FABIO AIOLLI

Appunti di programmazione (scientifica) in

Python

SOCIETÀ EDITRICE
ESCULAPIO

FABIO AIOLLI

Appunti di programmazione (scientifica) in Python

SOCIETÀ EDITRICE
ESCULAPIO

ISBN 978-88-7488-678-4

Prima edizione: Dicembre 2013
Ristampa corretta: Novembre 2014

Responsabile produzione: Alessandro Parenti
Redazione: Giancarla Panigali e Carlotta Lenzi

SOCIETÀ EDITRICE ESCULAPIO

40131 Bologna - Via U. Terracini 30 - Tel. 051-63.40.113 - Fax 051-63.41.136
www.editrice-esculapio.it

Indice

1 Concetti introduttivi **9**

1.1 Algoritmi 9
1.2 Macchina di Turing 10
1.3 Linguaggi 11
1.4 Python 13

2 Panoramica su Python **15**

2.1 Console Python, toccata e fuga 15
2.2 Un mondo di oggetti 16
2.3 Espressioni e valutazione 17
2.4 Riferimenti ad oggetti 20
2.5 Scripting 22
2.6 Input e output 22
2.7 Importare moduli esterni 23

3 Tipi di base del Python **27**

3.1 Tipi per tutti i gusti 27
3.2 Classificazione dei tipi standard 28
3.3 Tipi Atomici 29
3.3.1 Numeri 29
3.3.2 Stringhe 32
3.3.3 Conversione tra numeri e stringhe 32
3.3.4 Set di caratteri 33
3.4 Contenitori 34
3.4.1 Tuple 34
3.4.2 Stringhe e tuple: operatori di formato 35
3.4.3 Liste 36
3.4.4 Liste e stringhe: split() e join() 38
3.5 Operazioni sulle sequenze 39
3.5.1 Assegnamento in linea 39
3.5.2 Concatenazione 39
3.5.3 Ripetizione 39
3.5.4 Indicizzazione 40
3.5.5 Slicing e striding 40
3.5.6 Modifica di sotto-sequenze 41
3.6 Dizionari 42
3.7 Insiemi 43
3.8 Operazioni su iterabili 44
3.8.1 Operatori su tipi iterabili 44
3.8.2 Funzioni su tipi iterabili 44

3.9 Descrittori di lista 46
3.10 Assegnamenti, operatori, metodi e funzioni 47

4 Controllo del flusso **53**

4.1 Selezione 53
4.2 Iterazione 54
4.3 Espressioni a valori booleani 54
4.3.1 Espressioni basiche 55
4.3.2 Identità 55
4.3.3 Comparazione 56
4.3.4 Appartenenza 56
4.3.5 Connettivi logici 57
4.3.6 Espressioni condizionali 58
4.4 Costrutto di selezione (if, elif, else) 59
4.5 Cicli (while) 61
4.6 Iteratori (for) 64
4.6.1 Iterare su un range di interi 64
4.6.2 Iterare per riferimenti 64
4.6.3 Iterare per indice e riferimenti 65
4.7 while o for? 66

5 Funzioni e ricorsione **71**

5.1 Funzioni 72
5.1.1 Chiamata di funzioni 73
5.1.2 Definizione di funzioni 73
5.1.3 Ritornare oggetti 74
5.1.4 Visibilità delle variabili 77
5.1.5 Funzioni come argomenti 77
5.1.6 Funzioni annidate 78
5.2 Funzioni ricorsive 80
5.2.1 Ricorsione in avanti e all'indietro 81
5.2.2 Ricorsione o non ricorsione? 82
5.3 Correttezza di funzioni ricorsive 85

6 Le classi **89**

6.1 Definizione di classe 89
6.1.1 Classi e oggetti istanza di classe 89
6.1.2 Attributi e metodi di classe 90
6.1.3 Programmare con le classi: il colore è servito 92
6.2 Ereditarietà 95
6.2.1 Il poker è servito 96
6.3 Metodi speciali e overloading degli operatori 98
6.3.1 La classe Ratio 100

7 Strutture dati ricorsive **105**

7.1 Liste concatenate . . . 105
7.1.1 Definizione della classe lista concatenata . . . 107
7.1.2 Funzioni utili per le liste concatenate . . . 107
7.1.3 Funzioni per la gestione di liste concatenate . . . 108
7.2 Alberi binari . . . 109
7.2.1 Definizione della classe albero binario . . . 110
7.2.2 Funzioni utili per alberi binari . . . 111
7.2.3 Visite . . . 112
7.2.4 Funzioni per la visualizzazione di alberi binari . . . 113

8 Algoritmi **119**

8.1 Complessità computazionale . . . 119
8.1.1 Complessità in tempo . . . 120
8.1.2 Complessità in spazio . . . 121
8.2 Algoritmi di ordinamento . . . 122
8.2.1 Selection Sort . . . 123
8.2.2 Insertion Sort . . . 124
8.2.3 Bubble Sort . . . 125
8.2.4 Merge Sort . . . 127
8.3 Algoritmi di ricerca per chiave . . . 129
8.3.1 Ricerca lineare . . . 130
8.3.2 Ricerca binaria . . . 130
8.3.3 Ricerca su liste concatenate . . . 131
8.3.4 Ricerca su alberi binari . . . 132
8.4 Algoritmo backtracking . . . 134
8.4.1 Sotto-insiemi su n elementi . . . 135
8.4.2 Permutazioni di n elementi . . . 136
8.4.3 Disposizioni di c elementi su n . . . 136

9 Progetti **139**

9.1 Il gioco del campo minato . . . 139
9.1.1 Regole del gioco . . . 139
9.1.2 Progettazione e implementazione . . . 139
9.2 Progetti di analisi numerica . . . 142
9.2.1 Algoritmi di ricerca degli zeri di una funzione . . . 143
9.2.2 Integrazione numerica . . . 144

A Il modulo random **145**

B Il modulo math **147**

C File e modulo sys **149**

C.1 File di testo . . . 149
C.2 Modulo sys . . . 150

Prefazione

Saper programmare non significa necessariamente conoscere un linguaggio di programmazione nei suoi dettagli. Al contrario, per essere un buon programmatore occorre apprendere pochi princìpi fondamentali allo scopo di poter esprimere con un algoritmo la soluzione di un problema specifico. E quali sono le qualità di un algoritmo per poterlo definire un buon algoritmo? Un buon algoritmo deve rispettare quella che io chiamo 'la regola delle tre e', ovvero deve essere *efficace* sul problema da risolvere, *efficiente*, e il più possibile *elegante*. La conversione dei passi dell'algoritmo in uno specifico linguaggio formale, detto anche linguaggio di programmazione, sarà poi un passo naturale che non richiederà un eccessivo sforzo se la parte progettuale è stata effettuata in modo consono.

La scelta di Python come linguaggio di riferimento in questo libro è dovuta a vari fattori, tra i quali:

1. siamo interessati principalmente alla programmazione scientifica e Python offre molte funzionalità per il calcolo scientifico;

2. Python è un linguaggio semplice, nel senso che i programmi scritti in Python risultano tipicamente brevi e auto-esplicativi;

3. Python si presta come primo linguaggio di programmazione in quanto non richiede l'apprendimento di tecnicalità tipiche di altri linguaggi di programmazione di livello più basso che potrebbero distrarre l'aspirante programmatore dagli aspetti veramente importanti della programmazione;

4. Python usa il paradigma della programmazione orientata agli oggetti (OOP), caratteristica che ha in comune con i più comuni linguaggi di programmazione avanzati. L'apprendimento dello stile di programmazione OOP costituisce quindi un buon bagaglio spendibile nel mondo del lavoro.

Le caratteristiche sopra citate rendono Python una scelta ottimale per uno studente, non necessariamente informatico, che voglia addentrarsi nello stimolante mondo della programmazione.

Il lettore potrà chiedersi perché la parola *scientifica* compaia tra parentesi nel titolo del libro. Ciò non è affatto casuale. Il motivo è che tale parola assume qui due possibili accezioni. Sicuramente ci riferiamo alla programmazione scientifica in quanto programmazione rivolta ad applicazioni scientifiche, ma ci riferiamo anche all'approccio alla programmazione suggerito in questo libro: un approccio scientifico, appunto.

Programmare con criterio significa combinare correttamente tecniche standard individualmente corrette allo scopo di creare programmi che facciano esattamente quello che si vuole, in modo efficiente e in ogni possibile situazione che può verificarsi. Possiamo riassumere il concetto con la seguente affermazione: se

esistono più modi per fare la stessa cosa, allora molto probabilmente ne esisterà uno che è migliore (o più corretto) rispetto agli altri.

Contenuti

In questo libro esploreremo vari aspetti della programmazione: la sintassi e l'uso del linguaggio Python, gli aspetti fondamentali della programmazione, le strutture dati e gli algoritmi e, infine, alcune esempi di applicazioni orientate ai giochi e alla scienza. Il percorso che seguiremo è il seguente:

- Sintassi del linguaggio Python
- Tipi di base di Python
- Costrutti per la programmazione imperativa: selezione e iterazione
- Funzioni e ricorsione
- Programmazione OOP (le classi)
- Strutture dati ricorsive
- Algoritmi notevoli

Il libro è corredato da molti esercizi risolti e commentati per problemi di complessità variabile. Allo scopo di ottimizzare l'apprendimento il lettore è caldamente invitato a ideare varianti dei problemi già risolti nel libro, o estensioni di questi, in modo da esplorare il più possibile la immensa varietà di problemi che si possono presentare e sfidare la propria capacità nel risolverli.

Prerequisiti

In realtà non esistono prerequisiti particolari per questo libro a parte un po' di pratica con l'uso dei computer e una conoscenza delle tecniche e dei formalismi di base della matematica. Il libro non tratta gli aspetti architetturali dei computer, i sistemi operativi, il web, ecc. Conoscenze in questi settori chiaramente non possono che aiutare nella lettura.

Ringraziamenti

Un sentito ringraziamento va a tutti i ragazzi che negli anni hanno offerto il loro aiuto come supporto alla didattica per i vari corsi di Programmazione. In ordine alfabetico: Lorenzo Barasti, Andrea Burattin, Elisa Caniato, Alessia Ceccato, Giovanni Da San Martino, Federico Di Palma, Michele Donini, Mirco Gelain. Un grazie ai miei colleghi Silvia Crafa, Ombretta Gaggi e Fabio Marcuzzi per il loro continuo incoraggiamento e i loro preziosi consigli.

Ma soprattutto alla mia famiglia,
– Fabio.

1

Concetti introduttivi

1.1 Algoritmi

Un algoritmo è definibile come l' *insieme completo delle regole che permettono la soluzione di un determinato problema*. Volendo dare una definizione operativa dello stesso concetto possiamo dire che un algoritmo è *una procedura effettiva che indica le istruzioni (passi) da eseguire per ottenere i risultati voluti a partire dai dati di cui si dispone.*

Potremmo fare molti esempi di istanze di algoritmi con cui abbiamo a che fare più o meno quotidianamente, i quali includono:

Cucina: ricetta per cucinare gli spaghetti, preparare una pizza ecc.;

Turismo: indicazioni per raggiungere una certa località o un albergo;

Fai-da-te: istruzioni di montaggio di un mobile;

Matematica: insieme di passi per verificare se un numero è dispari, pari, primo, ecc. Il calcolo del massimo comun divisore tra due numeri $MCD(a, b)$ mediante l' algoritmo di Euclide, o del minimo comune multiplo $mcm(a, b)$ ottenibile con la nota formula $mcm(a, b) = (ab)/MCD(a, b)$.

Descriviamo in maggior dettaglio l'algoritmo di Euclide per il calcolo del massimo comun divisore di una coppia di numeri naturali `a` e `b`. Senza perdita in generalità assumiamo che `a > b` (se questo non fosse il caso possiamo tranquillamente scambiarne i valori). L'algoritmo è il seguente:

1. Finché `b` rimane diverso da `0`
   (a) `t = b`
   (b) `b = a % b` (`b` assume il valore del resto della divisione tra `a` e `b`)
   (c) `a = t`
2. `MCD(a,b) = a`

Una caratteristica fondamentale che contraddistingue un algoritmo da una descrizione sommaria in linguaggio naturale è la non ambiguità. L'elaboratore (o esecutore) deve essere in grado di riconoscere senza ambiguità ed eseguire ogni singola istruzione dell'algoritmo. Come possiamo vedere nell'algoritmo del MCD, anche se espressa in un linguaggio non prettamente formale, ogni istruzione è ben comprensibile ad un operatore che abbia un minimo di conoscenze matematiche e che conosca le operazioni matematiche basiche (assegnamento, resto di una divisione, ecc.).

Proviamo quindi a dare un elenco più dettagliato delle caratteristiche necessarie di un algoritmo e del relativo esecutore.

L'esecutore deve avere le seguenti caratteristiche:

1. l'insieme di istruzioni deve essere di cardinalità finita;
2. il tempo di esecuzione di ogni istruzione deve essere finito;
3. l'elaboratore ha una memoria;
4. l'elaborazione procede per passi discreti;
5. non esiste limite alla lunghezza dei dati di ingresso;
6. non c'è un limite alla memoria disponibile.

Mentre, l'algoritmo deve essere:

1. deterministico
2. esprimibile con un numero finito di istruzioni;
3. le istruzioni devono essere eseguibili dall'elaboratore;
4. numero di passi di esecuzione può essere anche illimitato.

1.2 Macchina di Turing

La Macchina di Turing (introdotta nel 1936 da Alan Turing) è il modello di calcolo (esecutore) più semplice che si possa immaginare. La macchina opera su di un nastro che possiamo pensare come una sequenza di caselle nelle quali possono essere registrati simboli da un alfabeto ben determinato e finito; essa è anche dotata di una testina di lettura e scrittura con cui è in grado di effettuare operazioni di lettura e scrittura su una casella del nastro.

Il funzionamento della Macchina di Turing, nella versione più semplice, può essere schematizzato come segue:

- I dati iniziali (input) risiedono sul nastro

- La macchina possiede uno stato interno
- Ad ogni passo, dipendentemente dallo stato in cui si trova in quel momento e dal carattere letto, la macchina effettua tre azioni:
  - modifica il contenuto della casella
  - si sposta a destra o sinistra di una posizione
  - cambia il suo stato interno

Gli algoritmi per una Macchina di Turing sono dati in forma tabellare dove, per ogni coppia (stato, carattere letto), indichiamo *(i)* il simbolo da scrivere nella casella corrente, *(ii)* lo spostamento a destra o sinistra da effettuare e *(iii)* il nuovo stato interno.

Nonostante la sua evidente semplicità, è stato dimostrato che nessun altro modello di calcolo, dimostrazione matematica o macchina fisica (compresi i super computer), supera in termini di potenza di calcolo (ovvero quantità e tipo di elaborazioni possibili e funzioni calcolabili) tale modello formale.

1.3 Linguaggi

Il termine *linguaggio* è utilizzato in vari contesti. Per esempio parliamo di

- linguaggio naturale
- linguaggio della musica e della pittura
- linguaggio del corpo
- ecc.

I suddetti linguaggi sono per loro natura *ambigui* e questa caratteristica li rende non adatti per esprimere algoritmi. Viceversa, i *linguaggi formali o di programmazione* sono linguaggi non ambigui, ovvero

- le frasi generabili sono regolate da regole grammaticali precise (*sintassi*)
- alle frasi sintatticamente corrette del linguaggio viene associato un significato univoco (*semantica*)

I linguaggi formali possono essere classificati in base al loro livello di astrazione, dal livello più basso corrispondente ad un insieme di istruzioni di base per la macchina su cui operano (trasferimento di dati tra memoria e registri della CPU, operazioni aritmetico logiche basilari, e poco altro), fino a linguaggi di alto livello con una sintassi del tutto simile a quella del linguaggio naturale. Partendo dal livello più basso abbiamo:

- microprogrammi (firmware, codifica delle istruzioni)
- linguaggio macchina (codice binario)

- assembly (simbolico)
- linguaggi di medio livello (C,C++)
- linguaggi di alto livello (Java, Python, Pascal, ecc.)

Ogni costrutto reso disponibile in un linguaggio di livello più alto viene implementato da una o più istruzioni di un linguaggio al livello più basso. Quindi, le istruzioni assembly saranno implementate utilizzando funzionalità del linguaggio macchina che a sua volta sono codificate nel firmware della CPU. Python nella sua versione standard, per esempio, è implementato completamente in C.

I linguaggi di medio-alto livello si differenziano a loro volta per il paradigma di programmazione utilizzato tra

- linguaggi procedurali: Pascal, Fortran, Cobol, C, ...
- linguaggi funzionali: Lisp, Haskell, ...
- linguaggi dichiarativi o logici: Prolog, ...
- linguaggi orientati agli oggetti: C++, Java, Python, ...

Appositi software (compilatori) si occupano di tradurre le istruzioni scritte in questi linguaggi, il cosiddetto *codice sorgente*, nell'equivalente codice direttamente eseguibile dalla macchina, cosiddetto *codice eseguibile*. Python e Java sono in realtà linguaggi interpretati, vale a dire i comandi vengono tradotti ed eseguiti uno dopo l'altro da un software chiamato interprete. La Python Virtual Machine (PVM) e la Java Virtual Machine (JVM), gli interpreti per Python e Java, rispettivamente, compilano in ByteCode, un linguaggio intermedio indipendente dalla macchina. La presenza di una Virtual Machine rende i programmi scritti in questi due linguaggi altamente portabili nel senso che possono essere eseguiti direttamente su macchine diverse senza particolari problemi. D'altro canto, l'esecuzione di un programma scritto in un linguaggio interpretato risulta generalmente più lenta di un equivalente compilato.

I linguaggi di alto livello offrono diverse tipologie di costrutti a seconda del tipo di paradigma per il quale sono stati progettati.

Selezione, ovvero strutture di controllo decisionali del tipo: se una espressione booleana `EXP` è vera, allora esegui un blocco di istruzioni `IstruzioniV`, altrimenti esegui un altro blocco di istruzioni `IstruzioniF`;

Iterazione, ovvero strutture di controllo iterative del tipo: fintanto che una espressione booleana `EXP` è vera esegui un blocco di istruzioni `Istruzioni`;

Programmazione OOP, ovvero costrutti per la creazione di classi e meccanismi di overloading e ereditarietà tra le classi;

Gestione delle eccezioni, ovvero costrutti per la gestione delle eccezioni e degli errori, per esempio costrutti `try except` dove una parte di codice dedicata alla gestione di un eccezione viene eseguita quando una particolare condizione anomala viene rilevata nel blocco di programma delimitato dal costrutto;

Programmazione funzionale, ovvero costrutti per la programmazione funzionale come funzioni lambda, map, filter ecc.

Unificazione: l'unificazione è utile nei linguaggi dichiarativi dove le variabili presenti nelle regole dei programmi vengono istanziate *unificando* i loro termini. Per questo tipo di linguaggi, spesso sono previsti meccanismi che servono a modificare la modalità di risoluzione di una certa regola. I meccanismi *backtracking* e *cut* in Prolog ne sono un esempio.

I linguaggi imperativi contengono sempre costrutti per la selezione e la iterazione. I linguaggi orientati agli oggetti aggiungono a questi i costrutti per la OOP e la gestione delle eccezioni. I costrutti per la programmazione funzionale sono naturalmente determinanti per i linguaggi funzionali puri ma possono essere presenti anche in linguaggi di altro tipo. L'unificazione è invece contemplata nel caso dei linguaggi dichiarativi e logici.

1.4 Python

Python è un linguaggio di programmazione ad alto livello, orientato agli oggetti. Ideato da Guido Van Rossum (Matematico/Informatico) all'inizio degli anni novanta. Il nome fu scelto per via della passione di Van Rossum per i *Monthy Python* e per la loro serie televisiva *Monty Pythons Flying Circus* (commedia televisiva britannica di enorme successo dei primi anni settanta).

Python in realtà è molte cose. In Python è possibile programmare seguendo diversi paradigmi di programmazione: imperativo, funzionale, OOP. Inoltre, la tipizzazione dinamica di Python lo rende un linguaggio di programmazione ideale per la prototipazione rapida di una applicazione. Parti di un programma scritto in Python possono venire successivamente tradotte e/o compilate in altri linguaggi più performanti, come il C per esempio, nel caso in cui l'efficienza sia un requisito particolarmente stringente.

Dal punto di vista applicativo, Python è stato utilizzato per scopi dei più vari: programmazione numerica e scientifica, programmazione di sistema, internet, database, elaborazione delle immagini, interfacce grafiche, controllo di robot, giochi e multimedia, ecc.

2

Panoramica su Python

In questo capitolo esploreremo alcune delle caratteristiche di base del linguaggio Python. Il Python si presenterà da subito per quello che è: un linguaggio intuitivo che permette di effettuare operazioni anche complesse con pochi semplici comandi. Dal punto di vista della semplicità di utilizzo esso non differisce molto da una comune calcolatrice e ciò è prevalentemente dovuto al fatto che Python è un linguaggio interpretato, orientato agli oggetti e con tipizzazione dinamica. Tutte queste caratteristiche permettono di formulare i comandi all'interprete nel modo più sintetico e intuitivo possibile. Per questa panoramica utilizzeremo principalmente la prima delle due modalità di interazione con l'interprete: la *modalità interattiva*.

I contenuti di questo capitolo sono molto importanti per comprendere la "meccanica" che sta dietro all'interprete Python. Parleremo di oggetti, valutazione, operazioni, funzioni e metodi, ovvero dei meccanismi di base del linguaggio. Tali contenuti risultano particolarmente importanti per chi avesse già una qualche esperienza con altri linguaggi di programmazione. Essi noteranno che le scelte progettuali che stanno dietro Python sono non di rado molto diverse da quelle fatte per altri linguaggi di programmazione.

