla parola nei linguaggi naturali.

Parsing: esame e analisi della struttura sintattica di un programma.

Istruzione di stampa: istruzione che ordina all'interprete Python di scrivere un valore sullo schermo.

WIKIBOOKS

Python/Versione stampabile

Wikibooks, manuali e libri di testo liberi.
< Python

Indice

Introduzione

Indice del libro

Filosofia

Python è un linguaggio multi-paradigma. Infatti permette in modo agevole di scrivere programmi seguendo il paradigma object oriented, oppure la programmazione strutturata, oppure la programmazione funzionale. Il controllo dei tipi viene fatto a runtime (dynamic typing) e usa un garbage collector per la gestione automatica della memoria.

Python ha qualche similarità con Perl, ma i suoi progettisti hanno scelto la via di una sintassi molto più essenziale e uniforme, con l'obiettivo di aumentare la leggibilità del codice. Come il Perl spesso è classificato *linguaggio di scripting*, ma pur essendo utile per scrivere script di sistema (in alternativa ad esempio a bash), la grande quantità di librerie disponibili e la facilità con cui questo linguaggio permette di scrivere software modulare favoriscono anche lo sviluppo di applicazioni molto complesse.

Contenitori standard

Python ha una serie di tipi contenitori come ad esempio liste, tuple e dizionari. Liste, tuple e stringhe sono *sequenze* e condividono la maggior parte dei metodi: si può iterare sui caratteri di una stringa con la stessa facilità con cui lo si può fare sugli elementi di una lista. Le liste sono array estendibili, invece le tuple sono array immutabili di lunghezza prefissata.

Altri contenitori di grande utilità sono i *dizionari*, conosciuti in altri contesti con il nome di hash table oppure array associativi. Come chiavi dei dizionari possono essere usati solo oggetti immutabili, in modo che in ogni caso sia preservata la consistenza, invece come valori associati alla chiave vanno bene oggetti arbitrari.

Organizzazione ad oggetti

Il sistema dei tipi Python è ben integrato con il sistema delle classi. Anche se i tipi base non sono precisamente classi, una classe può ereditare da essi. In questo modo è possibile estendere stringhe, dizionari, ... o perfino gli interi. L'ereditarietà multipla è supportata.

Vengono supportate anche funzionalità estensive di introspezione sui tipi e sulle classi. I tipi e le classi sono a loro volta oggetti che possono essere esplorati e confrontati. Gli attributi sono gestiti in un dizionario..

Altre caratteristiche

Come il Lisp e a differenza del Perl, l'interprete Python supporta anche un modo d'uso interattivo attraverso il quale è possibile inserire codice direttamente da un terminale, vedendo immediatamente il risultato. Questo è un bel vantaggio per chi sta imparando il linguaggio, ma anche per gli sviluppatori esperti: brevi tratti di codice possono essere provati in modo interattivo prima di essere integrati nel programma principale. Python dispone anche di un framework per lo unit testing che permette di creare serie esaustive di test.

Sintassi

Indice del libro

Python è stato progettato in modo da essere altamente leggibile. Visivamente si presenta in modo molto semplice e ha pochi costrutti sintattici rispetto a molti altri linguaggi strutturati come C, Perl o Pascal.

Per esempio, Python ha solo due forme di ciclo:

- `for`, che cicla sugli elementi di una sequenza o su di un iteratore (come il `foreach` del Perl);
- `while`, che cicla fin tanto che l'espressione booleana indicata risulta vera.

In sostanza manca dei cicli in stile C `for`, `do...while`, oppure di un `until` in stile Perl, ma tutti questi naturalmente possono essere espressi con dei semplici equivalenti. Allo stesso modo ha solamente il costrutto `if...elif...else` per le scelte condizionate -- niente `switch` oppure `goto`.

Uso degli spazi

Una cosa inusuale del Python è il metodo che usa per delimitare i blocchi di programma, che lo rende unico fra tutti i linguaggi più diffusi.

Nei linguaggi derivati dall'ALGOL -- come ad esempio Pascal, C e Perl -- i blocchi di codice sono indicati con le parentesi oppure con parole chiave. (Il C ed il Perl usano `{ }`; il Pascal usa `begin` ed `end`.) In questi linguaggi è solo una convenzione degli sviluppatori il fatto di indentare il codice interno ad un blocco, per metterlo in evidenza rispetto al codice circostante.

Python, invece, prende a prestito una caratteristica dal meno noto linguaggio di programmazione Occam -- invece di usare parentesi o parole chiavi, usa l'indentazione stessa per indicare i blocchi nidificati. Di seguito un esempio per chiarire questo. La versione C e Python di funzioni che fanno la stessa cosa -- calcolare il fattoriale di un intero:

L'indentazione in Python è obbligatoria, un'indentazione errata solleva un'eccezione di tipo SyntaxError. L'indentazione deve essere effettuata solo per mezzo di spazi o tabulazioni (TAB), usarli entrambi in uno stesso programma viene considerato errore.

Fattoriale in C:

```c
int fattoriale(int x) {
    if (x == 0) {
        return(1);
    } else {
        return(x * fattoriale(x-1));
    }
}
```

Fattoriale in Python:

```
def fattoriale(x):
    if x == 0:
        return 1
    else:
        return x * fattoriale(x-1)
```

All'inizio questo modo di indicare i blocchi può confondere le idee a chi viene da altri linguaggi, ma poi si rivela molto vantaggioso, perché risulta molto conciso e obbliga a scrivere sorgenti indentati correttamente, aumentando alquanto la leggibilità del codice quando passa di mano in mano.

Programmazione funzionale

Come detto sopra, un altro punto di forza del Python è la disponibilità di elementi che facilitano la programmazione funzionale. Come ci si può aspettare, questo rende ancora più comodo operare con liste o altri tipi contenitore. Uno di questi costrutti è stato preso dal linguaggio funzionale Haskell e consente il "riempimento" di una lista, come possiamo vedere nel seguente esempio in cui vengono calcolate le prime cinque potenze di due:

```
numeri = [1, 2, 3, 4, 5]
potenze_di_due = [ 2 ** n for n in numeri ]
```

Dal momento che Python permette di avere funzioni come argomenti, è anche possibile avere costrutti funzionali più sottili, come ad esempio la *continuation (http://www.ps.uni-sb.de/~duchier/python/continuations.html)*.

In Python esiste la parola chiave `lambda`, ma i blocchi `lambda` possono contenere solo espressioni, non statement. Non sono quindi il modo più generale per restituire una funzione. Si può usare invece la seguente tecnica, che restituisce una funzione il cui nome è definito in uno scope locale.

```
def add_and_print_maker(x):
    def temp(y):
        print "%d + %d = %d" % (x, y, x+y)
    return temp
```

Gestione delle eccezioni

Python supporta e usa estensivamente la gestione delle eccezioni come un mezzo per controllare la presenza di eventuali condizioni di errore. Addirittura è possibile intercettare l'eccezione generata da un errore di sintassi (Syntax Error)!

Le eccezioni permettono un controllo degli errori più conciso ed affidabile rispetto a molti altri modi possibili usati in genere per segnalare errori o situazioni anomale. Le eccezioni sono thread-safe; non sovraccaricano il codice sorgente come fanno invece in C i controlli sui valori di errore ritornati, e inoltre possono facilmente propagarsi verso l'alto nello stack delle chiamate a funzione quando un errore deve venire segnalato ad un livello più alto del programma.

Il modo di fare frequente consiste, invece che fare controlli preventivi, nell'eseguire direttamente l'azione desiderata e catturare invece le eventuali eccezioni che si verificassero.

Python consente la manipolazione delle eccezioni tramite i blocchi try ... except Le eccezioni sono di tipi diversi, generalmente quando si verifica un'eccezione l'interprete Python lo segnala indicando il file in cui si è verificata l'eccezione, l'eventuale riga di codice, il tipo di eccezione e una breve descrizione di essa.

Cercare di dividere un numero per zero solleva un'eccezione di tipo ZeroDivisionError.

```
def Dividi(a,b):
    try:
        print a/b
    except ZeroDivisionError:
        print "Errore! Si sta cercando di dividere per 0"
```

Per prima cosa viene eseguito il blocco try, se non viene sollevata nessuna eccezione il blocco except viene saltato e l'esecuzione di try terminata. Se invece durante l'esecuzione il blocco try incontra un'eccezione viene interrotto e se l'eccezione corrisponde a quella citata nel blocco except, allora l'esecuzione passa al blocco except.