2.1 Console Python, toccata e fuga

Assumiamo che Python sia già stato installato nel computer. Per eseguire Python nella modalità interattiva possiamo utilizzare il semplice comando `python` dal terminale del sistema operativo.

```
C:\Users\Aio> python
Python 2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (
    AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
    information.
>>>
```

Appena entrati nella modalità interattiva, Python si presenta mostrandoci alcune informazioni riguardo alla versione installata, l'architettura su cui è installata e alcuni suggerimenti per ottenere aiuto o informazioni addizionali. Qualunque espressione o comando valido inserito nella console a questo punto viene

valutato dall'interprete Python (PVM) e il valore calcolato viene quindi visualizzato sulla console. Per esempio, se inseriamo una frase tra singole o doppie virgolette, l'interprete risponde stampando a video la stessa stringa di caratteri.

```
>>> "Ciao Mondo!"
 'Ciao Mondo!'
```

Per dire all'interprete di non valutare l'inserimento dell'utente, è possibile utilizzare il carattere speciale "cancelletto" `#` (o *hash* in inglese) spesso usato nella programmazione per commentare il codice prodotto e renderlo così più leggibile. In pratica, tutto quello che segue il cancelletto non viene valutato dall'interprete. Provando a ripetere il comando di prima, ma questa volta commentato, vediamo che l'interprete rimarrà in attesa di una espressione o comando da valutare valido. Una ulteriore pressione del tasto `Invio` ci permetterà di continuare.

```
>>> # 'Ciao Mondo!'
 ...
```

Per uscire dalla modalità interattiva possiamo utilizzare il comando `quit()`.

```
>>> quit()
C:\Users\Aio>
```

2.2 Un mondo di oggetti

Python è un linguaggio orientato agli oggetti. Già questa caratterizzazione ci porta a concentrarsi da subito sul ruolo centrale dell'entità *oggetto* in questo linguaggio. Intuitivamente un oggetto è un contenitore per un dato. Vedremo che in Python è possibile creare e operare su oggetti tanto semplici come i numeri interi o reali, ma anche su oggetti ben più complessi come per esempio *matrici*, *file*, *database*, *immagini*, oppure su oggetti che a loro volta contengono altri oggetti. In effetti tutto in Python riguarda oggetti, relazioni tra oggetti e operazioni su oggetti, e per questo motivo è importante capire bene fin da subito come questi vengono gestiti dall'interprete Python.

Ogni oggetto in Python è contraddistinto da tre attributi fondamentali: una *identità*, un *tipo* e un *valore*. Vediamoli uno per uno.

L' **identità** è un attributo unico per uno specifico oggetto. Non possono esistere più oggetti con la stessa identità. Per ottenere l'identità di un oggetto si può ricorrere alla funzione `id()`. Per esempio, col seguente comando otteniamo l'identità del numero intero `3`.

```
>>> id(3)
 40556936L
```

Non siate sorpresi se il valore a voi ritornato come identità differisce da quello qui sopra riportato. L'identità assegnata ai vari oggetti viene infatti gestita autonomamente dall'interprete (l'utente non ne ha il controllo) e dipende dalla specifica sessione.

È comunque possibile verificare se due oggetti specificati sono in realtà lo stesso oggetto, ovvero se hanno la stessa identità, mediante l'operatore `is`:

```
>>> 3 is 3
 True
>>> 3 is 4
 False
```

Il **tipo** è quell'attributo di un oggetto che ne determina quali operazioni è possibile effettuare su di esso e quali valori può assumere. Per esempio possiamo applicare le operazioni aritmetiche standard tra numeri interi se i due operandi sono di tipo `int` (intero). Viceversa, non è permesso sommare un oggetto di tipo `int` con un oggetto di tipo `str` (una stringa o sequenza di caratteri). Il seguente comando all'interprete Python darà errore:

```
>>> 3 + "ciao" #da' un errore
 TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Possiamo verificare il tipo di un oggetto con la funzione `type()` che ritorna un oggetto di tipo `type` che ne descrive il tipo.

```
>>> type(3)
 int
>>> type("ciao")
 str
>>> type(3.14)
 float
```

Il **valore** di un oggetto, infine, rappresenta il dato vero e proprio contenuto nell'oggetto, ovvero il valore risultante quando l'oggetto viene valutato. Bisogna sempre ricordare che due oggetti possono avere uguale tipo e valore ma diversa identità. Questo succede quando in memoria esistono due oggetti distinti. L'esempio seguente mostra chiaramente questa cosa:

```
>>> 2.2 + 1.0 == 3.2
 True
>>> 2.2 + 1.0 is 3.2
 False
```

dove abbiamo usato l' operatore == per confrontare il valore di due oggetti.

Riassumendo, possiamo dare la seguente regola mnemonica: *(i)* l'identità rappresenta il *chi* dell'oggetto, ovvero la sua identità, a quale oggetto ci si riferisce in particolare; *(ii)* il tipo ne determina il *come*, ovvero come è possibile utilizzarlo; *(iii)* il valore invece rappresenta il *cosa*, ovvero cosa contiene in realtà l'oggetto.

2.3 Espressioni e valutazione

Una espressione è una combinazione di valori mediante operatori. Ogni espressione restituisce un oggetto di un tipo ben preciso determinato dal tipo di operazione (operatore) e dal tipo degli oggetti su cui opera (operandi). L'espressione

più semplice consiste di un singolo valore (o *letterale*) per esempio una stringa o un numero. Dal seguente esempio si evince come la valutazione di un'espressione che consiste di un singolo oggetto è il valore dell'oggetto stesso.

```
>>> "ciao gente"
 'ciao gente'
>>> 3.14159
 3.14159
```

Vediamo come sia possibile combinare i valori tramite espressioni leggermente più complesse. Se scriviamo:

```
>>> 2 + 3
 5
```

abbiamo un'espressione con operandi di tipo `int` con operatore `+` che, come si può facilmente immaginare, rappresenta la somma tra numeri, in questo caso interi. Il risultato sarà ancora un numero intero, ovvero `5`. Ma proviamo a ripetere la stessa operazione, questa volta su due stringhe.

```
>>> "pinco" + "pallino"
 'pincopallino'
```

In questo caso il risultato è un altro oggetto stringa che contiene la concatenazione delle stringhe date. Come è stato possibile ottenere due oggetti di tipo diverso con lo stesso tipo di operazione? Vedremo che questo comportamento è del tutto normale in Python e dobbiamo abituarcisi, quindi conviene affrontare l'argomento sin da subito. Per i più tecnici, il meccanismo ha a che fare con l'*overloading* (o sovraccaricamento) degli operatori. In parole povere significa che possiamo utilizzare lo stesso simbolo per effettuare operazioni semanticamente diverse ammesso che gli operandi utilizzati siano di tipo diverso. Prendiamo un altro esempio, la divisione, che in Python (come in altri linguaggi di programmazione) si indica con il simbolo `/` e proviamo a dividere due numeri interi.

```
>>> 2 / 3
 0
```

Qualcuno poteva aspettarsi che il valore ritornato fosse il numero 0.66666 con 6 periodico. E invece no, perché anche questo è un caso (un po' più subdolo) di overloading degli operatori. Quando entrambi gli operandi sono di tipo intero, l'operatore `/` rappresenta la divisione intera tra gli operandi. Per verificarlo possiamo scrivere:

```
>>> 2.0 / 3
 0.6666666666666
```

Cosa è cambiato rispetto a prima? Che adesso almeno uno degli operandi della divisione è di tipo `float` e in questo caso l'operazione `/` viene interpretata come una divisione nel senso classico. Un'avvertenza: dalle versioni più recenti di Python (3.x), il simbolo `/` rappresenta sempre la divisione classica e quindi restituisce un oggetto di tipo `float`. L'operatore di divisione intera è stato sostituito con il nuovo operatore `//`. Nelle versioni 2.x di Python è comunque possibile (e consigliabile) utilizzare l'operatore `//` esclusivo per la divisione intera.

Vediamo un altro esempio di overloading di un operatore. In Python, l'operatore $\star$ (asterisco) può essere utilizzato sia per la moltiplicazione classica tra numeri, sia per replicare una stringa. Se usiamo l'operatore $\star$ tra una stringa e un intero n, il risultato dell'espressione sarà una stringa contenente n repliche della stringa originale:

```
>>> "miao"*3
 'miaomiaomiao'
>>> 4*"bau"
 'baubaubaubau'
```

Le espressioni possono essere utilizzate a loro volta come sottoespressioni di espressioni più complesse. Quando più operatori compaiono in una stessa espressione, l'interprete Python valuta l'espressione ricorrendo alla *precedenza degli operatori*. In generale valgono le stesse regole della precedenza o priorità degli operatori matematici: prima le parentesi, poi l'elevamento a potenza, poi moltiplicazione e divisione, infine addizione e sottrazione. Quando due operatori hanno la stessa priorità, l'ordine di valutazione è quello da sinistra a destra. Facciamo qualche esempio per chiarire il concetto di priorità degli operatori:

```
>>> 2 * (4-1) # prima dentro la parentesi, poi il prodotto
 6
>>> (3-1)**(4-1) # prima dentro le parentesi, poi la potenza
 8
>>> 2*2**3 # potenza, poi prodotto
 16
>>> 1+2*2**3 # potenza, poi prodotto, poi somma
 17
>>> 3+4-2 # da sx a dx
 5
>>> 20/10*2 # da sx a dx
 4
>>> 20/(10*2) # prima la parentesi
 1
>>> "cu"*2+"ru"+"cu"*2 # prima repliche poi concatenazione
 'cucurucucu'
```

A parte i casi più semplici, è sempre raccomandabile l'uso di parentesi per il raggruppamento degli operatori e degli operandi. In tal modo riusciamo a specificare in modo esplicito le precedenze rendendo al tempo stesso il programma più leggibile.

Abbiamo parlato di operatori. Le funzioni e i metodi sono altri modi con cui si possono generare valori utilizzabili all'interno di espressioni. La differenza principale tra funzioni e metodi è che una funzione viene chiamata passando dei valori come argomenti. Per esempio, la funzione `len()` ritorna la lunghezza di una stringa passata come parametro, quindi `len('abecedario')` ritornerà un intero con valore `10`. Un metodo invece è una particolare funzione che opera

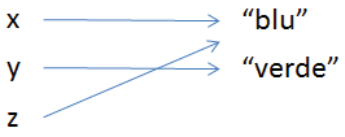


Figura 2.1: Riferimenti ad oggetti

su di un oggetto specifico. Nella chiamata al metodo, l'oggetto su cui vogliamo applicare il metodo e il nome del metodo sono separati da un punto. Per esempio, il metodo `count()` viene applicato su una stringa e prende un'altra stringa come parametro, come nella chiamata `'abecedario'.count('ec')` che ritorna il numero di occorrenze della sottostringa `'ec'` nella stringa `'abecedario'`.

2.4 Riferimenti ad oggetti

Negli esempi visti fino ad ora, dopo aver calcolato il valore di una certa espressione, perdiamo ogni traccia del relativo oggetto. Ne segue che tale valore non può essere direttamente recuperato per riutilizzarlo eventualmente in un'altra espressione. Come possiamo memorizzare i risultati parziali di una computazione di cui magari potremmo aver bisogno successivamente in un programma? Per questo, nei comuni linguaggi di programmazione, esistono le variabili (che in Python sono in realtà dei *riferimenti*). Iniziamo con un semplice esempio giusto per capire come funzionano:

```
>>> x="blu"
>>> y="verde"
>>> z=x
```

L'operatore = è da intendersi come l' *operatore di assegnamento* e non come operatore di eguaglianza (==). L'operatore di assegnamento serve a creare/modificare un riferimento ad un oggetto.

In Figura 2.1 è mostrato lo schema (o diagramma) di stato dopo l'esecuzione di questi semplici assegnamenti. L'esecuzione della prima riga fa sì che si crei un oggetto di tipo `str` a partire dal letterale `'blu'` e un riferimento chiamato `x` a tale oggetto. Da questo punto in poi, possiamo riferirci all'oggetto stringa `'blu'` indicando il nome `x`. In modo simile, nella seconda riga si crea un oggetto di tipo `str` a partire dal letterale `'verde'` e un riferimento ad esso chiamato `y`. Infine, nella terza riga, viene creato un altro riferimento chiamato `z` che si riferisce allo stesso oggetto a cui si riferisce `x`. In quest'ultimo caso non viene creato nessun nuovo oggetto ma solo un nuovo riferimento ad un oggetto già esistente! Possiamo facilmente verificare questa situazione con l'ausilio della funzione `id()`.

```
>>> id(x)
 56364488L
>>> id(y)
```

```
 56365008L
>>> id(z)
 56364488L
```

Ci accorgiamo che l'identità dell'oggetto riferito da `x` e da `z` è la stessa e ciò vuol dire che, nonostante abbiano nomi diversi, essi riferiscono lo stesso oggetto.

La scelta del nome che il programmatore assegna ad un riferimento deve osservare alcune semplici regole per evitare errori in esecuzione dell'interprete. In particolare:

1. Il primo carattere può solo essere un carattere alfabetico: i nomi dei riferimenti devono iniziare per un carattere maiuscolo (`A,..,Z`) o minuscolo (`a,..,z`) o al limite il segno di underscore (`_`).
2. Solo caratteri alfanumerici: ad esclusione del primo, che come abbiamo detto può solo essere alfabetico, gli altri caratteri del nome possono essere tutti i caratteri alfabetici (`A,..,Z, a,..,z`) non accentati, i caratteri numerici (`0,..,9`) e l'underscore. Sono esclusi quindi tutti i caratteri di interpunzione (virgola, punto, virgolette, punto esclamativo, ecc.) e gli operatori (`*,-,%`, ecc.).
3. Sono escluse le parole chiave del linguaggio: non è possibile assegnare un nome ad un riferimento che coincida con una parola riservata del linguaggio (per esempio `class`, `if`, `elif`, `finally`, `import`, `return`, `None`, ecc.).

Si tenga presente che, quando l'interprete usa i comandi, le variabili, le funzioni, ecc., opera sempre in modalità *case-sensitive*, cioè fa distinzione tra maiuscole e minuscole. Quindi anche i nomi dei riferimenti non fanno eccezione a questa regola e `x` e `X` saranno considerati nomi di riferimenti distinti.

Una caratteristica davvero utile in Python è il cosiddetto assegnamento in linea di più variabili. Questo tipo di assegnamento permette di modificare più riferimenti "contemporaneamente". Nel listato seguente è mostrato come sia possibile effettuare lo scambio dei valori di due variabili in una sola riga di codice in maniera molto semplice.

```
>>> x,y = 1,2 # assegnamento multiplo
>>> x,y = y,x # scambia il valore delle variabili
>>> x
 2
>>> y
 1
```

Dato un riferimento ad un oggetto, possiamo operare direttamente su di esso. Per esempio:

```
>>> x = 23 # x adesso riferisce l'oggetto 23
>>> x += 3 # equivalente a x = x+3, ovvero x -> 26
>>> x /= 2 # equivalente a x = x/2, ovvero x -> 13
>>> x *= 3 # equivalente a x = x*3, ovvero x -> 39
```

```
>>> x -= 4 # equivalente a x = x-4, ovvero x -> 35
>>> x %= 4 # equivalente a x = x%4 (modulo), ovvero x -> 3
>>> x
3
```

Da notare che, in tutti i casi sopra, l'oggetto riferito non viene modificato. Semplicemente viene creato un nuovo oggetto con il valore risultante e dirottato il riferimento verso tale oggetto.

2.5 Scripting

La modalità interattiva di Python ci permette di provare interattivamente il codice prodotto massimizzando il controllo del programmatore sulle variabili del sistema. Lo svantaggio principale di questa modalità è che essa non permette di rieseguire il codice più volte. La modalità che tratteremo adesso, lo *scripting*, permette invece di eseguire le istruzioni contenute in un file di testo precedentemente salvato nel nostro computer.

Il programma o script da eseguire è un semplice file di testo con estensione `".py"`. Immaginando di avere un file salvato con nome `nome_script.py`, il comando da lanciare per eseguire uno script è semplicemente

```
C:\Users\Aio> python nome_script.py
```

facendo attenzione che *(i)* l'interprete Python sia nel *path* del sistema; *(ii)* il file indicato si trovi nella directory corrente (in alternativa, dobbiamo indicare l'intero path del file).

2.6 Input e output

Nella modalità interattiva, l'interfaccia tra utente e sistema è delegata interamente all'ambiente Python, nel senso che l'utente immette espressioni o comandi da tastiera e l'output viene prodotto come il risultato della valutazione dell'espressione inserita. Nella modalità scripting, invece, l'interazione uomo-macchina è gestita dal programmatore che si dovrà occupare di inserire esplicitamente i comandi per la lettura dei dati dall'esterno e per la scrittura in output dei risultati. A questo scopo Python prevede due comandi principali:

- la funzione `raw_input()` per la richiesta da terminale di una stringa di caratteri;
- la funzione `print()` o il comando `print` per la scrittura a video.

La funzione `raw_input(<prompt>)` prende come argomento (opzionale) una stringa, che verrà utilizzata come prompt della richiesta all'utente, e ritorna la stringa contenente i caratteri inseriti dall'utente. Facciamo un semplice esempio:

```
1 frase_letta = raw_input("Inserire una frase:")
```

Eseguendo questo semplice script verrà prima visualizzato il messaggio *"Inserire una frase:"*, quindi il programma si mette in attesa di un input dall'utente. Una volta che l'utente ha inserito un testo e premuto il tasto `Invio`, la funzione `raw_input()` ritorna un nuovo oggetto di tipo `str` contenente il testo inserito. In questo esempio, l'oggetto risultante viene riferito col nome `frase_letta`.

La funzione `print(<obj>)` prende come argomento un qualsiasi oggetto e ne stampa a video una stringa che ne rappresenta il valore. Non per tutti i tipi di oggetti è definito un formato di visualizzazione e, quando un oggetto non ne è provvisto, viene semplicemente visualizzata una stringa che ne descrive il tipo. Nel seguente script di esempio vengono stampate una stringa, una lista e infine il nome (senza parentesi) di una funzione built-in di Python (la funzione `dir()`).

```
1 print("Hello World!")
2 print([1,2,[3,4],5])
3 print(dir) # stampa la riga <built-in function dir>
```

Eseguendo lo script si produce la seguente stampa a video:

```
Hello World!
[1, 2, [3, 4], 5]
<built-in function dir>
```

Fino alla versione 2.7 di Python, alla quale ci riferiamo in questo libro, per la stampa a video veniva utilizzato il comando `print` invece della funzione descritta sopra. L'utilizzo del comando non differisce molto da quello della funzione poiché in questo caso la sintassi è semplicemente `print <obj>`.

2.7 Importare moduli esterni

È generalmente buona prassi suddividere un programma in un insieme di parti indipendenti che assolvono a compiti particolari. Questa caratteristica di progettazione del software prende il nome di *modularità*. Un software modulare permette il riutilizzo di alcune sue parti e una più facile manutenibilità. Questo tipo di suddivisione del codice è naturalmente implementata in Python dai cosiddetti *moduli*.

Un modulo è un modo di organizzare in file separati il codice che implementa un insieme di funzionalità coerenti. Un modulo può contenere codice da eseguire, ma anche definizioni di variabili, funzioni e classi aggiuntive. Molte delle funzionalità di base più utili al programmatore sono già disponibili all'interno di moduli Python e quindi facilmente integrabili nel proprio codice all'occorrenza.

Prima di poter essere usati, i moduli necessitano di essere importati. Per importare un modulo si usa il comando `import` la cui sintassi è

```
import nome_modulo
```

Quando un modulo contiene codice Python eseguibile, l'importazione del modulo fa sì che il codice al suo interno venga subito eseguito.

Tra i moduli più utilizzati, che ritroveremo nel seguito, citiamo il modulo `math` per le funzioni e le costanti matematiche, il modulo `random` per la gestione dei numeri pseudo-casuali, il modulo `os` per l'interfaccia al sistema operativo e il modulo `sys` per le funzioni ausiliare utili per interagire con l'interprete. Vedere Appendici A, B e C per maggiori dettagli sui moduli qui sopra menzionati.

Una volta importato un modulo, sarà possibile utilizzare il suo contenuto, siano semplici nomi di variabili, funzioni o tipi aggiuntivi del modulo, facendo seguire il nome della risorsa al nome del modulo con l'operatore punto. Prendiamo brevemente il modulo `math` e vediamo un paio di esempi in modalità interattiva che dimostrano come sia semplce utilizzare le risorse presenti in un modulo.

```
>>> import math
>>> math.cos(2*math.pi) # coseno di 360 gradi (2*pi_greco)
 1.0
>>> math.sqrt(16) # radice quadrata di 16
 4.0
```

Come si vede nell' esempio, tutte le volte che si utilizza una risorsa del modulo è obbligatorio indicare anche il nome del modulo che contiene quella risorsa. Volendo aggirare questa restrizione possiamo comunque importare tutte le risorse del modulo utilizzando una diversa sintassi per l'importazione.

```
>>> from math import *
>>> cos(2*math.pi)
 1.0
>>> sqrt(16)
 4.0
```

Il simbolo `*` sta proprio per "tutto". Nel caso invece volessimo importare solo un numero limitato di risorse di un modulo, senza importare necessariamente tutto il modulo, possiamo indicare una a una le risorse che vogliamo importare come nel seguente esempio.