Tipi di Dati

Indice del libro

Tipi di dato

I tipi di dato messi a disposizione da Python sono i seguenti:

- Semplici:
  - interi (**int**),
  - interi lunghi (**long**),
  - numeri in virgola mobile (**float**)
  - numeri complessi (**complex**),
  - valori booleani (**bool**)
- Sequenze:
  - immutabili:
    - stringhe (**str**),
    - tuple (**tuple**),
    - insiemi immutabili (**frozenset**)
  - mutabili:
    - liste (**list**),
    - dizionari (**dict**)
    - insiemi (**set**)
    - file (**file**)

Alcune osservazioni:

- Gli interi lunghi sono arbitrariamente grandi.
- I valori booleani sono: ***True*** e ***False***.
- Per motivi di efficienza e di architettura del linguaggio le stringhe sono immutabili, cosicché qualsiasi operazione che in qualche modo potrebbe alterare una stringa (come ad esempio la sostituzione di un carattere) restituirà invece una nuova stringa.
- Le liste sono degli *array* dinamici che possono contenere dati di qualunque tipo.
- Le *tuple* sono come le liste ma sono immutabili.
- I dizionari contengono coppie di oggetti ***chiave***-***valore***; possono essere visti come degli array indicizzati invece che da interi, da un qualunque oggetto immutabile. La chiave deve infatti essere un oggetto immutabile.
- Gli insiemi sono contenitori non ordinati di oggetti non ripetuti.

Essendo Python a tipizzazione dinamica, tutte le variabili sono in realtà semplici puntatori ad oggetto (reference), sono gli oggetti invece ad essere dotati di tipo. Ad esempio un identificatore collegato ad un intero, un istante dopo può essere collegato ad una stringa o ad un array.

In Python c'è un forte controllo dei tipi a *runtime*. Si ha conversione implicita solo per i tipi numerici, per cui si può ad esempio moltiplicare un numero complesso per un intero, ma non c'è conversione implicita tra altri tipi, ad es. numeri e stringhe.

Esempio

Nel seguente programma ci sono delle funzioni che operano conversioni di tipo.

```
def elementidiversi(sequenza):
    """Restituisce il numero di elementi diversi contenuti in sequenza."""
    return len(set(sequenza))

def cifre(numero):
    """Restituisce la lista delle cifre di un numero."""
    return [int(c) for c in str(numero)]

def divisibilepertre(num):
    """Implementa il criterio di divisibilità per 3."""
    return sum(cifre(num)) % 3 == 0

def cambiabase(n, b):
    """Restituisce una stringa che rappresenta n in base b."""
    if n==0:
        return ""
    else:
        n, r = divmod(n, b)
        return cambiabase(n, b) + str(r)

def test():
    lista=[1, 3, 2, 1, 1, 3, 4, 2, 3]
    print("Nella lista %s ci sono %s elementi diversi\n" %
          (lista, elementidiversi(lista)))
    stringa="ambarabacicicoco"
    print("Nella stringa %s ci sono %s elementi diversi\n" %
          (stringa, elementidiversi(stringa)))
    n=65127842
    print("Le cifre di %s sono contenute nella lista: %s\n" %
          (n, cifre(n)))
    numeri=(6327432954857463849473273838, 75846585464325626511111111156)
    for n in numeri:
        print("%s è divisibile per 3: %s\n" % (n, divisibilepertre(n)))
    n=128
    for base in (2, 3, 4):
        print("%s in base %s è: %s" % (n, base, cambiabase(n, base)))


test()
```

Numeri

Indice del libro

Python, mette a disposizione 4 tipi di numeri: **interi**, **interi lunghi**, **in virgola mobile**, **complessi**.

Numeri interi

In **Python 2**, i numeri interi possono essere

- `int`: compresi nel range da `- sys.maxint - 1` a `sys.maxint` (il valore effettivo dipende dal sistema)
- `long`: dimensione arbitraria

Quando il risultato di un calcolo tra `int` esce dall'intervallo, il numero viene automaticamente convertito in intero lungo. Per esempio, `sys.maxint+1` restituisce un `long`.

In **Python 3**, esiste un solo tipo intero a precisione arbitraria, `int`.

```
>>> 2**30
1073741824
>>> 2**100
1267650600228229401496703205376L
```

Divisione

È stato modificato il comportamento di Python nel passaggio dalla versione 2.x alla versione 3.x. Nella versione 2.x il quoziente di due numeri interi è un numero intero:

```
>>> 7/2
3
```

Nella versione 3.x il quoziente di due numeri interi è un numero in virgola mobile:

```
>>> 7/2
3.5
>>> 6/2
3.0
```

Per fare in modo che anche Python 2.x si comporti come Python 3.x basta dare il comando:

```
from __future__ import division
```

A questo punto anche Python 2.x si comporta come Python 3.x:

```
>>> 7/2
3.5
```

```
>>> 6/2
3.0
```

Per ottenere la divisione intera in Python 3.x si usa l'operatore //:

```
>>> 7//2
3
```

Numeri interi lunghi

I numeri interi lunghi, **long**, non hanno limiti inferiori o superiori, o meglio i loro limiti derivano dalla quantità di memoria presente nella macchina.

Numeri in virgola mobile

I numeri in virgola mobile **float** vanno da -1.797693134862316e308 a 1.797693134862316e308.

Nell'usare i numeri in virgola mobile dobbiamo tenere presente che, per questioni di efficienza, sono memorizzati in base 2. Nel cambiare la base di un numero, si possono introdurre delle approssimazioni, infatti un numero *tranquillo* in base 10 ad esempio: 0.3 (tre decimi) diventa un numero *periodico* se rappresentato in base 2. Possiamo vedere come è rappresentato 0.3 internamente con il comando:

```
>>> 0.3
0.29999999999999999
```

Mentre l'istruzione print ci nasconde i dettagli interni e ci dà un risultato più familiare.

```
>>> print 0.3
0.3
```

Si deve tener conto di queste osservazioni se vogliamo usare un ciclo comandato da una variabile **float**:

```
>>> f=0
>>> while f!=1.0:
    print f
    f+=0.1


0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1.1
1.2
1.3
...
```

Questo ciclo non termina! L'istruzione "corretta" è:

```
>>> f=0
>>> while f<=1.0:
    print f
    f+=0.1
```

Ma non è una buona idea comandare un ciclo con una variabile **float** (provate a sostituire "<=" con "<"...). Bisogna sempre considerare i numeri in virgola mobile come numeri approssimati. La funzione **repr** ci mostra la rappresentazione interna del numero e ci dà l'idea di cosa succede dentro la macchina:

```
>>> f=0
>>> while f<=1.0:
    print repr(f)
    f+=0.1


0
0.10000000000000001
0.20000000000000001
0.30000000000000004
0.40000000000000002
0.5
0.59999999999999998
0.69999999999999996
0.79999999999999993
0.89999999999999991
0.99999999999999989
```

Numeri complessi

I numeri complessi (**complex**) sono numeri nella forma a+bj dove a e b sono numeri in virgola mobile.

I numeri complessi hanno due attributi: **imag** e **real** che sono i riferimenti alla parte reale e al coefficiente immaginario del numero e un metodo che restituisce il coniugato del numero: **conjugate** :

```
>>> c0=(5+6j)
>>> c0
(5+6j)
>>> c0.imag
6.0
>>> c0.real
5.0
>>> c0.conjugate()
(5-6j)
```

Operazioni

Si possono scrivere espressioni utilizzando diversi operatori, la seguente tabella li riassume:

Operazione	Operatore	Int	Float	Complex
addizione	x+y	3+8 --> 11	3.+8. --> 11.	(3+5j)+(8-7j) --> (11-2j)
sottrazione	x-y	3-8 --> -5	3.-8. --> -5.	(3+5j)-(8-7j) --> (-5+12j)
moltiplicazione	x*y	3*8 --> -5	3.*8. --> -5.	(3+5j)*(8-7j) --> (59+19j)
divisione	x/y	3/8 --> 0	3./8. --> 0.375	(3+5j)/(8-7j) --> (-0.097345+0.5398230j)
potenza	x**y	3**8 -->	3.**8. -->	(3+5j)**8 --> (-506864+1236480j)

		6561	6561.	
opposto	-x	-3 --> -3	-3. --> -3.	-(3+5j) --> (-3-5j)
divisione intera	x//y	8//3 --> 2	8.//3. --> 2.	(3+5j)//(8-7j) --> (-1+0j)
modulo	x%y	8%3 --> 2	8.%3. --> 2.	(3+5j)%(8-7j) --> (11-2j)
uguale	x==y	3==5 --> False	3.==5. --> False	(3+5j)==(8-7j) --> False
diverso	x!=y	3!=5 --> True	3.!=5. --> True	(3+5j)!=(8-7j) --> True
minore	x<y	3<5 --> True	3.<5. --> True	
maggiore	x>y	3>5 --> False	3.>5. --> False	
minore o uguale	x<=y	3<=5 --> True	3.<=5. --> True	
maggiore o uguale	x>=y	3>=5 --> False	3.>=5. --> False	
bit and	x&y	12&5 --> 4		
bit or	$x\|y$	5 --> 13		
bit xor	x^y	12^5 --> 9		
scorrimento a sinistra	x<<y	19<<2 --> 76		
scorrimento a destra	x>>y	19>>2 --> 4		

Ci sono anche alcune funzioni applicate ai numeri:

Operazione	Funzione	Int	Float	Complex
valore assoluto	abs(x)	abs(-7) --> 7	abs(-7.) --> 7.	abs(3+5j) --> 5.8309518948453007
quoziente, modulo	divmod(x)	divmod(32, 5) --> (6, 2)	divmod(32., 5.) --> (6.0, 2.0)	divmod((3+5j),(8-7j)) --> ((-1+0j), (11-2j))
conversione in stringa	str(x)	str(32) --> '32'	str(3.2) --> '3.2'	str(3+5j) --> '(3+5j)'
rappresentazione interna	repr(x)	repr(32) --> '32'	repr(3.2) --> '3.2000000000000002'	repr(3+5j) --> '(3+5j)')
conv. in intero	int(x)	int(37) --> 37	int(3.7) --> 3	
conv. in lungo	long(x)	repr(37) --> 37L	long(3.7) --> 3L	
conv. in virgola mobile	float(x)	float(32) --> 32.0	float(3.2) --> 3.2000000000000002	
conv. in ottale	oct(x)	oct(4354) --> '010402'		
conv. in esadecimale	hex(x)	hex(543543) --> '0x84b37'		

La funzione **int** accetta un secondo argomento che permette di interpretare la stringa passata come primo argomento come numero scritto in una determinata base:

```
>>> int('213', 4)
39
>>> int('213', 10)
213
>>> int('213', 12)
303
>>> int('213', 16)
531
```

La libreria math

La libreria '*math* mette a disposizione molte funzioni matematiche che non sono interne al linguaggio Python:

Funzione	Significato
acos(x)	restituisce l'arco coseno (misurato in radianti) di x
asin(x)	restituisce l'arco seno (misurato in radianti) di x
atan(x)	restituisce l'arco tangente (misurato in radianti) di x
atan2(y, x)	restituisce l'arco tangente (misurato in radianti) di y/x. tiene conto dei segni di x e di y
ceil(x)	restituisce l'approssimazione per eccesso di x come float
cos(x)	restituisce il coseno di x (misurato in radianti)
cosh(x)	restituisce il coseno iperbolico di x
degree(x)	converte l'angolo x da radianti in gradi
exp(x)	restituisce e elevato alla x
fabs(x)	restituisce il valore assoluto di x
floor(x)	restituisce l'approssimazione per difetto di x come float
fmod(x, y)	restituisce il resto della divisione tra x e y
frexp(x)	restituisce la mantissa e l'esponente di x
hypot(x, y)	restituisce la distanza euclidea sqrt(x*x+y*y)
ldexp(x, i)	restituisce x*(2**i)
log(x[, base])	restituisce il logaritmo di x nella base base
log10(x)	restituisce il logaritmo di x nella base 10
modf(x)	restituisce la parte frazionaria e la parte intera di x
pow(x, y)	restituisce la potenza x**y
radians(x)	converte l'angolo x da gradi in radianti
sin(x)	restituisce il seno di x (misurato in radianti)
sinh(x)	restituisce il seno iperbolico di x
sqrt(x)	restituisce la radice quadrata di x
tan(x)	restituisce la tangente di x (misurato in radianti)
tanh(x)	restituisce la tangente iperbolica di x

Oltre a queste funzioni, math fornisce due costanti:

Funzione	Significato
e	2.7182818284590451
pi	3.1415926535897931

Sequenze

Indice del libro

Tra i dati messi a disposizione da Python, alcuni hanno delle caratteristiche in comune: sono le **sequenze**. Le sequenze sono dei contenitori di oggetti che mantengono l'ordine. Sono sequenze le **liste**, le **tuple** e le **stringhe**.

Liste

Una lista è un contenitore ordinato di oggetti Python. Può essere costruita usando le parentesi quadre:

```
lista0=['a', 34, ['s', 47]]      # ['a', 34, ['s', 47]]
```

o il costruttore **list**:

```
lista1=list("delta")             # ['d', 'e', 'l', 't', 'a']
```

A volte può essere utile costruire una lista vuota da riempire in seguito:

```
lista2=[]
```

oppure

```
lista3=list()
```

La funzione **range(<n>)** restituisce una lista formata dai primi <n> numeri naturali:

```
lista4=range(10)                 # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Si può osservare qui che Python restituisce una lista formata effettivamente da 10 elementi da zero compreso a dieci escluso.

Python mette a disposizione un comodo metodo per costruire liste, la 'list comprehension'. Se ad esempio voglio una lista con i quadrati dei primi 10 numeri naturali posso scrivere:

```
lista6=[n*n for n in range(10)]  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Nella costruzione di liste si può aggiungere anche una condizione. Se ad es. volessi i quadrati dei soli numeri pari:

```
lista7=[n*n for n in range(10) if n % 2 == 0]  # [0, 4, 16, 36, 64]
```

Le liste sono oggetti modificabili, quindi è possibile inserire, cancellare o cambiare gli oggetti contenuti in una lista:

```
    lista=['Nel', 0.5, 'del', 'cammin', 'nostra', 'pippo', 'vita']
    print lista                    # ['Nel', 0.5, 'del', 'cammin', 'nostra', 'pippo', 'vita']
    lista[1]='mezzo'
    print lista                    # ['Nel', 'mezzo', 'del', 'cammin', 'nostra', 'pippo',
'vita']
    del(lista[5])
    print lista                    # ['Nel', 'mezzo', 'del', 'cammin', 'nostra', 'vita']
    lista.insert(4, 'di')
    print lista                    # ['Nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita']
```

Il metodo più utilizzato per popolare liste, dopo la *list comprehension* è l'uso del metodo *append*:

```
lista=[]
for i in range(3):
    lista .append(raw_input("Stringa n° %s: " % i))
print lista
['pippo', 'pluto', 'paperoga']
```

Tutti i metodi delle liste si possono ottenere con l'istruzione:

```
>>> help(list)
```

Tuple

Le tuple sono contenitori ordinati di oggetti Python. A differenza delle lista non sono modificabili, quindi una volta costruita una tupla non si può cancellare, cambiare o inserire un elemento. Può essere costruita usando le parentesi tonde:

```
tupla0=('a', 34, ['s', 47])      # ('a', 34, ['s', 47])
```

o il costruttore **tuple**:

```
tupla1=tuple("delta")            # ('d', 'e', 'l', 't', 'a')
```

Una tupla vuota risulta piuttosto inutile dato che, essendo immutabile, non può essere popolata.