```
>>> from math import cos,sqrt
>>> cos(2*math.pi)
 1.0
>>> sqrt(16)
 4.0
```

Esercizi del capitolo

Esercizio 2.1

1. *Quali sono le caratteristiche di un oggetto Python?*
2. *Qual è la differenza tra un oggetto e un riferimento?*
3. *Enumerare tutti i casi di overloading che conoscete che riguardano l'operatore + e i tipi int, float, str;*
4. *Enumerare tutti i casi di overloading che conoscete che riguardano l'operatore * e i tipi int, float, str.*

Esercizio 2.2

Dati i tre assegnamenti: `x = 1, y = 3, z = 0`.

Calcolare la somma tra `x` *e* `y` *e salvare il risultato in* `z`*;*

Porre `x` *uguale a* `y`

Incrementare `y` *di* `2`

a) Quanto vale `x`*?*

Calcolare il prodotto tra `y`*,* `z` *e* `x` *e salvare il risultato in* `z`*;*

b) Quanto vale `z`*?*

Decrementare `y` *di* `1`

c) E ora?

Salvare in `x` *il risultato di* $z^{1/y}$

d) Verificare che le prime cifre decimali di `x` *valgano* 783.

Esercizio 2.3

Indicare i valori assunti dalle stringhe `s1` *e* `s2` *dopo la seguente serie di operazioni:*

```
s1="a"
s2="b"
s2+=3*s1+2*s2+2*(s1*2+s2)
s1*=2
```

Esercizio 2.4

Usando la modalità interattiva di Python, calcolare le seguenti espressioni per `x=1`*,* `x=5` *e verificare che i risultati ottenuti siano quelli attesi*

- $2x + 8\frac{4^2}{2}$, (risultati attesi: 66 e 74)
- $2x + 4^{\frac{1}{2}}$, (risultati attesi: 4 e 12)

3

Tipi di base del Python

In questo capitolo approfondiremo i tipi standard del Python e le numerose operazioni disponibili per operare su di essi. Ci concentreremo prima di tutto sulla loro classificazione rispetto a certe proprietà astratte. Tale classificazione ci permetterà in futuro di determinare e ricordare più facilmente quali operazioni è possibile effettuare su quali tipi di dati.

3.1 Tipi per tutti i gusti

Python comprende una ricca famiglia di tipi di base che includono: tipi numerici, stringhe, tuple, liste, dizionari e insiemi. Oltre a questi, che sono solo una parte dei tipi standard disponibili, Python permette anche di definire nuovi tipi con l'ausilio delle classi, ma di questo parleremo in un prossimo capitolo.

I tipi su cui ci focalizziamo in questo capitolo sono:

Numeri che rappresentano i classici tipi numerici: numeri interi (`int`), numeri in virgola mobile (`float`), numeri complessi (`complex`), numeri booleani (`bool`).

Stringhe (`str`) che, come abbiamo già visto, sono sequenze di caratteri adatti a rappresentare un testo.

Tuple (`tuple`) che rappresentano valori composti a loro volta da un predefinito numero di altri valori possibilmente eterogenei (ovvero di tipi diversi tra loro). Per esempio, possiamo considerare un singolo ordine ad un fornitore come la tripla: nome fornitore (`str`), codice articolo (`int`), quantità (`float`).

Liste (`list`) che rappresentano elenchi di valori, possibilmente eterogenei, di lunghezza imprecisata. Per esempio, l'elenco degli studenti frequentanti inseriti in ordine di iscrizione: Francesca, Mauro, Gianni.

Dizionari (`dict`) che rappresentano informazioni di tipo associativo, ovvero ogni valore viene associato ad una chiave unica. Un esempio classico è l'indice analitico di un libro, dove ogni termine rimanda alla pagina, o alle

pagine, che trattano di quell'argomento. Un altro esempio è l'associazione tra codice (unico) di un cliente e il suo nominativo. Notare che, in questo ultimo caso, potrebbero anche esistere più clienti con lo stesso nome (un caso di omonimia) ma l'ambiguità viene risolta dall'unicità del codice.

Insiemi (`set`) che rappresentano insiemi di valori, possibilmente eterogenei, di lunghezza imprecisata. Per esempio, un oggetto che rappresenta la lista della spesa: "latte", "pane", "burro", "pasta".

3.2 Classificazione dei tipi standard

Prima di esplorare uno ad uno i principali tipi di base di Python, presentiamo qui una sommaria classificazione che risulterà utile successivamente in questo capitolo per comprendere il tipo di operazioni che è possibile applicare ai vari oggetti, nonché quando è meglio utilizzare un oggetto di un tipo o di un altro a seconda della particolare informazione che vogliamo memorizzare.

Modalità di memorizzazione. In Python esistono oggetti che possono contenere altri oggetti (per esempio un oggetto lista). Quando un tipo rappresenta un oggetto singolo diciamo che il tipo è *atomico*. Viceversa, quando gli oggetti di un tipo possono contenere altri oggetti, parliamo di tipo *contenitore*. Esempi di tipo atomico sono tutti i tipi numerici e le stringhe. Esempi di tipo contenitore sono le tuple, le liste, i dizionari e gli insiemi.

Modalità di aggiornamento. Alcune tipologie di oggetti Python sono *mutabili* nel senso che è possibile modificare il valore dell'oggetto. Viceversa, per altri tipi di oggetti, il valore viene definito in fase di creazione dell'oggetto e non può essere modificato. Questa definizione di immutabilità, seppur sufficiente ai nostri scopi attuali, risulta in verità incompleta. Raffineremo più avanti la definizione di tipo mutabile/immutabile per renderla più precisa e meno ambigua. Esempi di tipi immutabili sono tutti i tipi numerici, le stringhe e le tuple. Tipi mutabili sono le liste, i dizionari e gli insiemi. Una caratteristica importante relativa alla modalità di aggiornamento è che solo oggetti di un tipo immutabile possono essere utilizzati come chiavi di un dizionario o come elementi di un insieme.

Modalità di accesso. Infine, possiamo classificare un oggetto Python a seconda della modalità di accesso al dato contenuto. Quando l'oggetto è adatto per contenere una molteplicità di elementi al suo interno (per esempio in una stringa o in un qualsiasi contenitore), allora è spesso possibile accedere ai singoli elementi in modo indiretto oppure iterare sugli elementi. In questo caso diciamo che l'oggetto è *iterabile*. Tra gli oggetti iterabili è opportuno fare un'altra distinzione riguardo l'accesso agli elementi contenuti. In alcuni casi, gli elementi sono posizionalmente ordinati, come in una sequenza, e sarà possibile effettuare un accesso tramite un indice che parte da `0` per indicare il primo elemento, `1` per

Tipo	Mutabile	Contenitore	Sequenza	Associativo
Stringhe	NO	NO	SI	NO
Tuple	NO	SI	SI	NO
Liste	SI	SI	SI	NO
Dizionari	SI	SI	NO	SI
Insiemi	SI	SI	NO	NO

Tabella 3.1: Classificazione dei tipi di base iterabili

indicare il secondo, e così via. In questi casi parliamo di accesso *sequenziale* al dato. In altri casi, l'accesso agli elementi è di tipo *associativo*, ovvero, simile a quello delle sequenze ma, invece che mediante indici interi, l'accesso viene effettuato mediante una *chiave* unica. Stringhe, liste e tuple sono tipi sequenziali (o sequenze) mentre i dizionari sono di tipo associativo.

Una prima serie di tipi predefiniti e la loro catalogazione in una delle succitate categorie è presentata in Tabella 3.1. Per evitare inutili ripetizioni, molte funzioni presentate più avanti in questo stesso capitolo, saranno raggruppate per categoria. Una importante caratteristica del linguaggio Python è proprio la coerenza. In Python, le funzioni che hanno uguale scopo e agiscono su una caratteristica comune tra due tipi fondamentali diversi hanno lo stesso nome e la stessa modalità di utilizzo quando applicate a oggetti di un tipo o dell'altro.

3.3 Tipi Atomici

Iniziamo a parlare dei tipi atomici, ovvero i tipi numerici e le stringhe.

3.3.1 Numeri

In Python non esiste un unico tipo per descrivere tutti i numeri ma bensí un'intera gamma di tipi simili uno all'altro ma ognuno con caratteristiche peculiari. I due tipi più utilizzati sono i numeri interi (`int`) e i numeri in virgola mobile (`float`) che abbiamo già incontrato in precedenza. I tipi numerici di base supportano le normali operazioni numeriche. Il simbolo + corrisponde all'addizione, il simbolo − alla sottrazione, l'asterisco $\star$ effettua la moltiplicazione, il simbolo / (slash) la divisione e i due asterischi $\star\star$ l'elevamento a potenza.

Numeri interi e in virgola mobile

Iniziamo con i tipi numerici di gran lunga più utilizzati, ovvero i numeri interi (tipo `int`) e i numeri in virgola mobile (tipo `float`). Facciamo alcuni esempi di letterali numerici di base.

```
>>> x = 1234 # un intero
>>> y = -24 # un intero negativo
>>> type(y)
```

```
 int
>>> z = 1.23 # un float
>>> 3.1e120 # un altro float
 3.1e+120
>>> v = 3.1E+120 # ancora un altro
 3.1e+120
```

Numeri Complessi

I numeri complessi sono definiti mediante una coppia di numeri in virgola mobile che rispettivamente rappresentano la parte reale e la parte immaginaria del numero. Python codifica un numero complesso aggiungendo un suffisso `j` (o `J`) alla parte immaginaria. Ecco alcuni semplici esempi che dimostrano l'utilizzo dei numeri complessi.

```
>>> 1j # un numero complesso
 1j
>>> 3.12 + 3J
 (3.12 + 3j)
>>> c = 1 + 1j*5
>>> type(c)
 <type 'complex'>
>>> c.real
 1.0
>>> c.imag
 5.0
```

Booleani

Un altro tipo predefinito del Python è il tipo booleano (`bool`). I valori booleani rappresentano i valori di verità (vero e falso). In prima istanza tali valori possono essere pensati come i valori numerici `1` e `0` e in effetti, in molti casi, i valori booleani possono venir usati al posto degli interi. Vedremo più avanti quando è conveniente utilizzare i valori booleani invece degli interi e quali sono le relazioni tra questi due tipi. Intanto facciamo qualche semplice esempio.

```
>>> True
 True
>>> False
 False
>>> True or False
 True
>>> True and False
 False
>>> type(True)
 <type 'bool'>
>>> True==1 # interessante no? True e 1 hanno lo stesso valore
 True
>>> True is 1 # ma non sono lo stesso oggetto
 False
```

Gerarchia tra Numeri

I numeri in Python rispettano una certa gerarchia che è coerente al loro utilizzo in applicazioni matematiche. In particolare, i valori booleani (`False` e `True`) possono essere usati anche come i numeri interi (0,1) rispettivamente. I numeri interi sono numeri in virgola mobile particolari (senza cifre decimali). Infine, i numeri in virgola mobile sono particolari numeri complessi senza parte immaginaria. La gerarchia è quindi la seguente:

$$\text{bool} \subset \text{int} \subset \text{float} \subset \text{complex}$$

Questa gerarchia dei numeri permette a Python di identificare il tipo dell'oggetto risultato quando un operazione coinvolge numeri di natura diversa trasformando tutti i numeri coinvolti al tipo "più grande" (più a destra nella gerarchia descritta in precedenza). Facciamo un esempio in cui si costruiscono due numeri, uno `float` ed uno `complex`, se ne verifica il tipo individualmente, e poi verifichiamo che il tipo del risultato della somma dei due numeri sia effettivamente del tipo più grande.

```
>>> c = 2+3j
>>> f = 3.1
>>> type(c)
 <type 'complex'>
>>> type(f)
 <type 'float'>
>>> type(c+f)
 <type 'complex'>
```

Conversione di numeri

Il primo tipo di funzioni che vedremo un po' più in dettaglio sono le funzioni di conversione da un tipo ad un altro. Queste funzioni prendono come argomento un oggetto di un tipo e restituiscono un oggetto del tipo richiesto, se questo è possibile, altrimenti produrranno un errore. Vediamo un primo esempio con la funzione `int()` che genera un oggetto di tipo `int`.

```
>>> x = 3.14
>>> type(x)
 float
>>> y=int(x)
>>> y
 3
>>> type(y)
 int
```

Applicare la funzione `int()` ad un argomento di tipo `float` restituisce un nuovo oggetto `int` contenente il numero originale troncato.

3.3.2 Stringhe

Le stringhe sono fondamentalmente sequenze di caratteri. In programmazione esse hanno un largo utilizzo poiché possono essere usate per rappresentare qualsiasi informazione di tipo testuale come parole, frasi, il contenuto di un file di testo, indirizzi internet/email, pagine web, e così via. Python codifica le stringhe come sequenze di caratteri, o simboli, delimitati da apici singoli o doppi. Diamo di seguito due esempi di stringhe *palindrome* (provate a leggerle al contrario).

```
>>> s1 = 'I topi non avevano nipoti'
>>> s2 = "Alle carte t'alleni nella tetra cella"
```

Nota che l'uso della doppia notazione (singoli e doppi apici) ci permette di includere apici o doppi apici nella stringa stessa. Esistono delle stringhe particolari. Per esempio, la stringa vuota (che non contiene caratteri) è rappresentata con due singoli apici uno dopo l'altro `''` o due doppi apici `""` senza chiaramente inserire spazi tra i due apici poiché in questo caso la stringa non sarebbe più vuota.

Abbiamo già visto come le stringhe possano comparire all'interno di espressioni. Facciamo qualche esempio ulteriore.

```
>>> s = '' # stringa vuota
>>> s1 = "abc"
>>> s2 = "def"
>>> s3 = (s1+s2)*3
>>> s3
 abcdefabcdefabcdef
```

3.3.3 Conversione tra numeri e stringhe

Possiamo convertire numeri con stringhe e viceversa. Nel primo esempio qua sotto abbiamo una stringa che rappresenta un numero (intero in questo caso) e lo traduciamo nella stringa corrispondente.

```
>>> x = "3"
>>> type(x)
 str
>>> y=int(x)
>>> y
 3
>>> type(y)
 int
```

Allo stesso modo è possibile ottenere oggetti di un altro tipo numerico a partire da oggetti di tipo stringa che contengono un letterale valido per quel tipo specifico:

```
>>> x = "3"
>>> float(x)
 3.0
>>> x = "3.1459"
>>> float(x)
```

```
 3.1459
>>> c = "4.3+2.1J"
>>> complex(c)
 (4.3+2.1j)
```

E viceversa per ottenere una stringa a partire da dati numerici:

```
>>> x = 3
>>> str(x)
 '3'
>>> x = 3.1459
>>> str(x)
 '3.1459'
```

Python ci offre molti metodi che operano sulle stringhe. Vale la pena ricordare che, essendo il tipo stringa un tipo immutabile, tali metodi non vanno a modificare la stringa su cui sono applicati ma bensì permettono di generare nuove stringhe a partire da queste. Invitiamo il lettore a consultare la documentazione per l'elenco completo di tali metodi. Qui di seguito offriamo solo alcuni esempi di metodi per le stringhe tra i più utilizzati.

```
>>> "abcde".capitalize()
 Abcde
>>> "abcde".center(10)
 '  abcde   '
>>> "abcbcabc".count('bc')
 3
>>> "abcbcabc".find('bc')
 1
>>> 'ab3 ab2'.isalnum()
 False
>>> 'abab'.isalpha()
 True
>>> '234'.isdigit()
 True
>>> 'abracadabra'.islower()
 True
>>> "ab;.bc;.cd".replace(';.','-')
 'ab-bc-cd'
>>> "  abc   ".strip()
 'abc'
>>> "aAbBcC".swapcase()
 'AaBbCc'
>>> "aBBbbc".upper()
 'ABBBBC'
```

3.3.4 Set di caratteri

In Python, a differenza di molti altri linguaggi di programmazione, non esiste il tipo carattere. Il carattere in Python viene semplicemente rappresentato da una stringa di lunghezza uno.

Il set di caratteri più spesso utilizzato è l'insieme ASCII che consiste di 256 codici distinti che coprono praticamente tutti i caratteri presenti nelle lingue occidentali, anche quelli accentati, e i più comuni caratteri di controllo che possono essere ottenuti utilizzando i tasti presenti nella tastiera di un computer. Un esempio di carattere di controllo è la combinazione CTRL-C spesso utilizzata per chiudere un applicazione. Nel caso sia necessario rappresentare più caratteri, per esempio caratteri asiatici, esiste la codifica UNICODE che associa 2 byte, invece che 1 byte, ad ogni carattere. In questo modo si possono rappresentare 65536 caratteri diversi.

In Python esistono due funzioni, `ord()` e `chr()`, per convertire caratteri nei loro codici ASCII e codici ASCII nel corrispondente carattere, rispettivamente. Di seguito viene mostrato l'utilizzo di queste due funzioni.

```
>>> ord('a')
 97
>>> ord('b')
 98
>>> ord('z')
 122
>> ord('z')-ord('a')+1 # nro caratteri tra 'a' e 'z' compresi
 26
>>> chr(65)
 'A'
>>> chr(90)
 'Z'
>> chr(ord('A')+6)
 'G'
```

Notare che tutti i caratteri alfabetici maiuscoli (`'A'`,...,`'Z'`) hanno codici consecutivi a partire da 65. I caratteri alfabetici minuscoli (`'a'`,...,`'z'`) si trovano a partire da 97 fino a 122 e sono a loro volta consecutivi.

3.4 Contenitori

Spesso è necessario utilizzare contenitori di oggetti, per esempio l'elenco degli studenti nella classe di Programmazione, la temperatura osservata in vari momenti della giornata, i semi delle carte da gioco (quadri, fiori, picche, cuori), ecc. Python ci offre tutta una serie di strutture dati (che chiameremo *contenitori*) per raggruppare oggetti. Il tipo di tali oggetti può anche essere eterogeneo. Per i nostri scopi, conviene pensare ai contenitori come oggetti particolari che contengono al loro interno riferimenti ad altri oggetti.

3.4.1 Tuple

Il primo tipo di contenitore che vediamo è la tupla. Una tupla è una sequenza di oggetti non necessariamente omogenei che si costruisce ponendo delle virgole tra i vari elementi, eventualmente tutto racchiuso tra parentesi. Gli elementi pos-

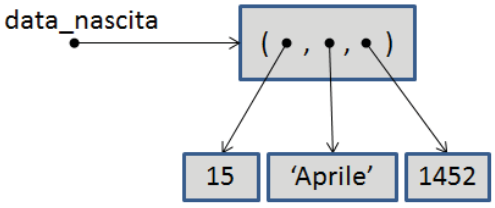


Figura 3.1: Diagramma di stato di una tupla

sono essere indicati sia usando il letterale corrispondente, sia usando il risultato della valutazione di una espressione. Vediamo subito un esempio:

```
>>> data_nascita=15,"Aprile",1452
>>> data_nascita
 (15,'Aprile',1452)
>>> leo=("Leonardo di ser Piero da Vinci",
         data_nascita, "Vinci")
```

Python denota sempre una tupla mettendone i valori tra parentesi tonde. In realtà una tupla si può indicare in entrambi modi. È solo necessario usare alcune (poche) accortezze. Per esempio, una tupla vuota si può indicare con parentesi aperta e chiusa `()` o utilizzando il costruttore di tupla `tuple()` senza argomenti, mentre una tupla con un solo elemento la si indica facendo seguire una virgola al suo unico valore, `('uno',)`.

```
>>> 1 # cosi' definiamo un intero
 1
>>> 1, # e cosi' una tupla di un solo elemento
 (1,)
>>> type((1,))
 <type 'tuple'>
```

Al pari delle stringhe, le tuple sono immutabili, ovvero non possiamo modificarne il contenuto. Questa caratteristica di immutabilità, che ad una prima analisi potrebbe sembrare limitante, vedremo che a volte si rende necessaria, come nel caso delle chiavi di un dizionario.

In Figura 3.1 è rappresentato il diagramma di stato della tupla `data_nascita` definita in precedenza. Da notare come la rappresentazione della tupla contenga in realtà dei riferimenti agli oggetti contenuti nella tupla invece che gli stessi oggetti. Questa rappresentazione rispetta fedelmente la rappresentazione in memoria usata dall' interprete Python.

3.4.2 Stringhe e tuple: operatori di formato

In Python disponiamo di un operatore per generare una stringa con un formato dato. Questo operatore, qui nominato *operatore di formato per le stringhe*, ha la sintassi

```
stringa_di_formato % tupla_argomenti
```

dove la stringa di formato contiene dei *segnaposto* nella forma `%<formato>` in cui verranno sostituiti i valori degli argomenti inclusi nella tupla nello stesso ordine. Vediamo subito un esempio di utilizzo.

```
>>> "ci sono %d numeri da %d a %d"%(2,2,4)
 'ci sono 2 numeri da 2 a 4'
```

In questo caso abbiamo indicato segnaposti usando il formato `%d` indicando così che i numeri inseriti sono di tipo intero. Ma è possibile anche indicare la presenza di altri tipi di formattazione tra i quali `%f` per i float, `%s` per le stringhe, o `%r` quando desideriamo che sia utilizzata la rappresentazione standard di Python per quell'oggetto. Per i tipi numerici possiamo anche indicare il numero di cifre della rappresentazione e il numero di cifre decimali.

Nella sessione seguente esemplifichiamo l'uso di alcuni tra i più utili descrittori di formato delle stringhe. Per una descrizione più dettagliata dei vari formati e delle opzioni disponibili si rimanda il lettore alla documentazione del linguaggio.

```
>>> "%c"%'a' # carattere
 'a'
>>> "%r"%(type(2)) # usa la rappresentazione data da repr()
 "<type 'int'>"
>>> "%s" % 3.23 # usa la rappresentazione data da str()
 '3.23'
>>> "%f" % -23.67 # float
 '-23.670000'
>>> "%d%f%%%d" % (1,3.7,-1) # %% per inserire il simbolo %
 '13.700000%-1'
>>> "%5s" % "dx" # giustifica a dx su 5 spazi
 '   dx'
>>> "%-7s" % "sx" # giustifica a sx su 7 spazi
 'sx     '
>>> "%10.7f" % 3.147 # ampiezza 10, numero cifre decimali 7
 ' 3.1470000'
```

3.4.3 Liste

Come la tupla, anche la lista è una sequenza di oggetti non necessariamente omogenei. La differenza principale tra queste due strutture dati è che la lista è un tipo mutabile.

Una lista si costruisce ponendo delle virgole tra i vari elementi, e racchiudendo il tutto tra parentesi quadrate. Anche qui, gli elementi possono essere indicati usando il corrispondente letterale o il risultato di una espressione.

Per creare una lista vuota possiamo usare la notazione `[]` oppure il costruttore di lista `list()`.