La scelta di usare lo stesso simbolo per le parentesi delle espressioni e per indicare le tuple, ha portato ad un problema di ambiguità. Con l'espressione:

```
a=(6)                            # 6
```

potremmo voler associare ad a il risultato dell'espressione *(6)* o la tupla che contiene il solo elemento *6*. GVR ha dato la precedenza all'espressione. Per realizzare una tupla formata da un solo elemento dobbiamo aggiungere una virgola:

```
a=(6,)                           # (6,)
```

Tutti i metodi delle tuple si possono ottenere con l'istruzione:

```
>>> help(tuple)
```

Stringhe

Le stringhe sono sequenze di caratteri. Anche le stringhe non sono modificabili, quindi una volta costruita una stringa non la si può modificare. Può essere costruita usando gli apici, singoli o doppi in diversi modi:

```
stringa0="foo"
stringa1='bar'
stringa2="""altro modo"""
stringa3='''e infine'''
```

Le ultime due stringhe possono essere scritte anche su più righe. La funzione **str** permette di convertire altri oggetti in stringhe:

```
stringa4=str(45.3)                          # '45.3'
```

Altro modo per costruire stringhe componendo diversi elementi:

```
stringa5="sqrt(%s^2+%s^2)=%s" % (3, 4, 5)      # 'sqrt(3^2+4^2)=5'
```

I simboli *%s*, presenti nella stringa, sono dei *segnaposto* che vengono sostituiti, ordinatamente, dagli elementi presenti nella tupla che segue l'operatore *%*.

Tutti i metodi delle stringhe si possono ottenere con l'istruzione:

```
>>> help(str)
```

Scorrere le sequenze

È possibile scorrere tutti gli elementi di una sequenza usando un iteratore. Python mette a disposizione molti iteratori, ma il più usato è: **for**.

La sintassi è:

```
for <variabile> in <sequenza>:
    <istruzioni>
```

un semplice esempio di uso dell'iteratore **for** è:

```
def stampa_in_verticale(sequenza):
    """Visualizza gli elementi di una sequenza, uno sotto l'altro."""
    for elemento in sequenza:
        print elemento

def test():
    lista=['Et', 'telefono', 'casa']
    stampa_in_verticale(lista)
    print
    tupla=('M.Beri', 'Python', 'Apogeo', 2007, 7.50)
    stampa_in_verticale(tupla)
```

```
    print
    stringa="scala"
    stampa_in_verticale(stringa)
    print

test()
```

Anche i file di testo e altri oggetti Python si possono scorrere come le sequenze. Ad esempio se abbiamo un file di testo "commedia.txt" le istruzioni:

```
    testo=file("commedia.txt")
    stampa_in_verticale(testo)
    print
```

insieme alla funzione scritta sopra, permetteranno di stampare il suo contenuto una riga sotto l'altra.

Accedere ai singoli elementi

Si può accedere ai singoli elementi di una sequenza indicandone la posizione tra parentesi quadre. Si deve tenere presente che in Python ogni sequenza inizia dalla posizione zero quindi il primo elemento ha indice 0 il secondo 1 e così via:

```
stringa="Tre civette sul comò"
print stringa[0]                # 'T'
print stringa[1]                # 'r'
print stringa[3]                # ' '
```

Il meccanismo dell'indicizzazione può essere usato con tutte le sequenze quindi funziona allo stesso modo anche con liste e tuple:

```
lista=[26, "pippo", [13, 12, 2008], (6, 35)]
print lista[0]                  # 26
print lista[1]                  # 'pippo'
print lista[2][1]               # 12
```

L'ultima istruzione stampa l'elemento di posto 1 dell'elemento di posto 2 di lista. L'ultima istruzione indica il meccanismo per trattare matrici a enne dimensioni. Nel caso di una matrice a due dimensioni possiamo scrivere una funzione che ne visualizzi gli elementi:

```
def stampa_matrice(m):
    """Visualizza gli elementi di una matrice."""
    for riga in m:
        for e in riga:
            print e, "\t",
        print

def test():
    matrice=[[34, 64, 83],[38, 17, 25],[12, 18, 69]]
    stampa_matrice(matrice)
```

La stringa "\t" indica un tabulatore.

È anche possibile usare come indici numeri negativi, in questo caso -1 indica l'ultimo elemento, -2 il penultimo e così via:

```
lista=[26, "pippo", [13, 12, 2008], (6, 35)]
print lista[-1]                 # (6, 35)
print lista[-2][0]              # 13
```

Affettare sequenze (slicing)

Data una sequenza è possibile ricavarne un'altra che contenga parte dei suoi elementi. La sintassi assomiglia a quella dell'indicizzazione, si usano sempre parentesi quadrate, ma questa volta devono essere presenti due indici che individuano la fetta da copiare. I due indici devono essere separati dal simbolo ":". Nei prossimi esempi mi riferisco a una stringa ma funzionano allo stesso modo con qualunque sequenza, quindi anche con liste o tuple.

```
stringa="Tre civette sul comò"
s=stringa[4:11]
print s                             # civette
```

Coerentemente con il resto del linguaggio il primo indice è compreso nel risultato il secondo è escluso quindi l'istruzione precedente estrae la sottostringa dal carattere di posto 4 compreso al carattere di posto 11 escluso, 11 - 4 = 7 caratteri. Se devo estrarre i primi elementi di una sequenza posso sottintendere il primo indice:

```
s=stringa[:3]
print s                             # Tre
```

Allo stesso modo per estrarre gli ultimi elementi di una sequenza posso omettere il secondo indice:

```
s=stringa[16:]
print s                             # comò
```

Nell'ultimo caso sarebbe stato più comodo usare un indice negativo:

```
s=stringa[-4:]
print s                             # comò
```

La sintassi dell'affettamento può accettare un terzo argomento che indica un intervallo, così se voglio estrarre solo le lettere di posto pari della parola 'civette':

```
s=stringa[4:11:2]
print s                             # cvte
```

A volte è utile avere una copia di una lista che è un oggettio mutabile, la si può ottenere con l'istruzione:

```
lista=['uno', 'due', 'tre']
copia=lista[:]
print copia                             # ['uno', 'due', 'tre']
```

Non c'è bisogno di costruire copie di un oggetto immutabile, ma perché con le liste questo può servire? Consideriamo la seguente porzione di codice:

```
lista0=['uno', 'due', 'tre']
lista1=lista0
print lista0                            # ['uno', 'due', 'tre']
print lista1                            # ['uno', 'due', 'tre']
```

Sembrerebbe che il programma abbia realizzato la copia di lista0 e l'abbia assegnata alla variabile lista1. Ma le cose non sono così: il programma ha creato un nuovo identificatore che si riferisce sempre allo stesso oggetto lista. Ora se modifico un elemento della lista: e stampo di nuovo gli oggetti collegati a lista0 e lista1 ottengo:

```
lista0=lista0[1]='cinque'
print lista0                            # ['uno', 'cinque', 'tre']
print lista1                            # ['uno', 'cinque', 'tre']
```

Questo avviene perché ho modificato l'oggetto lista che è collegato sia a lista0 sia a lista1. Il seguente programma crea invece un nuovo oggetto lista che collega all'identificatore lista1:

```
lista0=['uno', 'due', 'tre']
lista1=lista0[:]
print lista0                            # ['uno', 'due', 'tre']
print lista1                            # ['uno', 'due', 'tre']
lista0=lista0[1]='cinque'
print lista0                            # ['uno', 'cinque', 'tre']
print lista1                            # ['uno', 'due', 'tre']
```

Il programma precedente ha costruito una sola lista collegata a due identificatori diversi, il secondo, dopo aver costruito una lista collegata all'identificatore lista0, ne costruisce un'altra, con gli stessi elementi e la collega all'identificatore lista1. In questo modo le modifiche effettuate sull'oggetto collegato all'identificatore lista0 non hanno effetto sull'oggetto collegato a lista1.

Dizionari

I dizionari sono degli *array* che vengono indicizzati non da numeri interi, ma da un qualunque oggetto *immutabile*. L'oggetto che serve da indice viene anche detto *chiave*, mentre l'oggetto associato a quella chiave viene detto *valore*.

Costruzione

Un dizionario può essere costruito usando le parentesi graffe:

```
>>> d={'a':45, 7:98, 497:list('pippo'), 3.14:'pi greco', (2, 7):'p'}
>>> print d
{'a': 45, 497: ['p', 'i', 'p', 'p', 'o'], 3.1400000000000001: 'pi greco', (2, 7): 'p', 7: 98}
```

Alcune osservazioni:

- Le chiavi possono essere interi, numeri in virgola mobile, stringhe, tuple o, comunque un qualunque oggetto immutabile.
- I valori possono essere oggetti qualunque.
- I dizionari non mantengono l'ordine di inserimento dei dati.
- 3.14 si è trasformato in: 3.1400000000000001. Ma questo non c'entra con i dizionari, c'entra con la rappresentazione dei numeri in basi diverse.