```
>>> lista_vuota=[]
>>> list()
 []
```

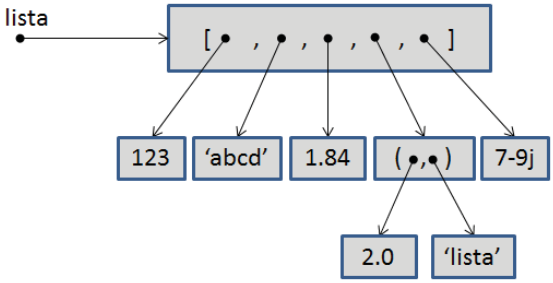


Figura 3.2: Diagramma di stato di una lista

Iniziamo col definire una semplice lista che contiene un insieme di numeri interi

```
>>> prima_lista=[2,3,4]
>>> prima_lista
 [2,3,4]
```

oppure con elementi eterogenei

```
>>> lista=[123, 'abcd', 1.84, (2.0,'lista'),7-9j]
```

In Figura 3.2 è rappresentato il diagramma di stato della lista appena definita. Si può notare come, mediante l' utilizzo dei riferimenti, i contenitori possono essere agevolmente annidati creando così strutture dati di complessità arbitraria.

Come abbiamo detto, le liste sono un tipo mutabile. Questo significa che il contenuto di un oggetto lista può cambiare. Di seguito vediamo alcuni dei metodi definiti sul tipo lista che ci permettono di manipolare oggetti di questo tipo.

```
>>> elle = ['a',1,('b',3+1j)] #lista  di esempio
>>> elle.append('nuovo') # aggiunge un elemento alla fine
>>> elle
 ['a',1,('b',3+1j),'nuovo']
>>> elle.extend([1,2]) # concatena un'altra lista alla fine
>>> elle
 ['a',1,('b',3+1j),'nuovo',1,2]
>>> elle.reverse() # inverte l'ordine degli elementi
>>> elle
 [2,1,'nuovo',('b',3+1j),1,'a']
>>> elle.insert(1,'inserito') # inserisce oggetto in un
                              # certo indice
>>> elle
 [2,'inserito',1,'nuovo',('b',3+1j),1,'a']
>>> elle.remove('nuovo') # elimina un elemento
>>> elle
 [2,'inserito',1,('b',3+1j),1,'a']
>>> elle.index('a') # ritorna l'indice corrispondente
```

```
                    # ad un elemento
 5
>>> del elle[5] # elimina l'elemento di indice 5
>>> elle
 [2,'inserito',1,('b',3+1j),1]
>>> elle.sort() # ordina gli elementi dal minore al maggiore
>>> elle
 [1, 1, 2, 'inserito', ('b', (3+1j))]
>>> el = elle.pop() # rimuove e ritorna ultimo elemento
                    # della lista
>>> el
 ('b', (3+1j))
```

Nella sessione appena vista abbiamo utilizzato il fatto che le liste sono caratterizzabili come sequenze e quindi ha senso indicare l'indice di un certo oggetto nel contenitore. Nota che l'indice corrisponde a `0` per il primo elemento della lista, `1` per il secondo, e così via. Per lo stesso motivo ha senso anche richiedere l'ordinamento dei valori nella sequenza.

3.4.4 Liste e stringhe: split() e join()

Un metodo delle stringhe piuttosto utile è `split()`. Questo metodo ritorna la lista delle parole presenti in una stringa. Come definire una parola è compito del programmatore. Per default le parole nella stringa sono separate da spazi o carattere di ritorno a capo. Possiamo comunque dare un parametro aggiuntivo per determinare un separatore di parole diverso.

Nel caso più semplice la sintassi di questo metodo è `s.split(<sep>)` dove `<sep>` è una stringa che funge da separatore e `s` è la stringa che vogliamo dividere. Quando il separatore non è indicato, i caratteri spazio (`' '`) e ritorno carrello (`'\n'`) sono utilizzati come separatori. Facciamo qualche esempio per chiarire il comportamento della `split()`.

```
>>> "Alla bisogna\n tango si balla\n".split()
 ['Alla', 'bisogna', 'tango', 'si', 'balla']
>>> "Alla bisogna\n tango si balla\n".split(" ")
 ['Alla', 'bisogna\n', 'tango', 'si', 'balla\n']
>>> "Alla bisogna\n tango si balla\n".split("\n")
 ['Alla bisogna', ' tango si balla', '']
```

La funzionalità inversa è ottenuta con il metodo delle stringhe `join()` che appunto serve per "unire" una lista di stringhe in un'unica stringa indicando eventualmente il separatore che si vuole utilizzare. Anche se può sembrare controintuitivo, il metodo `join()` è un metodo della classe `str` e va applicato alla stringa che contiene il separatore.

```
>>> "".join(['a','b','c']) # senza separatore
 'abc'
>>> " ".join(['a','b','c']) # separatore spazio
 'a b c'
```

```
>>> "\n".join(['a','b','c']) # separatore termine riga
 'a\nb\nc'
```

3.5 Operazioni sulle sequenze

Con le liste abbiamo terminato i tipi sequenza: `str`, `tuple` e `list`. Vediamo adesso un insieme di modi per manipolare sequenze.

3.5.1 Assegnamento in linea

L'assegmanento in linea di più variabili ci permette di assegnare oggetti a più riferimenti contemporaneamente. La sintassi dell'assegnamento multiplo è

```
RIF1, RIF2, ... , RIFn = SEQ
```

dove `RIFx` sono n nomi di riferimenti e `SEQ` è una sequenza contenente esattamente n espressioni. Vediamo qualche esempio di utilizzo.

```
>>> a,b,c = 1,2,3
>>> a,c = c,a # scambia i valori di a e c
>>> a,b,c = "abc"
>>> a,b,c,d = [1,-1,(4+2j,3),"aca"]
```

3.5.2 Concatenazione

Data una coppia di sequenze s_1 e s_2 dello stesso tipo T, possiamo generare una nuova sequenza s_3 di tipo T contenente la concatenazione di s_1 e s_2 con l'operatore $+$. Per esempio:

```
>>> s1,s2 = "esse1", "esse2"
>>> s1+s2
 'esse1esse2'
>>> t1,t2 = (1,2), (3,4,5)
>>> t1+t2
 (1,2,3,4,5)
```

3.5.3 Ripetizione

Data una sequenza s di tipo T e un intero n, possiamo generare una nuova sequenza s_1 di tipo T contenente n ripetizioni di s con l'operatore $\star$. Per esempio:

```
>>> s = "abc"
>>> s*3
 'abcabcabc"
>>> l = [1,2,3]
>>> l*3
 [1,2,3,1,2,3,1,2,3]
```

3.5.4 Indicizzazione

Data una sequenza s e un intero i, possiamo accedere al riferimento dell' $(i + 1)$-esimo elemento della sequenza con l'operatore `s[i]`. Per esempio:

```
>>> s = "abc"
>>> s[2]
 'c'
>>> l = [1,('a','b'),3]
>>> l[2]
 3
>>> l[1][1]
 'b'
```

3.5.5 Slicing e striding

Quando parliamo di *slicing*, o "affettamento", di una stringa o di una sequenza in generale, intendiamo un modo per estrarre una porzione della sequenza. La sintassi dello slicing è

```
seq[start:end]
```

dove `seq` è la sequenza che vogliamo "affettare", `start` è la posizione del primo elemento che vogliamo ottenere, `end` è la posizione successiva all'ultimo elemento che vogliamo ottenere, ovvero il primo elemento che *non* vogliamo ottenere. I valori `start` e `end` possono anche essere negativi. In questo caso, significa che l'enumerazione inizia dal fondo della lista. Per esempio, usando `stringa[-3:-1]`, si ottengono tutti i caratteri a partire dal terzultimo fino all'ultimo escluso. Quando non esplicitamente indicati, `start` ha valore di default uguale a 0 mentre `end` ha come valore di default il numero di elementi nella sequenza, ovvero consideriamo la fine della sequenza.

Ecco un esempio di interazione con slicing sulle stringhe. L'utilizzo con altri tipi di sequenze risulta intuitivo e viene lasciato come utile esercizio.

```
>>> stringa = "ciao a tutti!"
>>> stringa[1:4]
# ritorna una stringa con il secondo,
# il terzo e il quarto carattere
# della stringa originale
 'iao'
>>> stringa[1:]
# ritorna la sottostringa che parte
# dal secondo elemento fino alla fine
 'iao a tutti!'
>>> stringa[:10]
# ritorna la sottostringa formata
# dai primi 10 caratteri
 'ciao a tut'
>>> stringa[-3:]
# ritorna la sottostringa contenente gli ultimi 3 caratteri
 'ti!'
```

```
>>> stringa[-3:-1]
# ritorna la sottostringa contenente gli ultimi 3 caratteri
# meno l'ultimo
 'ti'
```

La sintassi dello *striding* (che potremmo tradurre con 'avanzamento') è

```
seq[start:end:step]
```

e differisce dallo slicing solo perchè si aggiunge il passo del movimento (lett. la falcata) sulla sequenza. Un passo positivo significa che vogliamo percorrere la sequenza nel senso naturale, da sinistra a destra; un passo negativo significa che vogliamo percorrere la sequenza in senso inverso. Se non indicato, il passo di default è inteso essere 1. Altrimenti, esso rappresenta il numero di caratteri di avanzamento nella scansione della sequenza. Vediamo qualche esempio finale da provare per chiarire il concetto di striding.

```
>>> a = "0123456789"
>>> a[:8:3] # '0123456789'[0:8:3]
 036
>>> a[3::3] # '0123456789'[3:10:3]
 369
>>> a[6:2:-1] # '0123456789'[6:2:-1]
 '6543'
>>> a[7:2:-2] # '0123456789'[7:2:-2]
 '753'
```

3.5.6 Modifica di sotto-sequenze

Per le sequenze mutabili, l'indicizzazione, lo slicing e lo striding, possono anche essere usati nella parte sinistra di un assegnamento per indicare una particolare sotto-sequenza che vogliamo modificare. Nella parte destra dello stesso assegnamento andremo ad indicare la sequenza da cui ottenere i valori da rimpiazzare. Nel caso dello striding, la sotto-sequenza estratta nella parte sinistra e la sequenza indicata nella parte destra dell'assegnamento devono necessariamente avere la stessa lunghezza. Per l'indicizzazione e lo slicing, invece, le due sequenze possono avere lunghezze diverse. Studiamo il seguente esempio:

```
>>> elle = [1,2,3,4,5]
# elle -> [1,2,3,4,5]
>>> elle[2]=6
# elle -> [1,2,6,4,5]
>>> elle[1:3]=[7,8,9]
# elle->[1,7,8,9,4,5]
>>> elle[1:5:2]=[10,11]
# elle->[1,10,8,11,4,5]
>>> elle[-1:-4:-2] = ('a','b')
# elle -> [1, 10, 8, 'b', 4, 'a']
>>> elle[1:-2] = 'cde'
# elle -> [1,'c','d','e',4,'a']
```

Possiamo anche usare il comando `del` per cancellare da una sequenza una determinata sotto-sequenza indicata mediante indicizzazione, slicing o striding.

```
>>> elle = [1,2,3,4,5,6]
>>> del elle[1:-4] # elle -> [1, 3, 4, 5, 6]
>>> del elle[2] # elle -> [1, 3, 5, 6]
>>> del elle[::2] # elle -> [3, 6]
```

3.6 Dizionari

Un dizionario è un contenitore di coppie: chiave (key), valore (value). La differenza principale rispetto alle liste e alle tuple è che esso non è ordinato come una sequenza di valori. La posizione di un elemento è irrilevante. Quello che conta è l'associazione con la sua chiave.

Per creare un dizionario vuoto si possono usare le parentesi graffe aperta e chiusa {}, oppure il suo costruttore `dict()`. Per crearlo con già qualche coppia al suo interno basterà raccogliere tra parentesi graffe le coppie da inserire nel formato `key:value` e le coppie separate da virgole.

```
>>> primo_vuoto = {}
>>> altro_vuoto = dict()
>>> uno_pieno = {1:"primo",2:"secondo",5:"quinto"}
```

Una struttura dati come un dizionario non sarebbe utile se non fosse possibile inserire nuove associazioni o modificare/eliminare quelle esistenti. Vediamo innanzitutto come sia facile inserire nuove coppie usando la sintassi `d[key]=value` dove `d` è il dizionario da modificare e `key`, `value` sono la chiave e il valore da inserire, rispettivamente.

```
>>> dizio= {}
>>> dizio["gianni"] = 37
>>> dizio["michele"] = 26
>>> dizio
 {'michele' : 26, 'gianni' : 37}
```

Per modificare il valore di un' associazione già esistente basterà sovrascrivere su quello vecchio.

```
>>> dizio= {'monica':'346-87123221','carolina':'330-2424244'}
>>> dizio["monica"] = '329-11198911'
>>> dizio
 {'monica': '329-11198911', 'carolina': '330-2424244'}
```

Per cancellare un elemento da un dizionario possiamo utilizzare il comando `del` nel modo seguente.

```
>>> dizio = {'a':1,'b':2,'c':3}
>>> del dizio['b']
>>> dizio
 {'a':1,'c':3}
```

Chiavi e valori inseriti in un dizionario possono essere eterogenei dal punto di vista del tipo. L' unica restrizione è che le chiavi devono essere di tipo immutabile e, nel caso esse siano annidate, anche tutti i tipi degli oggetti annidati dovranno essere a loro volta immutabili.

```
>>> {(1,'b'):[1,2]}     # valido
>>> {[1,'b']:[1,2]}     # non valido
>>> {(1,{2:3}):[1,2]}   # non valido
>>> {(1,(2,[])):[1,2]} # non valido
```

Terminiamo la nostra discussione sui dizionari presentando brevemente i metodi principali per ottenere la liste di chiavi e/o valori presenti nel dizionario e per eliminare tutte le voci presenti nel dizionario.

```
>>> d = {'k1':1,'k2':2}
>>> d.keys() # ottiene la lista delle chiavi
 ['k1','k2']
>>> d.values() # ottiene la lista dei valori
 [1,2]
>>> d.items() # lista di coppie chiave/valore
 [('k1',1),('k2',2)]
>>> d.clear() # rimuove tutte le coppie
>>> d
 {}
```

3.7 Insiemi

Un `set` è una struttura dati che rappresenta insiemi di oggetti non ordinati e senza duplicati. A differenza dei tipi di dati standard che abbiamo visto fino ad ora, l'insieme non ha un letterale specifico associato ma viene indicato con la parola chiave `set`. Non essendoci un particolare letterale, l'unico modo per costruire un insieme è utilizzare il costruttore indicando eventualmente una sequenza di oggetti da cui attingere per inizializzare l'insieme. Se la sequenza passata come parametro al costruttore contiene elementi duplicati, l'insieme generato sarà comunque composto dal solo insieme di oggetti unici presenti nella sequenza.

```
>>> insieme_vuoto = set()
>>> insieme_pieno = set(["lun","mar","mer"])
>>> set("byebyeguy")
 set(['y','u','b','e','g'])
```

Python definisce tutta una serie di operazioni sugli insiemi che corrispondono alle operazioni standard della teoria insiemistica. Di seguito mostriamo brevemente alcuni esempi d'uso di tali operatori.

```
>>> disp = set([1,3,5,7,9]) # num dispari
>>> pari = set([2,4,6,8]) # num pari
>>> disp | pari # unione
 set([1,2,3,4,5,6,7,8,9])
>>> mul3 = set([3,6,9]) # multipli di 3
```

```
>>> disp & mul3 # intersezione
 set([9,3])
>>> pari - mul3 # complemento (pari ma non multiplo di 3)
 set([8,2,4])
>>> disp ^ mul3 # diff simmetrica (in uno ma non nell'altro)
 set([7,5,6,1])
```

Infine, abbiamo metodi per l'inserimento e la rimozione di particolari elementi da un insieme.

```
>>> s = set([1,2,3,4,5])
>>> s.add('a') # aggiunge un elemento all'insieme
>>> s
 set(['a',1,2,3,4,5])
>>> s.remove(3) # rimuove un elemento dall'insieme
>>> s
 set(['a',1,2,4,5])
```

3.8 Operazioni su iterabili

Passiamo ora in rassegna un gruppo di operatori, funzioni e metodi che hanno in comune tutti i tipi iterabili.

3.8.1 Operatori su tipi iterabili

Possiamo verificare l'appartenenza o meno di un determinato oggetto ad un iterabile con l'operatore `in` e `not in`, rispettivamente.

```
>>> carte_in_mano = set([(3,'f'),(1,'q'),(1,'c'),(10,'f')])
>>> (3,'f') in carte_in_mano
 True
>>> (1,'p') in carte_in_mano
 False
>>> (1,'p') not in carte_in_mano
 True
```

3.8.2 Funzioni su tipi iterabili

Una funzione molto utilizzata in Python è la funzione `len()` che ritorna il numero di elementi contenuti in un iterabile (la sua lunghezza).

```
>>> studenti = ["gino","lino","pino","rino"]
>>> len(studenti)
 4
>>> matrice = [[1,2,3],[2,3,4],[3,4,5]]
>>> len(matrice) # numero degli elementi della lista esterna
 3
```

Nota che, nel caso di contenitori annidati, la lunghezza si riferisce al numero di elementi nel primo livello, contando ogni eventuale contenitore ivi contenuto come un singolo elemento.
Data una collezione di elementi numerici (interi, in virgola mobile o complessi), possiamo calcolare la somma degli elementi nella collezione con la funzione `sum()`.

```
>>> vettore = [-2.4,2.1,9.0,7.4]
>>> sum(vettore)
 16.1
```

Data una collezione di elementi su cui è definita una relazione d'ordine (<) è possibile anche calcolare altre statistiche come il minimo e il massimo.

```
>>> nomi = ["Fabio","Gianni","Alberto"]
>>> min(nomi)
 'Alberto'
>>> max(nomi)
 'Gianni'
```

Ancora, ammettendo che esista una relazione di ordine tra gli elementi della collezione, possiamo ottenere la lista degli elementi dell'iterabile ordinati utilizzando la funzione `sorted()`, eventualmente con l'opzione `reverse=True` se vogliamo ordinare dal maggiore al minore.

```
>>> nomi = ["Fabio","Gianni","Alberto"]
>>> nomi_in_ordine = sorted(nomi)
>>> nomi_in_ordine
  ['Alberto', 'Fabio', 'Gianni']
>>> nomi_in_ordine_inverso = sorted(nomi,reverse=True)
>>> nomi_in_ordine_inverso
 ['Gianni', 'Fabio', 'Alberto']
```

A volte può essere utile combinare in una lista i valori contenuti in due o più iterabili. Facciamo un semplice esempio: immaginiamo di avere una lista che contiene il nome di N studenti e un'altra lista che contiene il cognome degli stessi studenti. A partire da queste due liste vogliamo generare un'unica lista contenente N coppie dove ogni tupla contiene il nome e il cognome di uno studente. Per fare questo possiamo usare la funzione `zip()` di seguito esemplificata.

```
>>> nomi = ["Fabio","Gianni","Sara"]
>>> cognomi = ["Aiolli","Busco","Mori"]
>>> zip(nomi,cognomi)
  [('Fabio', 'Aiolli'), ('Gianni', 'Busco'), ('Sara','Mori')]
```

Possiamo notare che nella funzione `zip()` l'ordine degli elementi contenuti negli iterabili da combinare è effettivamente rilevante. Abbiamo però detto in precedenza che gli insiemi e i dizionari sono sì iterabili ma non sono sequenze, e quindi non hanno un ordine predefinito degli elementi. Per i dizionari e gli insiemi questa ambiguità viene risolta considerando la lista di chiavi del dizionario o degli elementi dell'insieme, rispettivamente. Nessuna assunzione può essere fatta sull'ordine degli elementi all'interno di queste liste. Qui sotto vediamo un

esempio assolutamente lecito dove vengono combinati un dizionario e un insieme, entrambi di lunghezza due.

```
>>> zip({1:'a','k':'b'},set(['s1','s2']))
 [(1, 's2'), ('k', 's1')]
```

Possiamo vedere che la lista prodotta contiene due coppie, ognuna costruita prendendo una chiave del dizionario unita ad un elemento dell'insieme. Possiamo notare anche che l'ordine di apparizione delle chiavi del dizionario e degli elementi dell'insieme è assolutamente arbitrario.

3.9 Descrittori di lista

Data una collezione iterabile è possibile costruire una lista avente valori calcolati come funzione dei valori nella collezione. La sintassi generale, nel caso più semplice, è la seguente:

```
[f(x) for x in IT]
```

dove `f()` è una espressione applicata ai valori `x` ottenuti via via dall'iteratore `IT`. Facciamo un esempio in cui vogliamo calcolare i quadrati di tutti i numeri da 0 a 9 in una lista ottenuta con la funzione `range()`. Avremo:

```
>>> [x*x for x in range(10)]
 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Come ulteriore opzione, è possibile anche indicare una condizione che devono soddisfare i valori ottenuti dalla collezione. La sintassi in questo caso è:

```
[f(x) for x in IT if g(x)]
```

la quale può essere letta come *"vogliamo tutti i valori `f(x)` per ogni `x` in `IT` per i quali l'espressione `g(x)` è vera"*. Riprendendo l'esempio precedente possiamo selezionare solo i quadrati dispari:

```
>>> [x*x for x in range(10) if x%2]
 [1, 9, 25, 49, 81]
```

L'espressione `f(x)` può essere a sua volta un descrittore di lista. Un tipico esempio dove torna utile annidare descrittori di lista è il calcolo della matrice trasposta di una matrice rappresentata come lista di liste (le liste interne rappresentano le righe della matrice). Immaginiamo di avere una matrice 3×4 (lista di 3 liste di lunghezza 4). Vogliamo calcolare la matrice trasposta, ovvero quella matrice con i valori "ribaltati" rispetto alla diagonale. Un modo per farlo è il seguente, che usa appunto i descrittori di lista.

```
>>> mat = [[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]]
>>> [[riga[i] for riga in mat] for i in range(4)]
 [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Il riferimento `riga` contiene ad ogni passo, una dopo l'altra, le righe di `mat`. Per ogni indice da 0 a 3 quindi accediamo a `riga[i]`, ovvero l'elemento nella colonna di indice i, per costruire le righe della matrice trasposta.

I descrittori di lista possono comprendere anche più di un iteratore (`for`). La sintassi nel caso di due iteratori è la seguente (per più di due iteratori è molto simile e lasciata al lettore come esercizio):

```
[f(x1,x2) for x1 in IT1 if g(x1)
          for x2 in IT2(x1) if h(x1,x2)]
```

la quale va letta come *"vogliamo tutti i valori `f(x1,x2)` per ogni `x1` in `IT1` per i quali l'espressione `g(x1)` è vera, per ogni `x2` in `IT2`, tali che l'espressione `h(x1,x2)` è vera"*. Nota che il contenuto dell'iterabile `IT2` può eventualmente dipendere dal valore assunto dalla variabile `x1`. Nel caso di più di due iteratori la stessa regola si estende facilmente. Il contenuto dell'iterabile i-esimo può dipendere dalle $i-1$ variabili quantificate in precedenza.

Come primo esempio di doppio iteratore utilizziamo il descrittore di lista per costruire una lista delle combinazioni di valori provenienti da due iterabili con la condizione che i valori siano diversi.