Accesso

Per ottenere un valore di un dizionario si utilizza la stessa sintassi di indicizzazione delle liste:

```
>>> print d[7]
98
>>> print d[3.14]
pi greco
>>> print d[(2, 7)]
p
```

Il metodo **keys()** permette di ottenere tutte le chiavi contenute in un dizionario e iterando con le chiavi è possibile estrarre tutti i valori corrispondenti:

```
>>> d={'a':7, 'b':9, 'c':3, 'd':0, 'e':9, 'f':6, 'g':5}
>>> print d
{'a': 7, 'c': 3, 'b': 9, 'e': 9, 'd': 0, 'g': 5, 'f': 6}
>>> for k in d.keys():
    print k, '-->', d[k]


a --> 7
c --> 3
b --> 9
e --> 9
```

```
d --> 0
g --> 5
f --> 6
```

Per ottenere gli elementi secondo un preciso ordine delle chiavi posso ordinarle prima di usarle nel ciclo **for**:

```
>>> chiavi=d.keys()
>>> chiavi.sort()
>>> for k in chiavi:
    print k, '-->', d[k]


a --> 7
b --> 9
c --> 3
d --> 0
e --> 9
f --> 6
g --> 5
```

Per modificare un valore basta utilizzare il suo indice:

```
>>> d['e']=44
>>>
>>> d
{'a': 7, 'c': 3, 'b': 9, 'e': 44, 'd': 0, 'g': 5, 'f': 6}
```

Per aggiungere un elemento si utilizza sempre la stessa sintassi:

```
>>> d['h']=4
>>> d
{'a': 7, 'c': 3, 'b': 9, 'e': 9, 'd': 0, 'g': 5, 'f': 6, 'h': 4}
```

Non sempre sappiamo quali elementi sono contenuti in un dizionario, quindi Python mette a disposizione due metodi che si comportano in modo diverso se la chiave è contenuta o non è contenuta nel dizionario. Infatti se tento di ottenere il valore associato ad una chiave inesistente ottengo un errore:

```
>>> print d['m']
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    print d['m']
KeyError: 'm'
```

Il metodo **get** invece funziona anche se la chiave non è presente:

```
>>> print d.get('m')
None
```

get ha anche un secondo argomento, opzionale che viene dato come risultato se la chiave non è presente:

```
>>> print d.get('g', 5)
5
```

```
>>> print d.get('m', 99)
99
```

Un esempio

I dizionari hanno anche molti altri metodi, ma questi permettono già di svolgere con poche linee di codice compiti piuttosto complessi. Possiamo scrivere una procedura che conti le occorrenze delle lettere presenti in una certa stringa:

```
>>> def contacaratteri(s):
    d={}
    for c in s:
        d[c]=d.get(c, 0)+1
    chiavi=d.keys()
    chiavi.sort()
    for k in chiavi:
        print k, '-->', d[k]


>>> contacaratteri('ambarabacicicoco')
a --> 4
b --> 2
c --> 4
i --> 2
m --> 1
o --> 2
r --> 1
```

Tutti i metodi dei dizionari si possono ottenere con l'istruzione:

```
>>> help(dict)
```

File

Indice del libro

Quando un programma termina, le informazioni contenute nei suoi dati vanno perse. Per conservare delle informazioni tra un'esecuzione e un'altra di un programma o per passare dei dati da un programma ad un altro, si usano i **file**. Un file è una sequenza di informazioni scritte su un supporto di memoria permanente. La *classe* **file** permette di *leggere*, *scrivere* e *modificare* file.

File di testo

Per scrivere stringhe in un file di testo basta aprire il file in scrittura e usare il suo metodo *write*. Terminato di scrivere nel file basta chiuderlo e questo è disponibile per altre applicazioni.

```
fo=file('prova.txt', 'w')
fo.write('Primo file di testo:\n\n')
fo.write('terza riga,\n')
fo.write('quarta riga riga,\n')
fo.write('fine del file.')
fo.close()
```

Leggere un file è altrettanto semplice: aprire il file in lettura, leggere le righe scorrendo il file come una sequenza:

```
for riga in file('prova.txt'):
    print riga,
```

Se vogliamo salvare in un file dei dati numerici possiamo tradurli in stringhe e salvarli come stringhe. Ad es. creiamo una matrice di numeri interi:

```
def randmat(dr, dc, massimo):
    import random
    m=[[random.randrange(massimo) for c in xrange(dc)] for r in xrange(dr)]
    return m

matrice=randmat(5, 4, 100)
```

possiamo salvare la matrice in un file usando le seguenti istruzioni:

```
def salvamat(matrice, nomefile):
    fo=file(nomefile, 'w')
    for riga in matrice:
        riga=[str(val) for val in riga]
        fo.write('\t'.join(riga)+'\n')
    fo.close()

salvamat(matrice, 'matrice.txt')
```

Una funzione che legga la matrice dal file e costruisca una lista di liste è:

```
def leggimat(nomefile):
    fi=file(nomefile)
    m=[[int(e) for e in riga.split('\t')] for riga in fi]
    fi.close()
    return m
```

E infine qualche riga di codice controlla che lettura e scrittura siano avvenute correttamente.

```
print matrice
matriceletta=leggimat('matrice.txt')
if matrice==matriceletta:
    print 'Slvataggio e lettura effettuati con successo'
else:
    print 'La matrice salvata e quella letta sono diverse!'
    print 'quella letta è:'
    print matriceletta
```

File binari

Un file binario contiene dati che non sono organizzati in stringhe di caratteri ma è semplicemente una sequenza di byte. Per interpretare un file binario bisogna conoscerne la struttura. Supponiamo di avere un file che contiene 5 stringhe pascal lunghe 20, 20, 1, 2 e 46 caratteri. Ogni stringa pascal ha inoltre un carattere, posto all'inizio, che indica la lunghezza della stringa. Il file dunque è organizzato in blocchi di 94 caratteri:

```
05 41 64 61 6d 69 41 64 61 74 00 00 00 00 00 37 .AdamiAdat.....7
4d 6f 64 69 66 07 41 72 6d 61 6e 64 6f 6e 61 67 Modif.Armandonag
67 69 75 6e 67 65 72 65 20 64 01 46 01 32 61 28 giungere d.F.2a(
45 44 43 44 43 41 2d 2d 2d 44 2d 44 43 2d 42 2d EDCDCA---D-DC-B-
2d 42 2d 42 44 2d 2d 2d 2d 2d 44 45 44 42 42 41 -B-BD-----DEDBBA
2d 2d 2d 2d 44 44 45 2d 0f 00 12 00 5d 00 08 42 ----DDE-....]..B
61 74 74 69 73 74 69 74 00 00 00 00 00 37 4d 6f attistit.....7Mo
64 69 66 05 4d 61 72 69 6f 6f 64 20 61 67 67 69 dif.Mariood aggi
75 6e 67 65 72 65 20 64 01 4d 01 32 61 28 44 44 ungere d.M.2a(DD
43 44 43 41 43 2d 2d 44 2d 44 43 2d 42 2d 2d 42 CDCAC--D-DC-B--B
2d 42 44 2d 2d 2d 2d 2d 44 45 44 42 42 43 2d 2d -BD-----DEDBBC--
2d 2d 44 44 45 2d 0f 00 11 00 5c 00 07 43 6f 62 --DDE-....\..Cob
65 6c 6c 6f 6f 74 00 00 00 00 00 37 4d 6f 64 69 elloot.....7Modi
66 07 47 69 6f 72 67 69 6f 20 61 67 67 69 75 6e f.Giorgio aggiun
67 65 72 65 20 64 01 4d 01 32 61 28 44 44 41 41 gere d.M.2a(DDAA
43 43 41 2d 42 44 2d 44 42 2d 42 2d 2d 42 42 45 CCA-BD-DB-B--BBE
45 2d 2d 42 2d 2d 2d 45 44 41 42 2d 2d 2d 2d 2d E--B---EDAB-----
2d 42 45 2d 0e 00 11 00 57 00                     -BE-....W.
```

Per estrarre i dati e tradurlo in un file di testo con i campi separati da virgole, devo leggere tutti i blocchi che compongono il file, interpretarli e scrivere i dati ottenuti in un file di testo. Per aprire in lettura il file binario di nome "CL2PI" si usa il comando:

```
fi=file("CL2PI", 'rb')
```

Per leggere un blocco di 94 caratteri sotto forma di sequenza di byte:

```
r=fi.read(94)
```

ora ci serve una funzione che legga i 5 campi presenti nel record e ce li restituisca come tupla di sstringhe:

```
def estrairecord(r):
    """Riceve una riga e la traduce in base alle specifiche."""
    return (estrai_stringa_pascal(r, 0),
            estrai_stringa_pascal(r, 21),
            estrai_stringa_pascal(r, 42),
            estrai_stringa_pascal(r, 44),
            estrai_stringa_pascal(r, 47),
           )
```

E una funzione che data la posizione del primo carattere di una stringa pascal, quello che ne fornisce la lunghezza, estragga la stringa:

```
def estrai_stringa_pascal(s, i):
    """Estrae dalla stringa s la sottostringa che inizia in i e che ha
       come primo byte la lunghezza, secondo la codifica del Pascal."""
    ls=ord(s[i])
    return s[i+1:i+ls+1]
```

Il programma può essere completato con la funzione che legge il file binario e scrive un file di testo:

```
def estraidati(nomefilein, nomefileout):
    """Traduce il file di ingresso e scrive il file di uscita."""
    fi=file(nomefilein, 'rb')
    fo=file(nomefileout, 'w')
    while True:
        r=fi.read(94)
        if r:
            print estrairecord(r), '\n'
            fo.write(','.join(estrairecord(r))+'\n')
        else:
            break
    fo.close()

estraidati('CL2PNI', 'CL2PNI.txt')
```

Selezione

Indice del libro

Python ha solo un'istruzione per implementare la selezione:

```
if <condizione>:
    <istruzioni>
[elif <condizione>:
     <istruzioni>]*
[else:
     <istruzioni>
```

Implementazione

Il costrutto fondamentale prevede una condizione da valutare e una serie di istruzioni da eseguire se essa risulta vera e un'altra serie di istruzioni da eseguire se invece risulta falsa. Ad esempio:

```
if eta>=18:
    print('sei maggiorenne')
else:
    print('sei minorenne')
```

È possibile l'uso del blocco `elif` (*else if*) che permette di valutare una seconda condizione nel caso quella iniziale risulti essere non vera. Ad esempio, possiamo aggiungere un ulteriore controllo nell'esempio precedente:

```
if eta>=18:
    print('sei maggiorenne')
elif eta<0:
    print('età non valida!')
else:
    print('sei minorenne')
```

Il numero di condizioni `elif` inseribili in un blocco `for` non è soggetto a limitazioni..

Esempi di uso

Alcuni esempi possono chiarire i casi di uso più comuni.

Taglio di un valore all'interno di un intervallo

```
def taglia(valore, minimo, massimo):
    """Restituisce valore se è compreso tra minimo e massimo
       se è minore di minimo restituisce minimo altrimenti massimo."""
    if valore<minimo:
        return minimo
    elif valore>massimo:
        return massimo
```

```
    else:
        return valore

n=int(raw_input('Scrivi un numero: '))
print('Il tuo numero tagliato tra 10 e 20 è: %s' % taglia(n, 10, 20))
```

Interprete di operazioni

Programmino che interpreta una stringa contenente un'operazione da eseguire scritta in lettere, sia in forma prefissa sia in forma infissa.

```
def somma(a, b):
    return float(a)+float(b)

def sottrai(a, b):
    return float(a)-float(b)

def moltiplica(a, b):
    return float(a)*float(b)

def dividi(a, b):
    return float(a)/float(b)

def eseguioperazione(stringa):
    """Cerca di eseguire l'operazione contenuta in stringa."""
    s0, s1, s2 = stringa.split(' ')
    if s0=='somma':
        return somma(s1, s2)
    elif s0=='sottrai':
        return sottrai(s1, s2)
    elif s0=='moltiplica':
        return moltiplica(s1, s2)
    elif s0=='dividi':
        return dividi(s1, s2)
    elif s1=='piu':
        return somma(s0, s2)
    elif s1=='meno':
        return sottrai(s0, s2)
    elif s1=='per':
        return moltiplica(s0, s2)
    elif s1=='diviso':
        return dividi(s0, s2)
    else:
        return 'non so eseguire questa operazione'

operazione='somma 3 67'
print operazione, '==>', eseguioperazione(operazione)
operazione='dividi 45 9'
print operazione, '==>', eseguioperazione(operazione)
operazione='45 diviso 9'
print operazione, '==>', eseguioperazione(operazione)
operazione='3 per 8'
print operazione, '==>', eseguioperazione(operazione)
operazione='accalappia 3 8'
print operazione, '==>', eseguioperazione(operazione)
```

Altro interprete di operazioni

Programmino che interpreta una stringa contenente un'operazione da eseguire scritta in lettere, sia in forma prefissa sia in forma infissa. Questa versione usa i dizionari che possono contenere, come valori, anche funzioni. Il risultato è più compatto, più flessibile, più efficiente, più... pythonico.

```
def somma(a, b):
    return float(a)+float(b)

def sottrai(a, b):
    return float(a)-float(b)

def moltiplica(a, b):
    return float(a)*float(b)

def dividi(a, b):
    return float(a)/float(b)

operazioni={'somma': somma, 'sottrai': sottrai,
            'moltiplica': moltiplica, 'dividi': dividi,
            'piu': somma, 'meno': sottrai,
            'per': moltiplica, 'diviso': dividi}

def eseguioperazione(stringa):
    s0, s1, s2 = stringa.split(' ')
    nomeoperazioni=operazioni.keys()
    if s0 in nomeoperazioni:
        return operazioni[s0](s1, s2)
    elif s1 in nomeoperazioni:
        return operazioni[s1](s0, s2)
    else:
        return 'non so eseguire questa operazione'

operazione='somma 3 67'
print operazione, '==>', eseguioperazione(operazione)
operazione='dividi 45 9'
print operazione, '==>', eseguioperazione(operazione)
operazione='45 diviso 9'
print operazione, '==>', eseguioperazione(operazione)
operazione='3 per 8'
print operazione, '==>', eseguioperazione(operazione)
operazione='accalappia 3 8'
print operazione, '==>', eseguioperazione(operazione)
```

Iterazione

Indice del libro

Python ha solo l'istruzione **while** per implementare l'iterazione nel senso di altri linguaggi come il Pascal o il C:

```
while <condizione>:
    <istruzioni>
[else:
     <istruzioni>]
```

Mentre l'istruzione **for** permette di iterare sugli elementi di una sequenza:

```
for <variabile> in <sequenza>:
    <istruzioni>
[else:
     <istruzioni>]
```

Entrambi i cicli ammettono due istruzioni:

- **continue** che permette di saltare le rimanenti istruzioni del ciclo passando all'iterazione successiva,
- **break** che termina il ciclo
- se è presente la clausola **else**, le sue istruzioni vengono eseguite se il ciclo termina normalmente, mentre non vengono eseguite se il ciclo termina a causa dell'istruzione **break**.

while

L'istruzione **while** permette di ripetere un blocco di *istruzioni* fin quando si mantiene vera una certa *condizione*.