```
>>> [(x, y) for x in [-1,2] for y in [3,2] if x != y]
 [(-1, 3), (-1, 2), (2, 3)]
```

Immaginiamo ora di voler raccogliere in una singola lista tutti gli elementi riga per riga di una matrice. Nella soluzione qui sotto facciamo dipendere il secondo iterabile (`row`) dal contenuto del primo iterabile (`mat`).

```
>>> mat = [[1,2],[3,4]]
>>> [item for row in mat for item in row]
 [1,2,3,4]
```

3.10 Assegnamenti, operatori, metodi e funzioni

Finora abbiamo trattato varie tipologie di costrutti Python: assegnamenti, operatori, funzioni e metodi. Per comprendere bene il ruolo di ognuno di essi occorre approfondire due aspetti fondamentali che riguardano l'eventuale *valore ritornato* e l'*effetto collaterale* (o *side-effect*) del comando sullo stato del programma. Parliamo di effetto collaterale di un costrutto quando la valutazione di tale costrutto comporta una modifica nello stato del programma, tipicamente una modifica dei valori delle variabili.

Iniziamo con l'assegnamento. La sua sintassi è

```
<rif> = <exp>.
```

dove `<rif>` è sempre un riferimento (già esistente o meno) ad un oggetto e `<exp>` una qualsiasi espressione che ritorna un oggetto. L'assegnamento provoca sempre un effetto collaterale sullo stato del programma poiché il riferimento viene dirottato verso l'oggetto risultante dalla valutazione dell'espressione a destra. Questo fa sì che il valore delle variabili possa cambiare dopo l'esecuzione di un assegnamento. In Python, a differenza di altri linguaggi di programmazione,

l'assegnamento non ritorna alcun valore. Questo implica che è assolutamente sbagliato usare un assegnamento laddove sia richiesta una espressione.

A differenza dell'assegnamento, la valutazione di operatori, metodi e funzioni produce *sempre* un oggetto che a sua volta potrà essere utilizzato all'interno di altre espressioni o a destra di un assegnamento.

Gli operatori si dividono in *operatori unari* del tipo `op obj`, e *operatori binari* del tipo `obj1 op obj2`. Una regola generale è che l'applicazione di un operatore non ha alcuna ripercussione sul valore degli operandi, quindi non produce effetti collaterali. Prendiamo il caso dell'operatore binario `+` per le liste. Siano `l1` e `l2` due liste, l'operatore `l1 + l2` restituisce un *nuovo* oggetto che contiene la concatenazione delle due liste ma le due liste su cui ha operato rimangono invariate. Quello che succede nella realtà in questo caso è che il nuovo oggetto lista prodotto conterrà delle copie degli stessi riferimenti che erano presenti negli operandi. Prendiamo la seguente sessione:

```
>>> l1 = [1,2,3]
>>> l2 = [4,5,6]
>>> l3 = l1 + l2
>>> l3 # l3 riferisce un nuovo oggetto indipendente da l1 e l2
 [1,2,3,4,5,6]
>>> l3[0]=0 # non ha ripercussioni su l1 e l2
>>> l1,l2
 ([1,2,3], [4,5,6])
```

Notare che la modifica della lista `l3` non provoca alcun effetto sulle liste `l1` e `l2`.

Le funzioni e i metodi (funzioni associate ad oggetti di un determinato tipo) sono applicati ad uno o più oggetti usando la sintassi

`nome_fun(<parametri>)` **o** `obj.nome_met(<parametri>)`.

Come per gli operatori, l'applicazione di una funzione o metodo ritorna sempre un oggetto, eventualmente uguale a `None`. A differenza degli operatori, però, l'applicazione di una funzione può provocare anche effetti collaterali.

Riprendiamo il caso studiato precedentemente, e vediamo come si comporta il metodo `extend()` delle liste. Volendo ottenere una situazione simile a quella vista per l'operatore concatenazione, possiamo provare ad utilizzare tale metodo come nella sessione seguente.

```
>>> l1 = [1,2,3]
>>> l2 = [4,5,6]
>>> l3 = l1.extend(l2)
>>> l1 # l1 e' stata modificata
 [1,2,3,4,5,6]
>>> l3 # e l3 adesso vale None
 None
```

In questo caso, `l1` è stata modificata nella concatenazione di `l1` e `l2` ma `l3` riferisce invece all'oggetto ritornato dal metodo `extend()`, che è `None`!

Esercizi del capitolo

Esercizio 3.1 (*)

Sia data una stringa `s`*, calcolare una lista con le frequenze delle vocali in* `s` *utilizzando i descrittori di lista.*

Soluzione:

```
>>> [(v,s.count(v)) for v in "aeiou"]
```

Esercizio 3.2 (*)

Il cifrario di Cesare *è un antico algoritmo crittografico. Ogni lettera del testo in chiaro è sostituita nel testo cifrato dalla lettera che si trova un certo numero di posizioni (*`k`*) successive nell'alfabeto. Per esempio la cifratura della frase* `'ave cesare'` *con* $k = 3$ *diventa* `'dyh fhvduh'`*.*

Dare una soluzione utilizzando un unico descrittore di lista per le seguenti:

1. *Codifica di una parola nel cifrario di Cesare*
2. *Decodifica di una parola cifrata*
3. *Codifica di una intera stringa nel cifrario di Cesare (si suppone che la stringa sia composta esclusivamente da parole formate da lettere e separate da spazi)*
4. *Decodifica di una intera frase (stesse assunzioni del punto sopra)*

Soluzione:

```
>>> cod = "".join([chr(ord('a')+(ord(c)-ord('a')+k)%26)
                for c in s])
```

```
>>> dec = "".join([chr(ord('a')+(ord(c)-ord('a')-k)%26)
                for c in s])
```

```
>>> cod_frase = " ".join(
  ["".join([chr(ord('a')+(ord(c)-ord('a')+k)%26)
  for c in p]) for p in s.split()])
```

```
>>> cod_frase = " ".join(
  ["".join([chr(ord('a')+(ord(c)-ord('a')-k)%26)
  for c in p]) for p in s.split()])
```

Esercizio 3.3 (*)

Date due liste `A`,`B` *di numeri. Dare un unico descrittore di lista che produce la matrice dei prodotti.*

Esempio: *Siano* `A=[1,2,3]` *e* `B=[5,10,15,20]`*. La lista prodotta dovrà essere* `[[5,10,15,20],[10,20,30,40],[15,30,45,60]]`*.*

Soluzione:

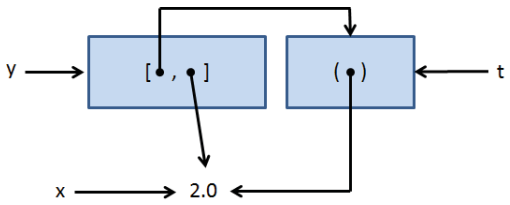


Figura 3.3: Diagramma di stato dell' esercizio 3.6

```
>>> [[x*y for y in B] for x in A]
```

Esercizio 3.4

Data la seguente sessione, sostituire il simbolo `?` *con il valore corretto.*

```
>>> a=['a','b',['b','c'],1,2,3]
>>> del a[0]
>>> a[1][0]='a'
>>> c=a[2:4]
>>> d=a[1]
>>> e=c+d
>>> e
 ?
```

Esercizio 3.5

Data la lista `L = [1,2,3,4,5]`*. Modificare* `L` *in modo tale che diventi la lista* `[3,4,6,7,5,0]` *utilizzando il minor numero possibile di metodi per le liste visti in questo capitolo.*

Esercizio 3.6

Indicare la breve sessione interattiva (3 comandi) che ci permette di ottenere il diagramma di stato in Figura 3.3. La soluzione può considerarsi corretta se `x==2.0` *(tipo float) e l' espressione* `x is t[0] is y[0][0]` *restituiscono entrambe* `True`*.*

Esercizio 3.7

Sia data la seguente affermazione: "Non è possibile cambiare il valore di un oggetto di tipo immutabile"*. Dire se tale affermazione è sempre vera alla luce dell'esempio che segue.*

```
>>> x = [1,2]
>>> t = (x,1)
>>> x.append(3)
>>> t
 ?
```

Provare a disegnare il diagramma di stato per l'esempio precedente. L' oggetto riferito da `t` *è stato realmente modificato? Dare infine un'affermazione che modifichi il meno possibile quella precedente e che però sia sempre vera.*

Esercizio 3.8 (*)

Dati N (un numero intero) e L (una lista di interi) si dia un unico descrittore di lista che produce la lista di tuple con numero (cardinalità) e somma dei multipli in L per ogni intero n tale che $2 \leq n \leq N$.

Esempio: Per $N = 3$, $L = [4, 6, 9, 10, 11, 12]$, la lista prodotta deve essere $[(4, 32), (3, 27)]$.

Soluzione:

```
[(len(q),sum(q))
  for q in [x for x in L if x%n==0]
  for n in range(2,N+1)]
```

Esercizio 3.9 (*)

Si dia un unico descrittore di lista che produce la lista di terne di numeri interi compresi tra 0 e 10 con somma 10. Evitare duplicazioni di triple. P.e. $(2, 5, 3)$ e $(3, 5, 2)$ NON *dovrebbero comparire insieme.*

Suggerimento: *notare che per evitare la duplicazione possiamo considerare solo le terne ordinate.*

Soluzione:

```
N = 10
[(a,b,N-a-b) for a in range(N+1) for b in range(a,N+1-a)
                                                 if 2*b <= (N-a)]
```

Esercizio 3.10 (*)

Dato $N \geq 3$, si dia un unico descrittore di lista che produce la lista di terne di stringhe della forma `'a..ab..bc..c'`, ovvero con un numero $0 < N_a < N$ di caratteri `'a'`, un numero $0 < N_b < N$ di caratteri `'b'`, un numero $0 < N_c < N$ di caratteri `'c'`, tali che $N_a + N_b + N_c = N$.

Soluzione:

```
['a'*na+'b'*nb+'c'*nc for na in range(1,N)
                      for nb in range(1,N)
                      for nc in range(1,N)
                      if na+nb+nc==N]
['a'*na+'b'*nb+'c'*(N-na-nb) for na in range(1,N)
                             for nb in range(1,N-na)]
```

Esercizio 3.11 (*)

Date 2 liste A,B di numeri, dare un unico descrittore di lista che produce la matrice dei prodotti.

Esempio: Siano `A=[1,2,3]`, `B=[5,10,15,20]`. La lista prodotta dovrà essere: `[[5,10,15,20],[10,20,30,40],[15,30,45,60]]`

Soluzione:

```
[[a*b for b in B] for a in A]
```

Esercizio 3.12

Dati i seguenti assegnamenti:

```
x = 1.3
l = [7,1,14]
t = ("my",x,l)
```

determinare quali tra le seguenti serie di istruzioni

```
a)
   a = t[:2]+l[:]
   a[-1][2]=16
b)
   a = t[:]
   a[-1][2] = 16
c)
   a = t[:2]+(l[:],)
   a[-1][-1]+=2
d)
   a = t[:]
   a[-1]=l[:]
   a[-1][2]=16
```

verificano:

```
a -> ("my",1.3, [7,1,16])
t -> ("my",1.3,[7,1,14])
```

Esercizio 3.13

Determinare i valori di `a,b,c,d,e` *dopo i seguenti comandi:*

```
a = 'a'
b = 'b'
c = ({1:1,2:2},3)
d = [a,b,c]
d[0]='h'
c += (4,)
e = d[:]
e[2][0][1]=a
e[2][0][3]='a'
e[1] += 'b'
b = e[2][0][2]+3
```

4

Controllo del flusso

Di norma, il flusso del programma procede sequenzialmente seguendo l'ordine delle istruzioni contenuto in uno script. Se un linguaggio di programmazione non prevedesse un modo per poter variare dinamicamente (cioè a tempo di esecuzione) il flusso del programma, allora tale linguaggio di programmazione permetterebbe di creare solo programmi molto semplici e poco flessibili, comunque non capaci di risolvere l'enorme varietà di problemi che si possono presentare. A questo scopo, nei comuni linguaggi di programmazione, sono sempre presenti dei costrutti che permettono di variare il flusso di un programma a seconda dello stato in cui si trova in un certo momento della sua esecuzione. Questo viene fatto, tipicamente, testando il valore di verità (vero o falso) di un espressione che coinvolge il valore di una o più variabili.

In questo capitolo discuteremo delle caratteristiche fondamentali di un linguaggio di programmazione che ci permettono di gestire il controllo del flusso di un programma: la selezione e l'iterazione.

4.1 Selezione

Una caratteristica presente in tutti i linguaggi di programmazione è la selezione che ci permette di modificare il normale flusso sequenziale di un programma a seconda della valutazione di una certa condizione sullo stato del programma. Consideriamo i due semplici problemi qui sotto dove è richiesta la selezione:

Discoteca per maggiorenni. Vogliamo realizzare un programma per verificare se un ragazzo possa o meno entrare in una discoteca aperta solo a maggiorenni. Possiamo assumere di avere una variabile intera (`eta`) che contiene l'età del giovane.

Tasso di sconto. Il Sig. Guido, responsabile del settore commerciale dell'azienda di trasporti Piano S.p.A., vorrebbe automatizzare il calcolo del tasso di sconto da applicare ai clienti sulla base dell'importo dovuto (che assumiamo contenuto in una variabile). Se l'importo è maggiore di 100 euro, il tasso di sconto da applicare sarà del 15%, altrimenti del 10%.

In entrambi gli esempi ci viene richiesto di fare cose diverse a seconda dello stato di alcune variabili del programma. Nel primo esempio, dobbiamo presentare un

messaggio di conferma se il giovane ha già compiuto la maggiore età (`eta>=18`), o al contrario un messaggio che vieti l'ingresso in discoteca se tale condizione non è verificata. Nel secondo esempio, dobbiamo assegnare al valore di una variabile intera `sconto` il valore 10 o 15 dipendentemente dall'importo dell'ordine del cliente.

4.2 Iterazione

Un'altra caratteristica presente nella maggior parte dei linguaggi di programmazione è l'iterazione, ovvero la capacità di un linguaggio di esprimere la ripetizione di un insieme di comandi a seconda della valutazione di una certa condizione. Consideriamo un altro paio di semplici esempi dove è richiesta l'iterazione:

Tabella quadrati. Vogliamo definire un programma che, dato un qualsiasi valore intero positivo n, stampi in forma tabellare le coppie di valori i e i^2, per $i = 1, \ldots, n$.

Media voti studenti. Vogliamo leggere da input la serie di voti (in trentesimi) degli esami sostenuti da uno certo studente e poi calcolarne la media. Il numero di esami sostenuti non è conosciuto a priori. Per segnalare il termine dell'inserimento dei voti l'utente inserisce il valore 0 (che chiaramente non fa media).

Per quanto riguarda il primo problema proposto abbiamo già imparato come è possibile stampare a video la coppia di valori i e i^2 una volta fissato il valore i. Intuiamo però che un inserimento manuale delle stampe di tutte le coppie di numeri sarebbe alquanto scomodo per n grandi. Tale inserimento sarebbe addirittura impossibile, al tempo di scrittura del programma, se il valore n intendiamo leggerlo dinamicamente dall'utente, durante l'esecuzione del programma stesso. Ecco che l'iterazione ci dà la possibilità di esprimere in un linguaggio di programmazione il fatto di dover ripetere zero, una o più volte una determinata sequenza di comandi.

Nel secondo problema, invece, l'iterazione ci dà la possibilità di esprimere in un linguaggio di programmazione il fatto di dover ripetere la richiesta di inserimento di un nuovo voto (e eventualmente i calcoli necessari al fine del computo della media) fino a che l'utente non immetta un valore uguale a 0.

4.3 Espressioni a valori booleani

Prima di addentrarci nei costrutti veri e propri che implementano la selezione e la iterazione abbiamo bisogno di capire come poter valutare lo stato del programma in fase di esecuzione in modo da prendere dinamicamente delle decisioni. In questa sezione discutiamo i modi con i quali è possibile creare espressioni da utilizzare per ottenere valori di verità (vero o falso) in Python e quindi utilizzabili per modificare il flusso del programma. In particolare vedremo: *espressioni basiche*, *identità*, *comparazione*, *appartenenza*, *connettivi logici* e *espressioni condizionali*.

4.3.1 Espressioni basiche

Il modo più semplice di definire una espressione è indicare un singolo oggetto o riferimento. Chiameremo questo tipo di espressione *basica*. La valutazione come valore di verità di un'espressione di questo tipo è banalmente la valutazione come valore di verità dell'oggetto stesso.

Ogni oggetto di tipo predefinito in Python ammette una propria valutazione come valore di verità. Per ogni tipo predefinito esiste una particolare istanza di quel tipo, l'*istanza nulla*, che viene valutata a `False`. Ogni altro valore dello stesso tipo verrà invece valutato a `True`. Di seguito viene presentato un elenco delle istanze nulle per i vari tipi predefiniti:

- L'oggetto `None` di tipo `NoneType`
- Il booleano `False`
- Gli zeri degli oggetti numerici: `0,0L,0.0,0.0j`
- Ogni iterabile vuoto: `'',(),[],{},set([])`

Chiaramente la valutazione come valore di verità non si limita ai tipi di base ma bensì è disponibile anche per molti altri tipi definiti in Python. Per non appesantire la lettura consigliamo al lettore che ne volesse sapere di più di consultare il manuale di riferimento del linguaggio.

4.3.2 Identità

Abbiamo già incontrato l'operatore `is` che ci permette di verificare quando due oggetti in realtà sono lo stesso oggetto, ovvero hanno la stessa identità. Le espressioni costruite utilizzando questo operatore, ritornano un valore di verità.

```
>>> x,y = 1,1 # due interi uguali..
>>> x is y    # sono in realta' lo stesso oggetto!
 True
>>> x,y = 3.14,3.14 # due float uguali..
>>> x is y          # invece non sono lo stesso oggetto
 False
>>> [] is []   # i tipi mutabili: sempre oggetti diversi
 False
>>> x = [-1,1] # x ora riferisce l'oggetto lista [-1,1]
>>> y = x      # e ora anche y
>>> x is y     # quindi i due riferiscono lo stesso oggetto
 True
```

La semantica dell'operatore `is` è un po' ambigua quando vengono comparati oggetti di tipo immutabile aventi medesimo valore. In questo caso è possibile che l'interprete Python ottimizzi la memoria creando un solo oggetto per più riferimenti. Questo comportamento può apparire strano, e in effetti lo è. L'interprete Python riconosce i due oggetti come aventi lo stesso valore e, poiché immutabili, può contare sul fatto che tali oggetti manterranno lo stesso valore per tutta la loro

esistenza. A seconda della particolare implementazione, l'interprete Python può essere progettato in modo da creare una sola copia fisica in memoria (attenzione, non c'è una vera e propria regola). Dato che il comportamento potrebbe non essere uniforme su diverse piattaforme o diverse versioni del linguaggio, è sempre meglio evitare l'utilizzo dell'operatore di identità quando si ha a che fare con oggetti immutabili. Nel caso di tipi mutabili non abbiamo lo stesso problema di ambiguità; per i mutabili, ogni volta che un nuovo oggetto viene creato, gli viene assegnata una identità distinta.

Per concludere, il negato del comparatore `is` è `is not`:

```
>>> 3 is not 3
 False
>>> [1,2] is not [1,2]
 True
```

4.3.3 Comparazione

Le operazioni di comparazione servono per confrontare due oggetti dal punto di vista quantitativo. In Python abbiamo i seguenti operatori di comparazione: < (minore), > (maggiore), == (uguale), != (diverso), <= (minore o uguale) e >= (maggiore o uguale). Vediamo subito qualche semplice esempio del loro utilizzo.

```
>>> 3.59 <= 4.36      # ordinamento dei numeri reali
 True
>>> "caino" <= "abele" # ordinamento lessicografico
 False
>>> [1,2] < [2]       # comparazione di due liste
 True
>>> set(['a','b','c']) == set("abcbca")
 True
```

Bisogna notare in questo caso che, confrontando tipi diversi, l'interpretazione dell'operatore di comparazione cambia. Per esempio, quando consideriamo numeri reali (interi e in virgola mobile), viene utilizzata la relazione di ordinamento naturale. Per le stringhe, viene utilizzata la relazione di ordinamento lessicografico (o alfabetico). Per liste e le tuple, l'ordinamento segue la regola seguente: se hanno lunghezza diversa allora l'ordine è definito rispetto alla loro lunghezza, altrimenti l'ordine è definito rispetto al primi due valori che differiscono nelle due liste (tuple). Per i dizionari e gli insiemi la modalità di comparazione effettuata è decisamente più complessa e preferiamo non discuterla in questo testo. Curiosamente Python definisce un ordine anche per oggetti di tipo diverso (non entrambi numerici chiaramente) il quale è basato semplicemente sull'ordine lessicografico del nome del loro tipo, per esempio `3 < [1,2]` ma *solo* perché `'int'<'list'`.

4.3.4 Appartenenza

Un'altra operazione di tipo logico disponibile in Python è la verifica di appartenenza di un oggetto ad una collezione (stringa, lista, tupla, dizionario, insieme).

Essendo anche questa operazione piuttosto intuitiva, ci basterà qualche semplice esempio di utilizzo per comprenderne il funzionamento.