Ad esempio se vogliamo elencare i primi 10 numeri naturali:

```
cont=0
while cont < 10:
    print cont
    cont+=1
```

Per elencare i primi 10 numeri pari:

```
cont=0
while cont < 10:
    print cont*2
    cont+=1
```

Ma in generale **while** va usato quando non si conosce il numero di volte che il ciclo andrà ripetuto. Ad es. nella seguente successione l'uso di while è adeguato:

```
n=7
while n!=1:
    print n
    if n % 2 == 0:
        n/=2
    else:
        n=n*3+1
```

Una delle tante implementazioni del **Crivello di Eratostene** usa un paio di cicli while:

```
def crivello(n):
    """Restituisce i numeri primi inferiori a n."""
    c=range(2, n+1)                          # crea una lista con tutti i numeri
    i=0                                      # inizializza l'indice
    n=len(c)                                 # calcola l'ultimo numero da processare
    nr=n**0.5                                # calcola l'ultimo numero da processare
    while i<nr:                              # fin quando l'indice è minore di questo numero
        if c[i]:                             # controlla che il numero puntato da i sia
diverso da 0
            j=i+1                            # in questo caso inizializza j al valore
successivo di i
        while j<n:                           # fin quando j è minore di n
            if c[j] % c[i] == 0: c[j]=0      # se l'elemento è divisibile per c[i] cambialo a
zero
            j+=1                             # incrementa j
        i+=1                                 # incrementa i
    c=[e for e in c if e]                    # crea una nuova lista fatta dagli elementi non
nulli di c
    return c                                 # restituisci questa lista che contiene solo
numeri primi
```

for

L'istruzione **for** è molto diversa dall'omonima istruzione del Pascal o del C, non è un'istruzione di ciclo, ma è un *iteratore* su una sequenza.

Se **a** è array di stringhe, e voglio visualizzarle una sotto l'altra, in Pascal scriverò all'incirca:

```
...
for i:=1 to maxarray do
    writeln(a[i]);
...
```

In Python:

```
for elemento in a:
    print elemento
```

Durante l'esecuzione del ciclo **for** la variabile **elemento** viene collegata ad ognuno degli elementi della sequenza **a**: non servono indici, non serve sapere quanti sono gli elementi della sequenza, quando sono finiti termina il ciclo.

Ad esempio una funzione che sommi tutti gli elementi di una lista di numeri potrebbe essere:

```
def somma(lista):
    s=0
    for num in lista:
        s+=num
    return s
```

```
>>> l=[1, 3, 5, 7, 9]
>>> print somma(l)
25
```

Ora se volessi la media dei numeri contenuti in una lista, posso scrivere la funzione:

```
def media(lista):
    return somma(lista)/float(len(lista))

>>> print media(l)
5.0
```

Peccato solo che che, per questioni di efficienza, Python abbia implementato tra le funzioni native la funzione **add** che produce lo stesso risultato della nostra funzione **somma**.

Se volessi implementare una ricerca sequenziale in una lista, potrei usare **for** nel seguente modo:

```
def contenuto(elemento, lista):
    result=True
    for e in lista:
        if e==elemento:
            break
    else:
        result=False
    return result

>>> print contenuto('pippo', l)
True
>>> print contenuto('Pippo', l)
False
```

Se volessi avere non una funzione che mi dice semplicemente se l'elemento è contenuto in una lista ma qual è la sua posizione, potrei combinare l'iteratore **for** con la funzione **enumerate** che restituisce la coppia: *posizione*, *elemento*:

```
def posizione(elemento, lista):
    for i, e in enumerate(lista):
        if e==elemento:
            result=i
            break
    else:
        result=-1
    return result

>>> print posizione('pluto', l)
1
>>> print posizione('pippo', l)
0
>>> print posizione('Pippo', l)
-1
```

Anche qui peccato che Guido ci abbia preceduti e sempre per questioni di efficienza abbia implementato l'istruzioine **in** e il metodo **index** che fanno all'incirca quello che fanno le nostre funzioni **contenuto** e **posizione**.

for: la trappola

L'iteratore **for** presenta un trabocchetto... Per capirlo, partiamo da un esempio. Abbiamo una lista che contiene vari numeri, vogliamo togliere tutti quelli pari:

```
def toglipari(lista):
    for e in lista:
        if e%2==0:
            lista.remove(e)

>>> num=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> toglipari(num)
>>> print num
[1, 3, 5, 7, 9]
```

Sembra funzionare perfettamente, ma non è così! Se tentiamo di togliere i numeri pari da quest'altra lista:

```
>>> num1=[0, 1, 2, 4, 5, 6, 7, 8, 9]
>>> toglipari(num1)
>>> num1
[1, 4, 5, 7, 9]
```

otteniamo un risultato sconcertante!

Cosa è successo, come mai a volte funziona e a volte no? Questi sono gli errori più insidiosi, appaiono solo a volte, e, tipicamente, quando fanno più danno.

Il fatto è che si deve assolutamente evitare di modificare la sequenza su cui l'iteratore **for** sta lavorando.

Nel primo caso (**num=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**) viene controllato il primo elemento di **num**, è pari e viene tolto. Ora il primo elemento è 1 e viene controllato il secondo, vale 2, è pari e viene tolto, ora il primo elemento è 1 e il secondo è 3, viene controllato il terzo che è pari e viene tolto...

Nel secondo caso (**num1=[0, 1, 2, 4, 5, 6, 7, 8, 9]**) viene controllato il primo elemento di **num1**, è pari e viene tolto. Ora il primo elemento è 1 e viene controllato il secondo, vale 2, è pari e viene tolto, ora il primo elemento è 1 e il secondo è 4, viene controllato il terzo che è 5, è dispari e viene lasciato, poi viene controllato il quarto elemento...

Regola fondamentale per non cadere nella trappola è: "mai modificare una lista mentre si sta iterando su di essa".

Allora, come si può realizzare l'operazione precedente? Basta iterare su una lista e modificarne un'altra, inizialmente identica. Ma come costruire una lista identica ad una già presente? Ci sono due costrutti Python che permettono di farlo:

```
lista1=lista[:]     # lista1 contiene tutti gli elementi di lista

lista2=list(lista)  # lista2 è una copia di lista
```

Tenendo conto di questo si può riscrivere il comando precedente e provare che funziona:

```
def toglipari(lista):
    for e in lista[:]:
        if e%2==0:
            lista.remove(e)
```

```
>>> num=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> toglipari(num)
>>> print num
[1, 3, 5, 7, 9]
>>> num1=[0, 1, 2, 4, 5, 6, 7, 8, 9]
>>> toglipari(num1)
>>> print num1
[1, 5, 7, 9]
>>> num2=[0, 1, 3, 2, 6, 4, 5, 2, 7, 8, 2, 9]
>>> toglipari(num2)
>>> print num2
[1, 3, 5, 7, 9]
```

Librerie

Indice del libro

Libreria standard

Python ha una vasta libreria standard, il che lo rende adatto a molti impieghi. Inoltre ai moduli della libreria standard se ne possono aggiungere altri scritti in C oppure Python per soddisfare le proprie esigenze particolari. Tra i moduli già disponibili ce ne sono per scrivere applicazioni web (sono supportati MIME, HTTP e tutti gli altri standard internet). Sono disponibili anche moduli per creare applicazioni con interfaccia grafica, per connettersi a database relazionali, per usare le espressioni regolari e per fare molte altre cose.

La libreria standard è uno dei punti forti di Python. Essa infatti è compatibile con tutte le piattaforme, ad eccezione di poche funzioni, segnalate chiaramente nella documentazione come specifiche di una piattaforma particolare. Grazie a questo generalmente anche programmi Python molto grossi possono funzionare su Linux, Mac, Windows e altre piattaforme senza dover essere modificati.

Strumenti

Indice del libro

Varie

- Esempi di codice e spiegazioni per illustare le differenze tra linguaggi tipizzati implicitamente e quelli tipizzati esplicitamente. In particolare Python e Java
- Esiste anche una implementazione Java del linguaggio Python che si chiama Jython.

Questo modulo è solo un abbozzo. Contribuisci (https://it.wikibooks.org/w/index.php?title=Python/Versione_stampabile&action=edit) a migliorarlo secondo le convenzioni di Wikibooks

Esempi

Questo modulo illustrerà degli esempi di programmi più o meno difficili scritti in Python.

Massimo comune divisore di due interi

Una funzione che riceve come argomenti due interi e restituisce il loro massimo comune divisore.

Versione ricorsiva usa la differenza tra i due numeri:

```
def mcd(a, b):
    """Restituisce il Massimo Comune Divisore tra a e b"""
    if a*b == 0: return 1
    if a==b:  return a
    elif a>b: return mcd(a-b, b)
    else:     return mcd(b-a, a)
```

Versione iterativa, usa il resto della divisione tra i due numeri:

```
def mcd(a, b):
    """Restituisce il Massimo Comune Divisore tra a e b"""
    while b:
        a, b = b, a%b
    return a
```

Minimo comune multiplo tra due numeri

Funzione che si basa sulla precedente funzione iterativa per calcolare il MCD tra due numeri. Adattata alla versione 3.1.1 di Python.

```
def mcd(a, b):
    while b:
        a, b = b, a%b
    return a

def mcm(a, b):
    """Restituisce il Minimo Comune Multiplo tra a e b"""
    return a // mcd(a, b) * b
```

Successione di Fibonacci

Questa funzione genera in sequenza gli elementi della successione di Fibonacci minori del valore passato come argomento. Tratto da diveintopython.org. Adattata alla versione di Python 3.1.1.

```
def fibonacci(max):
    a, b = 0, 1
    while a < max:
```

```
        yield a
        a, b = b, a+b

for n in fibonacci(1000):
    print(n)
```

Calcolo del codice fiscale

Questo programma chiede all'utente alcuni dati e calcola il suo codice fiscale (CF). Questo programma presuppone l'esistenza di un file `codicicomuni.txt` nella stessa cartella. Il file `codicicomuni.txt` deve contenere i dati nella forma:

```
<codice>,<nome del comune>,<provincia>
<codice>,<nome del comune>,<provincia>
...
```

Questo programma non effettua controlli sui dati inseriti dall'utente.

```
# -*- coding: utf-8  -*-
#!/usr/local/bin/python

###
# Costanti di visibilità globale
##
MESI = 'ABCDEHLMPRST'
CONSONANTI = 'BCDFGHJKLMNPQRSTVWXYZ'
VOCALI = 'AEIOU'
LETTERE = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
REGOLECONTROLLO = {
    'A':(0,1),   'B':(1,0),   'C':(2,5),   'D':(3,7),   'E':(4,9),
    'F':(5,13),  'G':(6,15),  'H':(7,17),  'I':(8,19),  'J':(9,21),
    'K':(10,2),  'L':(11,4),  'M':(12,18), 'N':(13,20), 'O':(14,11),
    'P':(15,3),  'Q':(16,6),  'R':(17,8),  'S':(18,12), 'T':(19,14),
    'U':(20,16), 'V':(21,10), 'W':(22,22), 'X':(23,25), 'Y':(24,24),
    'Z':(25,23),
    '0':(0,1),   '1':(1,0),   '2':(2,5),   '3':(3,7),   '4':(4,9),
    '5':(5,13),  '6':(6,15),  '7':(7,17),  '8':(8,19),  '9':(9,21)
                 }

def leggicodicicomuni(nomefile):
    """Restituisce un dizionario ricavato da un file di testo CVS."""
    cc={}
    for riga in file(nomefile):
        codice, citta, prov = riga.split(',')
        cc[citta] = codice
    return cc

CODICICOMUNI=leggicodicicomuni('codicicomuni.txt')

###
# Funzioni
##
def leggidati():
    """Legge i dati necessari come stringhe immesse dall'utente."""
    cognome = raw_input('Cognome: ')
    nome = raw_input('Nome: ')
    giornonascita = raw_input('Giorno di nascita: ')
    mesenascita = raw_input('Mese di nascita (numero): ')
    annonascita = raw_input(u"Anno di nascita: ")
    sesso = raw_input(u"Sesso (M/F): ")
    cittanascita = raw_input(u"Comune di nascita: ")
    return (cognome, nome, giornonascita, mesenascita, annonascita,
            sesso, cittanascita)

def trelettere(stringa,nome=True):
    """Ricava, da stringa, 3 lettere in base alla convenzione dei CF."""
```

```
    cons = [c for c in stringa if c in CONSONANTI]
    if nome and len(cons)>3:
        return [cons[0],cons[2],cons[3]]
    voc = [c for c in stringa if c in VOCALI]
    chars=cons+voc
    if len(chars)<3:
        chars+=['X', 'X']
    return chars[:3]

def datan(giorno, mese, anno, sesso):
    """Restituisce il campo data del CF."""
    chars = (list(anno[-2:]) + [MESI[int(mese)-1]])
    gn=int(giorno)
    if sesso=='F':
        gn+=40
    chars += list("%02d" % gn)
    return chars

def codicecontrollo(c):
    """Restituisce il codice di controllo, l'ultimo carattere del CF."""
    sommone = 0
    for i, car in enumerate(c):
        j = 1 - i % 2
        sommone += REGOLECONTROLLO[car][j]
    resto = sommone % 26
    return [LETTERE[resto]]

def codicefiscale(cognome, nome, giornonascita, mesenascita, annonascita,
                  sesso, cittanascita):
    """Restituisce il CF costruito sulla base degli argomenti."""
    nome=nome.upper()
    cognome=cognome.upper()
    sesso=sesso.upper()
    cittanascita = cittanascita.upper()
    chars = (trelettere(cognome,False) +
             trelettere(nome) +
             datan(giornonascita, mesenascita, annonascita, sesso) +
             list(CODICICOMUNI[cittanascita]))
    chars += codicecontrollo(chars)
    return ''.join(chars)

###
# Programma principale
##
cf = codicefiscale(*leggidati())
print "Il tuo codice fiscale e\': %s" % cf
```

Test di Primalità

Questa funzione restituisce il valore booleano True se il numero passato come argomento è primo, False se il numero non è primo. Adattata alla versione Python 3.1.1

```
def primo(numero):
    l = int(numero ** 0.5) + 1  # I fattori primi di un numero sono minori o uguali alla
radice del numero stesso
    for n in range(2, l):       # Esegue un ciclo for per vedere se l'argomento numero ha
qualche divisore
        if numero % n == 0:
            return False        # Se numero ha un divisore, viene restituito False
    return True                 # Altrimenti True
```

Ecco una versione più veloce, con Python 2.6.4:

```
def primo(n):
    if n == 2: return True  # Se n è uguale a 2, è primo
    if n % 2 == 0: return False  # Se n è pari, non è primo
    for x in xrange(3, int(n**0.5)+1, 2):  # Cerco in tutti i numeri da 3 alla radice
```

```
quadrata di n, saltando i numeri pari
        if n % x == 0:  # Se n è divisibile per una qualunque x, non è primo
            return False
    return True  # Se non è divisibile per nessuna x, allora è primo
```

Scomposizione in fattori primi di un intero

Questa funzione restituisce una stringa contenente i fattori primi di un numero n dato in input.

Adattata alla versione 3.1.1 di Python

Esempio:

```
>>> scomposizione(232132)
232132 = 1 * 2 * 2 * 131 * 443
```

Ecco la funzione:

```
def scomposizione(n):
    fattore = str(n) + " = 1"                   # Creazione della stringa fattore
    d = 2                                       # Assegnamento del valore 2 alla variabile d
    while n >= d:                               # Inizio del ciclo while
        if n % d == 0:                        # Se il numero n è divisibile per d: la stringa
fattore viene incrementata
            fattore += ' * ' + str(d)
            n /= d                          # Il valore di n viene diviso per d
        else:
            d = d + 1                       # Se n non è divisibile per d, il valore di d viene
incrementato di 1
    return fattore
```

Divisori di un numero

Questa funzione elenca tutti i divisori di un numero, compreso 1.

Adattata alla versione 3.1.1 di Python

```
def divisori(n):
    a = []
    for x in range(1, n + 1):
        if n % x == 0:
            a.append(int(x))
    print(a)
```

Qui un'altra versione, che utilizza le list comprehension ed è più veloce, perché controlla tutti i numeri fino a n/2, adattata a Python 2.6.4:

```
def divisori(n): return [c for c in xrange(1, int(n/2)+1) if n % c == 0]+[n]
```

Crivello di Eratostene

Di seguito l'implementazione del Crivello di Eratostene. Funzionamento:

1. Viene creata una lista num di interi, da 0 fino al numero n;
2. Si sostituisce l'elemento in posizione uno con uno zero;
3. Per ogni numero x da 0 alla radice quadrata di n si controlla che l'elemento in posizione num[x] sia diverso da 0;
4. Se lo è si cambiano tutti i suoi multipli con degli 0;
5. Si eliminano tutti gli 0 dalla lista;
6. Si ritorna la lista.

Adattato alla versione 2.6.4 di Python.

```
def Crivello(n):
    num = list(range(n+1))
    num[1] = 0
    for x in range(int(n**0.5)+1):
            if num[x] != 0:
                    num[2*x::x] = [0]*(int(n/x)-1)
    return list(filter(None, num))
```