```
>>> paperopoli = ['pippo','pluto','paperino','paperone']
>>> 'nonnapapera' in paperopoli
 False
>>> 'pluto' in paperopoli
 True
>>> gruppi_preferiti = set(["U2", "Marillion", "Iron Maiden"])
>>> U2 in gruppi_preferiti
 True
>>> "Moda'" in gruppi_preferiti
 False
```

4.3.5 Connettivi logici

I connettivi logici sono gli operatori binari `and`, `or` e l'operatore unario di negazione `not`. Gli operandi possono essere oggetti di qualunque tipo, anche di tipo diverso tra loro. In realtà, i connettivi `and` e `or` in Python non ritornano necessariamente un valore booleano, come evidenzieremo negli esempi seguenti, ma quello che è importante è che l'oggetto ritornato viene valutato correttamente come un booleano quando richiesto utilizzando le semplici regole date precedentemente per le espressioni basiche.

La valutazione dei connettivi logici binari, similmente ad altri linguaggi di programmazione, viene eseguita con la tecnica cosiddetta *short-circuit* (corto-circuito). In pratica, la seconda delle due espressioni coinvolte viene effettivamente valutata solo nel caso in cui tale valutazione risulti strettamente necessaria al fine di calcolare il valore risultato di quel particolare connettivo logico. Per esempio, nel connettivo `and`, se la prima espressione risulta falsa, allora non è necessario valutare anche la seconda, e possiamo già concludere che l'intera espressione è falsa nel suo complesso. Viceversa, per la `or`, se la prima espressione risulta vera, non è necessario valutare anche la seconda espressione e possiamo già concludere che l'intera espressione risulterà vera. Vediamo ancora più nel dettaglio come procede la valutazione nei due casi descritti:

- `A and B`: prima viene valutata l'espressione `A`. Se questa è falsa o valutabile a `False`, allora l'espressione `A and B` ritorna l'oggetto risultante della valutazione di `A` e l'espressione `B` non viene in realtà valutata, altrimenti viene ritornata la valutazione dell'espressione in `B`.
- `A or B`: prima viene valutata l'espressione `A`. Se questa è vera, o valutabile a `True`, allora l'espressione `A or B` ritorna l'oggetto risultante della valutazione di `A` e l'espressione `B` non viene in realtà valutata, altrimenti viene ritornata la valutazione dell'espressione in `B`.

Passiamo quindi a mostrare alcuni esempi riassuntivi per comprendere meglio la tecnica short-circuit e la valutazione dei connettivi logici.

```
>>> 2 and 3
 3
>>> 2 and 0
 0
>>> "buono" or "cattivo"
 'buono'
>>> not ("cattivo" or "buono")
 False
>>> 'uno' or 1
 'uno'
>>> [] or [1,2]
 [1,2]
```

La precedenza per i connettivi logici segue la regola `not > and > or`. In ogni caso, l'utilizzo delle parentesi è sempre consigliabile per rendere il codice più leggibile. Alcuni esempi:

```
>>> 2 and 3 or 'si' # equivale a ((2 and 3) or 'si')
 3
>>> not 3 or [] and 2 # equivale a ((not 3) or ([] and 2))
 []
```

4.3.6 Espressioni condizionali

In Python esiste una particolare espressione, detta espressione condizionale, con la seguente sintassi:

```
EXPT if COND else EXPF
```

dove `EXPT` e `EXPF` sono a loro volta espressioni e `COND` è una condizione booleana. La valutazione verrà effettuata nel seguente modo. Prima viene valutata la condizione; se tale condizione risulta vera, allora viene valutata l'espressione `EXPT` e ritornato il suo valore, altrimenti viene ritornato il valore ottenuto dalla valutazione di `EXPF`.

Le espressioni condizionali risultano molto utili quando usate all'interno dei descrittori di lista. Per esempio, il seguente comando, data una lista `L`, ritorna la lista dei valori assoluti degli elementi di `L`.

```
>>> L = [-3,7.3,-1,-5,0.0,4]
>>> [x if x>0 else -x for x in L]
 [3,7.3,1,5,0.0,4]
```

Occorre precisare che l'operazione qui descritta è ben diversa da usare la condizione nei descrittori di lista come visto in Sezione 3.9. Nel caso attuale vogliamo inserire valori alternativi a seconda dell'elemento via via considerato. Nell'altro caso, ricordiamo, la condizione veniva utilizzata per decidere *se* l'elemento andasse considerato o meno.

4.4 Costrutto di selezione (if, elif, else)

Eccoci dunque arrivati a trattare il costrutto per la selezione. In Python, come in molti altri linguaggi di programmazione, esiste il costrutto di selezione `if` che, nel caso più semplice, ha la seguente sintassi:

```
if COND :
   BLOCCO
```

dove `COND` è un'espressione booleana e `BLOCCO` è il blocco di una o più istruzioni (a volte denominato *suite* o *corpo* dell'if), da eseguire solo se l'espressione `COND` risulta vera.

Opzionalmente, possiamo indicare anche il comportamento nel caso in cui la condizione `COND` risulti falsa aggiungendo il costrutto `else`, ovvero:

```
if COND :
   BLOCCO_SE_VERA
else :
   BLOCCO_SE_FALSA
```

Infine, se vogliamo considerare più di due ramificazioni, usiamo `elif` come descritto di seguito.

```
if COND_A :
   BLOCCO_SE_A_VERA
elif COND_B :
   BLOCCO_SE_A_FALSA_MA_B_VERA
...
else :
   BLOCCO_SE_COND_PRECEDENTI_TUTTE_FALSE
```

Notare come i costrutti `if`, `else`, `elif`, richiedano i due punti (`:`) per indicare che dalla riga seguente inizia il blocco di istruzioni relativo. Ogni blocco più interno in Python deve essere *indentato* (ovvero, orizzontalmente spostato verso destra) rispetto al relativo costrutto. Generalmente, ogni livello di indentazione consiste di 3 o 4 spazi. Ancora, ogni istruzione dello stesso livello in un blocco deve avere la stessa indentazione.

Per esemplificare l'utilizzo del costrutto di selezione, riprendiamo l'esempio sul tasso di sconto descritto precedentemente. Verosimilmente la parte riguardante il calcolo del tasso di sconto si tradurrà in qualcosa di questo tipo:

```
if importo >= 100 :
   sconto = 15
else :
   sconto = 10
```

Esercizio risolto. Il seguente programma richiede in input il genere (M o F) e l'età x di una persona e stampa a video la frase corretta tra le due frasi:

- sei un maschietto e hai x anni
- sei una femminuccia e hai x anni

Script 4.1: mf_eta.py

```
genere = raw_input("Inserire il genere:")
eta = int( raw_input("Inserire l'eta:") )
if genere in ('M','m'):
   print "sei un maschietto e hai %d anni"%eta
elif genere in ('F','f'):
   print "sei una femminuccia e hai %d anni"%eta
else:
   print "i dati immessi sono errati!"
```

Le prime due righe servono per leggere da tastiera il genere inserito come una stringa di un solo carattere che ne rappresenta l'iniziale, maiuscola o minuscola non ha importanza, e l'età di tipo intero (notare la conversione stringa-intero). Poi, possono verificarsi tre casi diversi. Nel primo caso, il genere immesso è maschile (uno tra `'M'` e `'m'`) e viene stampata la stringa corrispondente. Nel secondo caso (ramo `elif`), controlliamo se il genere è femminile ed eventualmente stampiamo il messaggio appropriato. Infine, nel ramo `else`, se nessuno dei due casi precedenti si è verificato, allora stampiamo un messaggio di errore.

Esercizio risolto. *Il seguente programma richiede in input una operazione su due numeri (con operandi e operatore separati da spazio), per esempio* $3.2 + 4.7$*, e ne ritorna il risultato. Le operazioni disponibili sono l'addizione, la sottrazione, la moltiplicazione e la divisione. Da notare la gestione del caso della divisione dove viene anche effettuata una verifica che il denominatore non sia nullo. Nel caso il programma avviserà l'utente con un messaggio d'errore.*

Script 4.2: calcolatrice.py

```
# exp: stringa che contiene l'espressione inserita
exp = raw_input("Digita l'espressione: ")
(x,op,y) = exp.split()  # prendo i valori dentro l'
   espressione
x,y = float(x),float(y) # str -> float
if op=='+':
   print x + y
elif op=='-':
   print x - y
elif op=='*':
   print x * y
```

```
11 elif op=='/':
12    if y==0:
13       print "Errore: divisione per zero!"
14    else:
15       print x / y
```

Esercizio risolto. *Dare un programma che implementi il seguente gioco con l'utente. Il programma sceglie un numero compreso tra 1 e 5. L'utente prova ad indovinarlo. Se l'utente indovina al primo colpo vince, altrimenti il computer dà un'altra possibilità all'utente specificando se il numero indicato dall'utente era minore o maggiore di quello "pensato". Se l'utente indovina al secondo tentativo vince, altrimenti vince il computer.*

In questo programma utilizzeremo la funzione `randint()` del modulo `random` per generare un numero casuale compreso tra 1 e 5 (riga 3).

Script 4.3: gioco.py

```
1  import random
2
3  numero_pensato = random.randint(1,5)
4  numero = int(raw_input("Ho pensato il numero. Adesso tocca a
       te!\n"))
5
6  if numero==numero_pensato:
7     print "Esatto! Hai vinto :("
8  else:
9     print "Sbagliato!\nIl tuo numero e' "\
10      + ("maggiore" if (numero>numero_pensato) else "minore")\
11      + " del mio.\n"
12    numero = int(raw_input("Adesso tocca ancora te!\n"))
13    if (numero==numero_pensato):
14       print "Ok hai vinto tu."
15    else:
16       print "Ho vinto io. Il numero era %d."%numero_pensato
```

Notare come in linea 10 del codice si usa l'espressione condizionale vista in Sezione 4.3.6 per ottenere la stringa `"maggiore"` quando `numero>numero_pensato` o la stringa `"minore"` altrimenti.

4.5 Cicli (while)

Un altro tipo di costrutti alla base di ogni linguaggio di programmazione sono i costrutti per l'iterazione. In questa sezione tratteremo il costrutto di iterazione condizionale presente in Python, ovvero il costrutto `while`, avente sintassi:

```
while COND :
    BLOCCO
```

Anche in questo caso, tutte le istruzioni appartenenti al blocco del `while` vanno opportunamente indentate.

Il costrutto di iterazione condizionale risulta particolarmente utile quando vogliamo ripetere una serie di azioni nel programma mentre una qualche condizione si mantiene vera. Il flusso del programma in presenza di un ciclo while è il seguente:

[*step_cond*] Viene testata la condizione `COND`. Se questa viene valutata falsa, si esce dal ciclo e il flusso prosegue dall'istruzione successiva al while. Se invece viene valutata vera, il flusso prosegue con **[*step_itera*]**.

[*step_itera*] Esegue tutto il blocco interno al while. Alla fine di tale esecuzione il flusso prosegue con **[*step_cond*]**.

Il tipico schema di programmazione iterativa condizionale è:

- inizializza variabili V presenti in `COND`
- fino a che `COND` rimane vera (`while COND:`)
  - fai qualcosa (eventualmente usando V)
  - modifica variabili V

Presentiamo subito una serie di esempi elementari dove applichiamo il costrutto di iterazione condizionale. Tutti i casi proposti seguono lo schema di programmazione iterativa condizionale descritto sopra. Il lettore è caldamente invitato a riconoscerne i vari passi.

Esercizio risolto. *Scrivere un programma che legga un intero da input standard e lo stampi sullo standard output fino a quando non legge il numero* 5.

Script 4.4: fino_a_cinque.py

```
1 num = int(raw_input("Inserire un numero intero: "))
2 while num!=5:
3     print "Valore: %d" % num
4     num = int(raw_input("Inserire un numero intero: "))
```

Esercizio risolto *Scrivere un programma che, data una stringa di soli caratteri alfabetici minuscoli, verifichi se è composta da sole consonanti.*

Script 4.5: solo_consonanti.py

```
s = "bcdfg" # inizializzazione stringa
solo_consonanti = True
i,n = 0,len(s)      # indice e lunghezza della stringa
while i < n and solo_consonanti:
   if s[i] in "aeiou":
      solo_consonanti = False
   else:
      i += 1
```

In questo esercizio si notano cose interessanti. Un algoritmo piuttosto generale per verificare che una condizione sia sempre vera (*congiunzione*) per una serie di generici oggetti è il seguente. Assumiamo inizialmente che la condizione sia vera inizializzando una variabile `condizione` a `True`. Poi, per ogni elemento della serie, controlliamo la condizione. Se rimane vera, semplicemente andiamo avanti; altrimenti, modifichiamo il valore della variabile `condizione` a `False` (abbiamo trovato il contro-esempio). Per avere un comportamento simile allo short-circuit dei connettivi logici, conviene mettere in *and* la variabile `condizione` nella espressione condizionale del ciclo in modo tale da fermare il ciclo non appena venga individuato un contro-esempio.

Esercizio risolto. *Scrivere un programma che, data una stringa di soli caratteri alfabetici minuscoli, verifichi se all'interno c'è almeno una vocale.*

Seguendo un ragionamento del tutto simile a quello fatto nell'esercizio precedente, ma questa volta su una *disgiunzione* di eventi, otteniamo la seguente soluzione.

Script 4.6: almeno_una_vocale.py

```
s = "bcafg" # inizializzazione stringa
una_vocale = False
i,n = 0,len(s)      # indice e lunghezza della stringa
while i < n and not una_vocale:
   if s[i] in "aeiou":
      una_vocale = True
   else:
      i += 1
```

Nota comunque che la specifica data è equivalente a quella dell'esercizio precedente. Infatti basterebbe notare che

$$c_1 \; or \; c_2 \; or \; .. \; or \; c_n \; = \; not(not \; c_1 \; and \; not \; c_2 \; and \; .. \; and \; not \; c_n)$$

Quindi, per verificare che almeno un carattere sia una vocale sarebbe bastato verificare la falsità del seguente fatto: "tutti i caratteri sono consonanti".

4.6 Iteratori (for)

Un altro meccanismo di iterazione molto usato in Python è costruito mediante l'iteratore `for` in un modo molto simile a quello utilizzato per i descrittori di lista. L'iteratore `for` attraversa uno ad uno tutti gli elementi di un iterabile. La sintassi è la seguente:

```
for X in IT:
    BLOCCO
```

Ad ogni iterazione, il riferimento `X` riferisce l'elemento corrente in `IT` e può essere direttamente utilizzato nelle computazioni eseguite nelle istruzioni del blocco.

4.6.1 Iterare su un range di interi

Nel caso più semplice, l'iteratore è usato per ripetere un blocco di istruzioni un determinato numero di volte. L'iterabile utilizzato in questo caso viene prodotto dalla funzione `range()`.

```
for v in range(4):
   print("Ciao!")
```

Esso produrrà il seguente output:

```
Ciao!
Ciao!
Ciao!
Ciao!
```

La medesima idea può essere applicata per range più complessi. Per esempio, per calcolare la somma dei numeri dispari da -3 a 10 escluso, possiamo usare un iteratore come segue.

```
n_ini,n_fin,n_step = -3,10,2
sum = 0
for v in range(n_ini,n_fin,n_step):
   sum += v
print "La somma dei numeri compresi tra %d e %d\
 con passo %d e' %d"%(n_ini,n_fin,n_step,sum)
```

4.6.2 Iterare per riferimenti

Il principale motivo per cui si usano gli iteratori è per ripetere una serie di azioni su tutti gli elementi di un iterabile. Qui sotto vediamo un esempio di un iteratore sulle stringhe usato per separare i caratteri della stringa. Il codice

```
print "caratteri:"
for c in "Giorni":
   print "%c"%c,
```

produce il seguente output:

```
caratteri: G i o r n i
```

Possiamo anche iterare in ordine inverso usando la funzione `reversed()` come nel seguente esempio. Il codice

```
print("caratteri:")
for c in reversed("Giorni"):
   print("%c"%c, end=" ")
```

produce il seguente output:

```
caratteri: i n r o i G
```

La stessa cosa può essere generalizzata ad ogni tipo di iterabile: lista, tupla, insieme, dizionario. Per i dizionari, la variabile dell'iteratore assumerà uno ad uno i valori delle chiavi del dizionario. Tale chiave può essere utilizzata all'interno del blocco per ottenere il valore associato a quella chiave. Per esempio,

```
rubrica = {'mara':'340-1123451','giulio':'329-7678854',
    'alfredo':'311-8764568'}
for nome in rubrica: # equivalente a for nome in rubrica.keys()
   print("%s -> tel. %s"%(nome,rubrica[nome]))
```

produce il seguente output:

```
alfredo -> tel. 311-8764568
mara -> tel. 340-1123451
giulio -> tel. 329-7678854
```

Alternativamente, possiamo anche utilizzare l'iteratore per ciclare sulle coppie di un dizionario. Il codice seguente è equivalente a quello dato precedentemente.

```
rubrica = {'mara':'340-1123451','giulio':'329-7678854',
    'alfredo':'311-8764568'}
for (nome,numero) in rubrica.items():
   print("%s -> tel. %s"%(nome,numero))
```

4.6.3 Iterare per indice e riferimenti

Un'altra classe di utilizzo degli iteratori è quella che ci permette di iterare sequenze utilizzando gli indici. In questo caso viene utilizzata la funzione `len()` per calcolare la lunghezza della sequenza e per ottenere il range di interi corrispondente ai suoi indici. Per esempio

```
sequenza = ['a','b','c']
# oppure  = ('a','b','c')
# oppure  = "abc"
for i in range(len(sequenza)):
   print("carattere: %s"%sequenza[i])
```

produce il seguente output:

```
carattere: a
carattere: b
carattere: c
```

Un metodo alternativo per ottenere indice e riferimento degli elementi in una sequenza è utilizzare la funzione `enumerate()`. Tale funzione restituisce coppie composte da un indice numerico, a partire da zero, unico per ogni iterazione, e il normale riferimento all'elemento della collezione. Per esempio,

```
sequenza = ['a','b','c']
# oppure  = ('a','b','c')
# oppure  = "abc"
for i,c in enumerate(sequenza):
   print("%d:%s"%(i,c))
```

produce il seguente output:

```
0:a
1:b
2:c
```

4.7 while o for?

L'iteratore **for** andrebbe usato quando vogliamo ripetere la stessa azione su tutti gli elementi di un iterabile. Gli elementi contenuti nell'iterabile non dovrebbero cambiare durante il ciclo. Viceversa, il ciclo **while** dovrebbe essere utilizzato quando vogliamo ripetere una certa azione fino a che si verifichi una certa condizione. Tipicamente, all'interno del blocco del while troveremo istruzioni che modificano le variabili utilizzate nella condizione di permanenza.

Seguono una serie di esercizi risolti dove viene evidenziata la scelta tra il ciclo while e l'iteratore for.

Esercizio risolto. *Richiedi e controlla un numero intero strettamente positivo e trova i divisori non banali mostrandoli all'utente.*

Script 4.7: divisori_n.py

```
n = int(raw_input("inserire il numero: "))
while n<=0 :
    print "il numero inserito non e' positivo."
    n =  int(raw_input("inserire il numero: "))

for i in range(2,n+1): #bastava anche 2<=i<=n/2
   if n%i == 0 : #numero n divisibile per i (i e' divisore)
      print i
```

Esercizio risolto. *Scrivere un programma che dato un numero intero maggiore di* 9 *da tastiera ne calcoli tutte le sue scomposizioni come somma di due numeri.*

Script 4.8: scomposizioni_n.py

```
### legge il numero > 9
n = int(raw_input("inserire il numero: "))
while n<=9 :
    print "il numero inserito non e' maggiore di 9."
    n =  int(raw_input("inserire il numero: "))
### stampa le scomposizioni
for i in range(1,n+1):
   print "%d %d"%(i,n-i)
```

Esercizio risolto. *Scrivere un programma che legga* 10 *numeri interi e li stampi a video via via che li legge.*

Script 4.9: dieci_interi.py

```
### soluzione non ottimale che usa il while
n_letti = 0
while n_letti<10:
   num = int(raw_input("Inserire un numero intero: "))
   print "Valore: %d" % num
   n_letti += 1
```

Script 4.10: dieci_interi.py

```
### soluzione migliore che usa il for
for n_letti in range(10):
   num = int(raw_input("Inserire un numero intero: "))
   print "Valore: %d" % num
```

Esercizio risolto. *Scrivere un programma che esegua la somma dei primi* n *numeri interi. Il numero* n *viene letto da input.*

Script 4.11: somma_n.py

```
### soluzione non ottimale
n = int(raw_input("inserire n: "))
i,sum = 1,0
while i<=n:
   sum += i
   i += 1
print "La somma dei primi %d numeri e': %d" % (n,sum)
```

Script 4.12: somma_n.py

```
### soluzione migliore
n = int(raw_input("inserire n: "))
sum = 0
for i in range(1,n+1)
   sum += i
print "La somma dei primi %d numeri e': %d" % (n,sum)
```

Esercizi del capitolo

Esercizio 4.1

Scrivere uno script che prenda in input un intero N da tastiera e stampi a schermo N linee di testo. La linea i-esima ($i = 1, \ldots, N$) conterrà i numeri da N a $N - i + 1$:

Esempio per $N = 4$:
4
4 3
4 3 2
4 3 2 1

Suggerimenti:

1. *per stampare senza andare a capo usare la virgola dopo il comando* `print`. *Esempio:* `print 'ciao',`
2. *per stampare una riga vuota usare* `print` *senza argomenti.*

Esercizio 4.2 (*)

Scrivere un programma Python che:

1. *legga una frase introdotta da tastiera. La frase si considera terminata all' introduzione del carattere di invio. La frase contiene sia caratteri maiuscoli che caratteri minuscoli;*
2. *stampi a schermo la frase letta;*
3. *stampi a video per ognuna delle lettere dell'alfabeto, il numero di volte che la lettera compare nella stringa.*

Soluzione:

Script 4.13: lettere.py

```
import string
str = raw_input("Frase: ")
print str
for c in string.letters:
   print "%s compare %d volte"%(c,str.count(c))
```

Esercizio 4.3

Dare uno script che richieda in input una serie di numeri interi fino a quando il numero inserito non è uguale a zero. Dopo l'inserimento del numero 0, il programma dovrà stampare a video il valore della somma, della media, del minimo e del massimo dei valori inseriti (0 escluso). Evitare l'uso di risorse (per esempio la memoria) non strettamente necessarie.

```
### Esempio di interazione:
inserire numero: 1
inserire numero: -4
inserire numero: 2
inserire numero: 5
inserire numero: 0
Somma: 4
Media: 1.0
Minimo: -4
Massimo: 5
```

Esercizio 4.4 (*)

Creare uno script verifyRand.py che simula N volte il lancio di un dado e stampa le frequenze di uscita di ciascuna faccia del dado. Verificare all'aumentare del numero di lanci la convergenza alla probabilità teorica di un sesto.

Suggerimento: usare `randrange` *dal modulo random.*

Soluzione:

Script 4.14: verifyRand.py

```
1 from random import randrange
2 N = 10000 # un numero sufficientemente grande
3 lanci = [0]*6 # nro uscite per ogni faccia
4 for i in range(N):
5    lanci[randrange(6)] += 1
6 print lanci
```

Esercizio 4.5

Creare uno script che, data una lista di interi L, stampi `True` *se la lista è ordinata, ovvero* `L[i]<L[j]` *se* `i<j` e stampi `False` altrimenti.

5

Funzioni e ricorsione

In un progetto medio-grande l'intero processo di programmazione può essere eseguito seguendo due approcci fondamentali e opposti: un approccio *top-down* o un approccio *bottom-up*. Nella modalità top-down il problema da risolvere viene prima ridotto in problemi più semplici. In ogni fase assumiamo l'esistenza di una soluzione corretta per i problemi ridotti e ci concentriamo su come combinare queste soluzioni. Viceversa, in un approccio *bottom-up*, partiamo dalla implementazione delle funzionalità più semplici che verosimilmente potranno servirci in futuro. Le funzionalità più complesse vengono poi definite sulla base di queste.

Entrambi questi approcci hanno pro e contro se presi individualmente. Utilizzare solo un approccio top-down spesso porta ad avere programmi che sono soluzioni ad-hoc per il problema in esame ma difficilmente estendibili o riutilizzabili su problemi diversi. Viceversa, l'approccio bottom-up richiede di implementare prima le funzionalità più semplici che risulteranno utili per la soluzione di una moltitudine di problemi anche molto diversi tra loro e che potranno quindi essere condivise. La soluzione basata su un approccio top-down risulta comunque più focalizzata sul problema specifico, generalmente più veloce da realizzare e spesso più efficiente di un approccio bottom-up puro.

Come spesso accade l'approccio migliore sta in una via di mezzo tra le due suddette strategie pure. Infatti, il miglior processo di programmazione, quando abbiamo a che fare con progetti di dimensioni medio-grandi, è quello che alterna fasi top-down a fasi bottom-up.

Python implementa egregiamente entrambe le strategie. Da una parte ci offre un largo spettro di funzionalità, tipi di dati e moduli esterni, già predefiniti nel linguaggio, che si possono direttamente utilizzare oppure personalizzare estendendoli, per esempio mediante definizione di nuovi tipi o classi (approccio bottom-up). D'altra parte, Python ci offre il meccanismo delle funzioni per la progettazione e implementazione modulare di algoritmi (approccio top-down).

Dal punto di vista dell'*utilizzatore*, una funzione può essere pensata come una scatola nera (*black box*) che implementa una certa funzionalità. Non siamo interessati a come questa funzionalità sia effettivamente implementata. Piuttosto, siamo interessati alla cosiddetta **interfaccia** della funzione: il tipo dei dati in ingresso e il tipo dei risultati in uscita. Inoltre, è importante specificare un insieme di vincoli sui parametri in ingresso (**pre-condizioni**) e delle proprietà sul

risultato prodotto in uscita al variare dei valori in ingresso che verificano le pre-condizioni (**post-condizioni**). Una funzione progettata in modo tale che le post-condizioni siano sempre verificate per ingressi che verificano le pre-condizioni si dice **corretta**. Dal punto di vista del *programmatore*, invece, oltre a dare una specifica il più possibile precisa, dovremo anche occuparci di come realizzare tale funzionalità in modo da rispettare le specifiche date, nel modo più efficiente e elegante possibile.

5.1 Funzioni

Il termine *funzione* rimanda al concetto matematico di funzione, ovvero un'associazione tra valori in ingresso e un valore calcolato. In modo simile alle funzioni matematiche, una funzione in Python è un pezzo di programma a cui si possono "passare" degli oggetti (argomenti della funzione) e "restituisce" un oggetto risultato. A differenza di una funzione matematica, però, in una funzione Python è anche possibile modificare lo stato del programma. In questo senso, una funzione in Python è più propriamente un sotto-programma.

Utilizzare le funzioni ha molti vantaggi, i quali includono:

Modularità. La programmazione modulare consiste nella costruzione di programmi ben suddivisi in parti (moduli o funzioni) il più possibile indipendenti, quindi sviluppabili separatamente (implementazione e testing separato) con relazioni di interazione con le altre parti ben definite (tramite la definizione di una *interfaccia*).

Modificabilità. Funzioni o moduli indipendenti con una chiara interfaccia permettono una facile modifica del codice che le implementa. Se l'interfaccia verso l'esterno rimane invariata, queste modifiche risultano assolutamente indolori per il resto del programma.

Riusabilità. Le funzioni permettono di raggruppare e generalizzare il codice per poterlo utilizzare più volte durante l'esecuzione di un programma. Inoltre, esse permettono di *fattorizzare* i programmi evitando inutili ridondanze.

Leggibilità. L'utilizzo delle funzioni ci permette di assegnare un nome ad una funzionalità ben precisa del programma che di solito corrisponde a più righe di codice. Il nome della funzione, se scelto correttamente, dovrebbe evocare meglio lo scopo del sotto-programma racchiuso nella funzione. Questo porta inevitabilmente ad una maggior leggibilità del programma (ad una prima occhiata al codice dovrebbe essere già chiaro cosa fa).

Compattezza. L'utilizzo delle funzioni permette di creare codice molto più compatto sia verticalmente, poiché una funzione "nasconde" più righe di codice, sia "orizzontalmente", in quanto l'utilizzo delle funzioni riduce il numero di indentazioni.

5.1.1 Chiamata di funzioni

Abbiamo già visto esempi di funzioni predefinite Python e come utilizzarle. La funzione `type()`, per esempio, prende come argomento un oggetto e ritorna un altro oggetto che rappresenta il tipo dell'oggetto passato per argomento.

```
>>> type('sono un tipo')
 str
```

La funzione `id()` è un altro esempio di funzione che, passato un oggetto come parametro, ritorna un altro oggetto che ne rappresenta l'identità (di tipo `long`). Possiamo per esempio scrivere:

```
>>> type(id('sono un tipo'))
 long
```

Qui abbiamo utilizzato una *funzione composta* ovvero abbiamo applicato la funzione `type()` all'oggetto risultato della funzione `id()` ottenendo quindi (un oggetto che rappresenta) il tipo dell'oggetto ritornato dalla funzione `id()`. Come è naturale nel caso di funzioni composte, l'invocazione delle funzioni coinvolte avviene sempre dall'interno verso l'esterno.

5.1.2 Definizione di funzioni

La sintassi generale per definire funzioni è la seguente:

```
def nome_funzione(<parametri_formali>):
   '''stringa di documentazione della funzione'''
   <corpo_della_funzione>
```

I parametri formali, come la stringa di documentazione, sono opzionali. Tali parametri rappresentano il nome (o riferimento) col quale vogliamo identificare gli oggetti passati come argomenti nella chiamata alla funzione. Se ce ne sono più di uno sono separati da virgole. Facciamo il primo esempio di definizione di una funzione che non ha argomenti e stampa un semplice saluto.

```
def saluti():
   print "Salve!\n"
```

Vediamo adesso come sia possibile definire funzioni con argomenti. Immaginiamo di voler definire una funzione che stampi saluti ad una persona in particolare. Definiamo quindi `saluti_a()` che prende come unico argomento la stringa indicante il nome della persona da salutare. Una implementazione di tale funzione potrebbe essere

```
def saluti_a(un_nome):
   print "Salve %s!\n"%un_nome
```

Per invocare (chiamare) una funzione definita dall'utente valgono le stesse regole delle funzioni predefinite. Indichiamo il nome della funzione e, tra parentesi, gli eventuali argomenti separati da virgole:

```
nome_funzione(<parametri_attuali>)
```

In Python è anche possibile indicare dei valori di default per i parametri della funzione, ovvero il valore da dare al parametro all'interno della funzione quando la funzione viene richiamata senza indicarne uno per quel particolare parametro.

```
def saluti_a(un_nome = 'senza nome'):
   print "Salve %s!\n"%un_nome
```

Con questa nuova definizione per la funzione `saluti_a()`, chiamare la funzione senza argomenti equivale a invocare la funzione `saluti_a('senza nome')`. Facciamo qualche esempio utilizzando le due semplici funzioni appena definite.

```
>>> saluti()
 Salve!
>>> saluti_a('Gino')
 Salve Gino!
>>> saluti_a('Luigi')
 Salve Luigi!
>>> saluti_a()
 Salve senza nome!
```

5.1.3 Ritornare oggetti

Le funzioni definite fino a questo punto nel capitolo non ritornano valori o, per meglio dire, ritornano un oggetto molto particolare: l'oggetto nullo (`None`). Esaminiamo le seguenti righe di codice:

```
>>> type(saluti())
Salve
<type NoneType>
```

Come detto in precedenza, la valutazione della composizione di due funzioni `type(saluti())` procede dall'interno verso l'esterno. Viene quindi eseguita prima la funzione `saluti()`, che stampa a video il saluto. Successivamente, l'oggetto ritornato (`None`) viene utilizzato come argomento della funzione più esterna (la funzione `type()`) che ne visualizza la descrizione del tipo.

Se vogliamo definire una funzione che ritorni un oggetto diverso da `None` all'ambiente chiamante dobbiamo esplicitamente usare il comando

```
return EXP
```

Quando, eseguendo una funzione, l'interprete incontra tale comando, la funzione ritorna l'oggetto risultante dalla valutazione dell'espressione `EXP` all'ambiente chiamante e l'esecuzione della funzione termina immediatamente. Come visto precedentemente, quando una funzione termina senza aver incontrato il comando `return`, l'oggetto restituito dalla funzione sarà l'oggetto `None`.

Facciamo un semplice esempio dove andiamo a definire una funzione che calcola il valore assoluto di un numero.

```
def valore_assoluto(x):
   if x<0:
      return -x
   return x
```

Seguendo il flusso di esecuzione della funzione vediamo che il valore del parametro `x` è confrontato con il valore zero. Nel caso tale valore sia minore di zero, il flusso entra nel ramo `if`, valuta `-x` e ottiene l'oggetto da restituire all'ambiente chiamante. A questo punto la funzione termina e il controllo ritorna all'ambiente chiamante. La parte rimanente della funzione (`return x`) non verrà eseguita. Quest'ultima istruzione viene eseguita solo nel caso in cui l'espressione valutata nella `if` risulti falsa. Notare che, grazie alla terminazione provocata dal comando `return`, possiamo risparmiarci un ramo `else`.

Esercizio risolto. *Scrivere una funzione* `chiedi_positivo()` *che richieda all'utente un numero intero positivo e stampi un messaggio di errore nel caso il numero inserito non sia corretto. Indicare pre e post condizioni.*

Script 5.1: chiedi_positivo.py

```
1  # Pre: str(msg), prompt da visualizzare
2  # Post: ritorna il numero (int) letto
3  def chiedi_numero(msg):
4     return int( raw_input(msg) )
5  # Pre: -
6  # Post: ritorna un numero positivo letto da input
7  def chiedi_positivo():
8     n = chiedi_numero("Inserire un numero intero positivo:")
9     while n<0:
10      n = chiedi_numero("Inserire un numero intero positivo:")
11    return n
```

In questo esercizio, abbiamo definito una funzione che a sua volta chiama un'altra funzione definita dall'utente enfatizzando così un buon esempio di codice fattorizzato correttamente allo scopo di massimizzare la riusabilità dello stesso.

Esercizio risolto. *Scrivere una funzione che verifichi se un numero intero positivo è primo.*

Script 5.2: primo.py

```
1  def is_primo(n):
2     for i in range(2,n/2):
3        if not n%i:
4           return False
5     return True
```

Abbiamo già visto in precedenza come verificare se un numero è un numero primo. In questo esercizio includiamo la procedura di verifica in una funzione allo scopo di permettere un semplice riutilizzo del codice.

Esercizio risolto. *Scrivere una funzione che calcoli il fattoriale di un numero intero positivo letto da tastiera.*

Script 5.3: fattoriale_n.py

```
from chiedi_positivo import *

def fatt_n(n):
    fat = 1
    for i in range(2,n+1):
        fat *= i
        i += 1
    return fat

def fattoriale_n() :
    n = chiedi_positivo()
    return fatt_n(n)
```

La funzione `fatt_n()` computa il fattoriale del parametro n come moltiplicazione dei valori 2, 3, ecc. fino al valore n. Si noti che la variabile `i` contiene l'elemento da moltiplicare nell'iterazione corrente e `fat` contiene, alla fine di ogni passo, la moltiplicazione $2 * 3 * \ldots * i$. Si noti inoltre che per $n = 1$ la funzione ritorna giustamente il valore 1. Quando $n > 1$ la funzione ritorna $2 * 3 * \ldots * n$ che è proprio il fattoriale del numero n.

Esercizio risolto. *Scrivere uno script che riceva da input due interi positivi n e k e calcoli il coefficiente binomiale*

$$C(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Script 5.4: binomiale.py

```
from chiedi_positivo import *
from fattoriale_n import *

n = chiedi_positivo()
k = chiedi_positivo()

# richiede i valori fino a quando n>=k
while n<k :
    n = chiedi_positivo()
    k = chiedi_positivo()

c = fatt_n(n)/(fatt_n(k)*fatt_n(n-k))
print "C(%d,%d) = %d" % (n,k,c)
```

La soluzione proposta utilizza pesantemente chiamate ad altre funzioni definite dall'utente per la fattorizzazione del codice. È facile constatare che, se non avessimo utilizzato le numerose chiamate a funzioni, il codice prodotto sarebbe stato molto più lungo e meno leggibile. Ciò nonostante, lo script non è del tutto ottimizzato dal punto di vista dell'efficienza poiché alcuni calcoli, in particolare quelli effettuati nelle tre chiamate a `fatt_n()` in linea 12, vengono così ripetuti più volte, inutilmente. Questo è un esempio di fattorizzazione del codice migliorabile.

5.1.4 Visibilità delle variabili

Le variabili definite in un programma sono visibili, nel senso che si può riferirirsi ad esse, nel cosiddetto *scope* della variabile. Le variabili in Python possono essere o *globali* o *locali*. Le variabili definite dentro una funzione sono locali (alla funzione). Quelle definite fuori dalle funzioni, in un modulo, sono dette globali.

Le variabili globali nascono nel momento in cui vengono definite e la loro vita dura per tutta l'esecuzione del modulo dove esse sono state definite. Le variabili locali, invece, nascono nel momento in cui vengono definite in una funzione e durano fino al termine dell'esecuzione della funzione stessa.

Nell'esempio che segue definiamo una variabile globale `una_globale` e una variabile locale `una_locale`, quest'ultima all'interno della funzione `funz()`.

```
1 una_globale = "abc"
2 def funz():
3    una_locale = "def"
4    print una_globale+una_locale
5 funz() # stampa abcdef
6 print una_globale # stampa: abc
7 una_globale += "gh"
8 funz() # stampa abcghdef
9 # qui la variabile una_locale e' inaccessibile!
```

Notare come all'interno della definizione di funzione `funz()` si utilizzi volta per volta il valore che assume la variabile globale nel momento dell'invocazione. Le due chiamate a `funz()` in riga 5 e 8 stamperanno valori diversi proprio perchè la variabile globale assume due valori diversi nei due momenti.

5.1.5 Funzioni come argomenti

Il comando `def` in Python non fa altro che creare un nuovo riferimento ad un oggetto di tipo `function`. Per verificarlo possiamo provare a valutare prima il nome della funzione omettendo parentesi e argomenti e poi il suo tipo.

```
>>> saluta #solo il nome della funzione
 <function saluta at 0x00000000002EA7C88>
>>> type(saluta)
 <type 'function'>
```

Ciò indica chiaramente che una funzione in Python è un oggetto come tutti gli altri, ovvero possiamo popolare una lista con delle funzioni, o passare una funzione come argomento ad un'altra funzione, e così via.

Esercizio risolto. *Data una funzione $f : \mathbb{R} \to Y$ (argomenti di tipo float e valore ritornato di tipo Y), un intervallo $[a, b]$ e un valore intero positivo n. Dare una funzione che ritorna una lista con le coppie $(x, f(x))$ per $n + 1$ valori con $a \leq x \leq b$, su intervalli di eguale ampiezza. Si indichino inoltre le pre-condizioni e le post-condizioni della funzione.*

Script 5.5: tabella_f.py

```
# Pre : function(f) : float -> Y
#       float(a,b) estremi intervallo, a<b
#       int(n) numero di intervalli n>=1
# Pos : ritorna una lista con n+1 coppie (x,f(x))
def tabella_f(f,a,b,n) :
    dx = float(b-a)/n
    xy = []
    for i in range(n+1):
        x = a+dx*i
        xy.append((x,f(x)))
    return xy
```

5.1.6 Funzioni annidate

Il fatto che le funzioni siano oggetti come tutti gli altri in Python ha un'altra conseguenza: è assolutamente lecito definire funzioni all'interno di definizioni di altre funzioni e per queste valgono le stesse regole di visibilità dei comuni riferimenti. Consideriamo la funzione di esempio `stampa_molti()` che stampa uno ad uno gli elementi di una lista passata per argomento:

```
def stampa_molti(a):
   def stampa_uno(s):
      print("chiamata stampa")
      print("-> %s"%s)
   print("chiamata esterna")
   for s in a:
     stampa_uno(s)
```

Il codice seguente dimostra che la funzione `stampa_uno()` è visibile solo all'interno di (ovvero è locale a) `stampa_molti()`. Infatti,

```
stampa_molti([1,"ciao",5.67])
stampa_uno("wow")
```

produrrà il seguente output:

```
chiamata esterna
chiamata stampa
-> 1
chiamata stampa
-> ciao
chiamata stampa
-> 5.67

Traceback (most recent call last):
  File "C:\UNI\CORSI\PYBOOK\ESCULAPIO\CODICE\annidate.py", line 10,
      in <module>
    stampa_uno("wow")
NameError: name 'stampa_uno' is not defined
```

Segue un altro esempio di applicazione delle funzioni annidate.

Esercizio risolto. *In matematica la congettura di Goldbach è uno dei più antichi problemi irrisolti nella teoria dei numeri. Essa afferma che ogni numero pari maggiore di 2 può essere scritto come somma di due numeri primi (eventualmente uguali). Dare una funzione* `do_goldbach(N)` *che, preso in input un numero intero* `N`*, verifichi per tutti gli* n *pari, tali che* $4 \leq n \leq N$*, se esiste una coppia di numeri primi* (a, b) *con* $n = a + b$*, e la stampi.*

Script 5.6: goldback.py

```
1  from primo import *
2
3  def do_goldbach(N):
4
5     def goldbach(n):
6        for i in range(2,n/2+1):
7          if is_primo(i) and is_primo(n-i):
8              return (i,n-i)
9        return None
10
11    for n in range(4,N+1,2):
12       g = goldbach(n)
13       if g:
14          print("n: %d=%d+%d"%(n,g[0],g[1]))
15       else:
16          print("n: %d (non esiste!)"%n)
17
18 ## do_goldbach(100)
```

La soluzione proposta si basa su una funzione `goldback()` che prende in input un intero n pari e verifica l'esistenza di due numeri primi per cui vale la proprietà di Goldbach. Tale funzione restituisce la coppia di numeri se esistono oppure `None` se non esistono. La funzione `do_goldback()` serve solo ad iterare

la procedura per i diversi numeri pari e a visualizzare la soluzione. Qui, l'annidamento ha il senso di rendere la funzione `goldback()` locale alla (e quindi utilizzabile solo dalla) funzione `do_goldbach()`.

5.2 Funzioni ricorsive

Per alcuni problemi è naturale dare una soluzione avendo a disposizione la soluzione dello stesso problema per un caso più "semplice". Detta così sembra un pò una contraddizione in quanto stiamo dicendo che per risolvere un problema abbiamo bisogno della sua soluzione. Ma prendiamo subito un esempio che chiarificherà il concetto.

Sappiamo che la funzione fattoriale di un numero intero n è definita come la moltiplicazione dei numeri interi da 2 fino a n. Il fattoriale di 1 è uguale a 1 per definizione. Più formalmente possiamo scrivere $F(n) = n * (n-1) * \cdots * 2$ e $F(1) = 1$. Ammettiamo per un attimo di conoscere il valore di $F(n-1)$, possiamo verificare che $F(n) = n * F(n-1)$ e quindi è facile calcolare il valore del fattoriale anche per n. Ripetendo il ragionamento, ammettendo di conoscere il fattoriale per $n-2$ è possibile calcolare il fattoriale di $n-1$ che ci serviva per calcolare il fattoriale di n. Di questo passo arriviamo a definire il fattoriale di un qualsiasi numero n in funzione solo del fattoriale di 1 ovvero ci siamo ridotti al *caso base*.

Ecco svelato il trucco. Ovvero, se riusciamo a definire la soluzione di un problema ($F(n)$ nel nostro caso) in termini della soluzione al problema nella versione più semplice, $F(n) = n * F(n-1)$ e contemporaneamente riusciamo a dare un caso base, ovvero la soluzione al caso *più semplice possibile*, allora abbiamo una soluzione *ricorsiva* per il problema!

Vediamo ora come è possibile implementare le semplici regole del calcolo ricorsivo del fattoriale di n utilizzando una funzione Python.

Script 5.7: fat_ric.py

```
1 def fatt_ricorsivo(n):
2     if n <= 1 :
3         return 1
4     return n*fatt_ricorsivo(n-1)
```

e proviamo a scrivere l'ordine delle chiamate nel semplice caso in cui $n = 3$.

```
fatt_ricorsivo(3)
   #chiamata alla funzione fatt_ricorsivo(2)
   3 * fatt_ricorsivo(2)
      #chiamata alla funzione fatt_ricorsivo(1)
      2 * fatt_ricorsivo(1)
         #caso base, la funzione restituisce 1
         if 1 <= 1 : return 1
      #adesso fatt_ricorsivo(2) puo' calcolare il valore di ritorno
      ritorna 2 * 1
   #adesso fatt_ricorsivo(3) puo' calcolare il valore di ritorno
   ritorna 3 * 2
---> 6
```

Esercizio risolto. *Scrivere uno script che tramite l'utilizzo di una funzione ricorsiva calcoli il MCD tra due numeri con il metodo delle sottrazioni successive. Lo schema ricorsivo è il seguente:*

- se $x = y$, allora $MCD(x, y) = x$ (caso base)
- se $x > y$, allora $MCD(x, y) = MCD(x - y, y)$
- se $x < y$, allora $MCD(x, y) = MCD(x, y - x)$

Script 5.8: mcds.py

```
1 def mcdr(x,y) :
2     if x==y : return x
3     elif x>y : return mcdr(x-y,x)
4     else return mcdr(x,y-x)
```

5.2.1 Ricorsione in avanti e all'indietro

Analizziamo il comportamento delle due funzioni ricorsive seguenti:

```
def avanti(s) :
    if not s: return
    print s[0],
    avanti(s[1:])

def indietro(s) :
    if not s: return
    indietro(s[1:])
    print s[0],
```

Le due funzioni ricorsive `avanti()` e `indietro()` differiscono solo per la posizione della chiamata ricorsiva rispetto alle altre istruzioni della funzione. Entrambe prendono una sequenza come parametro. Il caso base è lo stesso nei due casi, ovvero si verifica se la sequenza è vuota.

Nel caso della prima funzione (`avanti()`) si dice che la ricorsione è "in avanti" (anche detta "in coda"). In questo caso, la chiamata ricorsiva è l'ultima istruzione eseguita prima di ritornare dalla funzione. Il caso opposto si ha quando la chiamata ricorsiva risulta la prima istruzione, escludendo quelle istruzioni che gestiscono i casi base chiaramente. In quest'ultimo caso, come nella funzione `indietro()` parliamo di ricorsione "all'indietro".

Analizzando l'annidarsi delle chiamate ricorsive e delle stampe nei due esempi riportati sopra capiamo meglio il perchè della terminologia definita precedentemente. Nel primo caso le stampe vengono effettuate andando giù (o in avanti) nella ricorsione, mentre nel secondo caso le stampe vengono effettuate risalendo indietro dalle chiamate ricorsive.

```
avanti("abc")
   print "a",
   #chiamata alla funzione avanti("bc")
   avanti("bc")
      print "b",
      #chiamata alla funzione avanti("c")
      avanti("c")
         print 'c',
         #chiamata alla funzione avanti("")
         avanti("")
             #caso base, la funzione ritorna
         #la funzione avanti("c") ritorna
      #la funzione avanti("bc") ritorna
   #la funzione avanti("abc") ritorna
### STAMPA "abc"
```

```
indietro("abc")
   #chiamata alla funzione indietro("bc")
   indietro("bc")
      #chiamata alla funzione indietro("c")
      indietro("c")
         #chiamata alla funzione indietro("")
         indietro("")
             #caso base, la funzione ritorna
         print 'c',
         #la funzione indietro("c") ritorna
      print "b",
      #la funzione indietro("bc") ritorna
   print "a",
   #la funzione indietro("abc") ritorna
### STAMPA "cba"
```

Esercizio risolto. *Utilizzando insieme la ricorsione in avanti e indietro definire una funzione ricorsiva che, data una stringa, stampi la stringa a caratteri maiuscoli seguita dalla stessa stringa inversa a caratteri minuscoli.* Per esempio, se chiamata con argomento la stringa `'cIaO'`, la funzione dovrà stampare `CIAOoaic`.

```
def up_low(s) :
   if not s: return
   print s[0].upper(),
   up_low(s[1:])
   print s[0].lower(),
```

5.2.2 Ricorsione o non ricorsione?

A volte potremmo trovarci davanti a questo grande dilemma: dato un problema da risolvere, è meglio dare una soluzione ricorsiva o una equivalente iterativa?

La risposta a questa domanda chiaramente dipende dal problema che dobbiamo risolvere. Generalmente parlando, una soluzione ricorsiva (quando disponibile) è più elegante di una equivalente iterativa. Una soluzione ricorsiva però potrebbe essere meno efficiente. Come regola generale potremmo darci la seguente: a parità (o quasi) in termini di efficienza è più corretto usare la ricorsione; altrimenti meglio la versione iterativa.

Esaminiamo il prossimo esercizio per chiarire ulteriormente questo punto.

Esercizio risolto. *Scrivere una funzione ricorsiva per il calcolo del numero n-esimo di Fibonacci. Si consideri lo schema ricorsivo seguente:*

$$F(0) = 0,\ F(1) = 1,\ F(n) = F(n-1) + F(n-2)$$

Script 5.9: fib_ric.py

```
1 def fib_ric(n) :
2     if n == 0 :
3         return 0
4     if n == 1 :
5         return 1
6     return fib_ric(n-1) + fib_ric(n-2)
```

La soluzione dell'esercizio proposta in questo script è decisamente inefficiente. Osservandola con attenzione ci accorgiamo che molti calcoli vengono ripetuti inutilmente. Per esempio, immaginiamo di dover calcolare `fib_ric(3)`. Nell'ambiente della funzione vengono chiamate `fib_ric(2)` e `fib_ric(1)`. La chiamata a `fib_ric(2)` produce a sua volta altre due chiamate: `fib_ric(1)` e `fib_ric(0)`.

```
fib_ric(3)
   fib_ric(2)
      fib_ric(1)
         return 1
      fib_ric(0)
         return 0
      return 1 <-- 1 + 0
   fib_ric(1)
      return 1
   return 2 <-- 1 + 1
```

Anche solo da questo semplice esempio si evince che in totale vengono effettuate due chiamate a `fib_ric(1)` ripetendo quindi due volte lo stesso calcolo. È facile comprendere come, all'aumentare del valore del parametro, il numero di chiamate cresce seguendo una legge esponenziale!
Per comprendere empiricamente il comportamento di questa funzione, riportiamo il numero di chiamate effettuate al variare del parametro n.

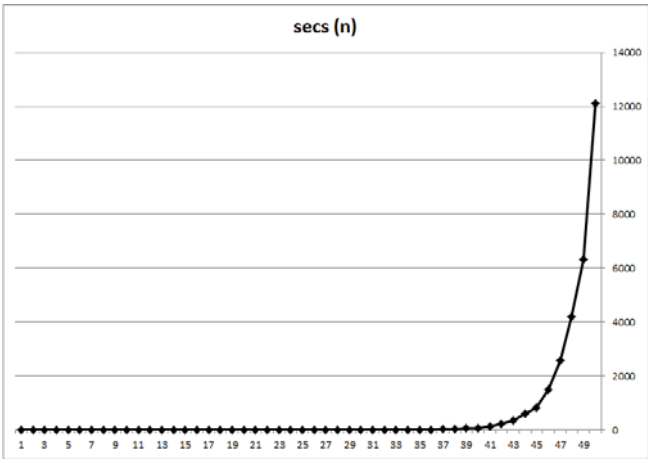


Figura 5.1: Tempi (in secondi) per il calcolo dell' n-esimo numero di Fibonacci.

```
fib_ric(1)  # 1 chiamata
fib_ric(2)  # 2 chiamate
fib_ric(10) # 89 chiamate
fib_ric(20) # 10946 chiamate
fib_ric(30) # 1346269 chiamate
fib_ric(40) # 165580141 chiamate (qualche minuto)
fib_ric(50) # il calcolo richiede piu' di tre ore!
```

In Figura 5.1 si può osservare l'andamento esponenziale del tempo richiesto per il calcolo al variare dello stesso parametro n.

Di seguito una versione alternativa, un po' più complessa e meno elegante, ma molto più efficiente, che si basa sul seguente schema ricorsivo:

$$F(n) = Q(n)[0]$$

dove

$$Q(0) = (0,0),\ Q(1) = (1,0),\ Q(n) = (x+y, x) \text{ e } (x,y) = Q(n-1)$$

L'idea qui sta nel farsi ritornare i numeri di Fibonacci di $n-1$ e $n-2$ in una coppia. Questi valori vengono utilizzati per il calcolo del numero di Fibonacci n-esimo.

Script 5.10: fib_eff.py

```
def fib_eff(n) :
   def fib_ric(n):
      if n == 1 : return (1,0)
      if n == 2 : return (1,1)
      (x,y) = fib_ric(n-1)
      return (x+y,x)
   return fib_ric(n)[0]
```

In questa versione la vera funzione ricorsiva è la funzione `fib_ric()` interna alla funzione `fib_eff()`. Ogni chiamata ricorsiva `fib_ric(n)` produce una tupla contenente due valori ($F(n)$, $F(n-1)$). Invitiamo il lettore ad eseguire questa funzione al variare del parametro n e verificare come sia possibile calcolare il valore dell'n-esimo numero di Fibonacci per n molto grandi in modo molto efficiente. Il numero di chiamate ricorsive cresce linearmente con n.

Ora, potremmo chiederci se è accettabile una soluzione come quella appena presentata rispetto ad una equivalente iterativa di cui diamo qui una possibile versione.

Script 5.11: fib_iterativa.py

```
1 def fib_iterativa(n) :
2     if n in (0,1) :
3         return n
4     f1,f2 = 1,0
5     for i in range(n):
6         f1,f2 = f1+f2,f1
7     return f1
```

La risposta in questo caso è evidentemente no. La soluzione iterativa del calcolo dell'ennesimo numero di Fibonacci appare migliore dal punto di vista della eleganza e della leggibilità. Il vantaggio della nuova formulazione ricorsiva è semmai che può essere facilmente estesa a generalizzazioni dei numeri di Fibonacci (*tribonacci*, *tetranacci*, ecc.), ovvero le successioni che si ottengono sommando gli ultimi k termini della successione con $k > 2$.

5.3 Correttezza di funzioni ricorsive

Una funzione si dice corretta se, per qualsiasi input che verifichi le pre-condizioni, essa produce sempre output che verificano le post-condizioni. Di seguito diamo uno schema generale a cui attenersi per dimostrare formalmente la correttezza di una funzione ricorsiva.

Immaginiamo di voler dimostrare la correttezza di una funzione ricorsiva $f(x)$ la quale richiama ricorsivamente $f(x')$ con $x' < x$ (x' più "semplice" di x). Assumendo le pre-condizioni vere per $f(x)$, ovvero PRE(x), la dimostrazione è di tipo induttivo e si sviluppa su tre fasi:

1. dimostrare che valgono le post-condizioni per il caso base, ovvero

   PRE(x) $\Rightarrow$ POS($f(1)$);

2. dimostrare che valgono le pre-condizioni negli argomenti della chiamata ricorsiva $f(x')$, ovvero

   PRE(x) $\Rightarrow$ PRE(x');

3. assumendo vere le post-condizioni per la chiamata ricorsiva $f(x')$, dimostrare le post-condizioni per la $f(x)$, ovvero

   $\text{POS}(f(x')) \Rightarrow \text{POS}(f(x))$.

Esercizi del capitolo

Esercizio 5.1 (*)

Dare una funzione `eq_gradoI()` *che, dati i coefficienti* a *e* b *di una equazione di primo grado, ritorni la radice dell'equazione. Si diano pre-condizioni in modo da evitare soluzioni degeneri.*

Soluzione:

Script 5.12: eq_gradoI.py

```
# Pre : float(a,b) coefficienti dell'equazione, a!=0
# Pos : float(x) = soluzione dell'equazione ax+b
def eq_gradoI(a,b) :
    return -float(b)/a
```

Esercizio 5.2 (*)

Dare una funzione `eq_gradoI()` *che, dati i coefficienti* a *e* b *di una equazione di primo grado, ritorni la radice dell'equazione o, nel caso non esista, una descrizione del caso degenere ('indeterminato' o 'indefinito').*

Soluzione:

Script 5.13: eq_gradoI_bis.py

```
# Pre  : float(a,b) coefficienti dell'equazione
# Pos  : se a!= 0, ritorna float(x) = soluzione
#          dell'equazione ax+b
#        se a = 0, b!=0, x indefinito
#          (ritorna 'indefinito')
#        se a = 0, b==0, eq. indeterminata
#          (ritorna 'indeterminato')
def eq_gradoI(a,b) :
    if a==0:
        if b==0:
            return 'indeterminato'
        else:
            return 'indefinito'
    return -float(b)/a
```

Esercizio 5.3 (*)

Scrivere una funzione Python `stat_frase()` *che:*

1. *prende in input una stringa. La frase contiene sia caratteri maiuscoli che caratteri minuscoli;*
2. *ritorni una tripla* `(cc,cv,d)` *dove* `cc` *e* `cv` *rappresentano il numero di consonanti e il numero di vocali presenti nella stringa, rispettivamente, e* `d` *è un dizionario dove per ogni carattere dell'alfabeto è associato il numero di occorrenze del carattere nella stringa.*

Soluzione:

Script 5.14: lettere_vc.py

```
def stat_frase(str):
   import string
   def vocale(c):
      if c in "aeiouAEIOU":
         return True
      return False
   ncon,nvoc = 0,0
   d = {}
   for c in string.letters:
      nc = str.count(c)
      d[c] = nc
      if vocale(c):
         nvoc += nc
      else:
         ncon += nc
   return (ncon,nvoc,d)
```

Esercizio 5.4 (*)

Dare schema ricorsivo e funzione ricorsiva `sum_list()` *che ritorna la somma degli elementi di una lista.*

Soluzione:

$$\begin{aligned} S([]) &= 0 \\ S([a|L]) &= a + S(L) \end{aligned}$$

Script 5.15: sum_list.py

```
def sum_list(li) :
   if not li :
      return 0
   return li[0] + sum_list(li[1:])
```

6

Le classi

I tipi di dati che abbiamo incontrato fino a questo momento (numeri, stringhe, liste, ecc.) ci permettono di rappresentare entità più o meno complesse. Per esempio, dovendo descrivere come è fatto un punto in due dimensioni possiamo ritenere che questo possa essere ben rappresentato da una tupla di due elementi. I tipi di base disponibili in Python sono tipi astratti e quindi generici e flessibili. Questa flessibilità si paga in termini di minor chiarezza e minor manutenibilità del codice prodotto. Per questi motivi, specialmente per progetti medio-grandi, risulta utile avere tipi di dati più specifici e ad hoc per un determinato problema. In questo Python ci aiuta con il supporto delle classi.

6.1 Definizione di classe

Le classi rappresentano il meccanismo principale della programmazione orientata agli oggetti (OOP). In pratica, definire una classe significa definire un nuovo tipo di dato e l'insieme di operazioni che è possibile effettuare su di esso. Tutti gli oggetti di una stessa classe avranno gli stessi attributi e la medesima modalità di interazione con le altre parti di un programma.

6.1.1 Classi e oggetti istanza di classe

Una classe è un astrazione di un'entità del mondo reale. Se pensiamo al concetto di persona, per esempio, ci verrà subito in mente che qualsiasi persona ha sempre un nome, una data di nascita e altri attributi generici. Immaginiamo di voler creare oggetti che rappresentano persone in Python. Creare la classe di oggetti `Persona` è molto semplice e lo si fa con il costrutto `class` come segue.

```
class Persona():
    '''classe persona'''
```

Ecco fatto. Abbiamo appena definito un nuovo tipo di dati: il tipo `Persona`. La stringa tra triplici apici è opzionale e rappresenta la stringa di documentazione della classe. Come per qualsiasi altro tipo di dati in Python, volendo creare un nuovo oggetto di tipo `Persona`, possiamo semplicemente scrivere:

```
alessia = Persona()
```

Per comprendere meglio cosa succede con la definizione di classi e istanze di classi, eseguiamo ora il seguente codice:

```
>>> Persona
<class __main__.Persona at 0x000000000309A588>
>>> alessia
<__main__.Persona instance at 0x00000000030E6908>
```

L'espressione `Persona` (senza parentesi) viene valutata come classe definita nel modulo `__main__`. Utilizzando il nome del riferimento ad un oggetto, invece che il nome della classe, possiamo verificare che `alessia` si riferisce ad una istanza (instance) di classe `Persona`.

6.1.2 Attributi e metodi di classe

Una classe si contraddistingue per *(i)* un insieme di attributi (detti anche membri) associati alla classe e *(ii)* un insieme di metodi della classe, ovvero, particolari funzioni che permettono di operare su tutte le istanze di quella classe.

Facciamo un semplice esempio estendendo la nostra classe basica `Persona`. Oltre ai già nominati attributi: nome (una stringa), cognome (una stringa), data di nascita (una tupla di tre interi), ecc., possiamo anche definire una serie di metodi per operare sui suoi attributi e ottenere informazioni aggiuntive riguardo a quella persona in particolare. Per esempio, potrebbe risultare utile una funzione che calcola l'età della persona in base alla sua data di nascita e alla data corrente. È facile comprendere come sia meglio trattare l'età di una persona come un "attributo calcolato" piuttosto che aggiungere un reale attributo alla classe. Infatti, tale caratteristica dipende anche dalla data corrente che non è di per sé un attributo della persona.

Una volta definita la classe e un oggetto istanza della classe, possiamo aggiungere attributi all'oggetto utilizzando l'operatore punto (`.`) come in

```
>>> alessia.nome = 'Alessia'
>>> alessia.cognome = 'Bentivoglio'
```

oppure, più convenientemente, possiamo decidere di definire un *costruttore* per la classe e passare i valori degli attributi da inizializzare come argomenti a tale costruttore. Questa procedura ci garantirà la definizione dei corretti attributi ad ogni creazione di una istanza di quella classe.

Il costruttore di classe è un metodo molto particolare di una classe che per convenzione ha nome `__init__()` e serve appunto all'atto della costruzione dell'istanza per inizializzare l'oggetto che stiamo creando. Proseguendo con l'esempio della classe persona, possiamo definire il costruttore nel modo seguente:

```
class Persona():
   def __init__(self,nome,cognome,dn):
      self.nome=nome
      self.cognome=cognome
      self.data_nascita=dn
```

La presenza del primo parametro (`self`) contraddistingue un metodo di una classe rispetto ad una classica funzione ed è utilizzato per riferirsi all'istanza specifica su cui opera il metodo. Gli altri parametri dichiarati nel costruttore servono per inizializzare l'oggetto. Nel nostro caso, inizializziamo i membri della classe (`self.nome`, `self.cognome`, `self.data_nascita`) coi valori passati come argomento al costruttore. Dopo aver definito un siffatto costruttore siamo in grado di creare nuove istanze di persone indicando nome, cognome e data di nascita nella chiamata al costruttore come ad esempio

```
>>> paolo = Persona("Paolo","Bianchi",(1981,11,7))
>>> luigi = Persona("Luigi","Rossi",(1945,3,21))
>>> giulio = Persona("Giulio","Verde",(1981,11,8))
```

Una volta che abbiamo le istanze di classe `Persona`, possiamo interrogarle per conoscere il valore dei loro membri, per esempio

```
>>> paolo.nome
 'Paolo'
>>> luigi.cognome
 'Rossi'
```

Passiamo ora a definire altri metodi utili per completare la definizione della classe `Persona`. Seguirà poi una descrizione dei metodi implementati.

Script 6.1: persone.py

```
class Persona():
   def __init__(self,nome,cognome,dn):
      self.nome=nome
      self.cognome=cognome
      self.data_nascita=dn ## data in formato (aaaa,mm,gg)

   def presentati(self):
      print "Ciao! sono %s %s, nato il %02d/%02d/%04d"% \
      (self.nome,self.cognome, self.data_nascita[2], \
      self.data_nascita[1],self.data_nascita[0])

   def piu_vecchio_di(self,p):
      return self.data_nascita < p.data_nascita

   def eta(self):
      from datetime import date
      return date.today().year-self.data_nascita[0]
```

Il metodo `presentati()` stampa una stringa con i dati della persona

```
>>> luigi.presentati()
 Ciao! sono Luigi Rossi, nato il 21/03/1945
>>> paolo.presentati()
 Ciao! sono Paolo Bianchi, nato il 07/11/1981
```

Il metodo `piu_vecchio_di()` prende come argomento un secondo oggetto (`p`) di tipo `Persona` e verifica se la persona su cui si sta applicando il metodo sia o

meno più vecchio di `p`. Notare che, grazie al formato delle date di nascita che abbiamo scelto, ovvero il formato (aaaa,mm,gg), la comparazione delle due date risulta immediata utilizzando la comparazione tra tuple. Un breve test di questo metodo ci conferma che l'implementazione è corretta.

```
>>> luigi.piu_vecchio_di(paolo)
 True
>>> paolo.piu_vecchio_di(paolo)
 False
>>> giulio.piu_vecchio_di(paolo)
 False
```

Il metodo `eta()`, infine, serve a calcolare l'età di una persona. L'età è calcolata come differenza tra l'anno corrente e l'anno di nascita della persona. Otteniamo così

```
>>> date.today().year
2013
>>> luigi.eta()
68
>>> paolo.eta()
32
```

Ci sono due modalità diverse per invocare un determinato metodo, ovvero

- sull'oggetto, come fatto finora, utilizzando la sintassi:
  `oggetto.nome_metodo(argomenti)`
- oppure, sulla classe, utilizzando la sintassi:
  `nome_classe.nome_metodo(oggetto,argomenti)`

La prima modalità è sicuramente la più utilizzata perché più leggibile e intuitiva per il programmatore. In realtà, l'interprete Python internamente usa sempre la seconda modalità di chiamata. Nel caso il programmatore utilizzi la prima modalità, l'interprete Python riuscirà comunque a tradurla correttamente. Notare che, nella seconda modalità, viene indicato precisamente come devono venire passati gli argomenti alla funzione, oggetto compreso. Vedremo tra poco un caso, quando parleremo di gerarchia di classi ed ereditarietà, in cui saremo costretti ad utilizzare la seconda modalità di chiamata.

6.1.3 Programmare con le classi: il colore è servito

Immaginiamo di voler calcolare la probabilità che uno qualsiasi tra gli N giocatori di una partita a poker ottenga un colore servito. Lo vogliamo fare simulando una serie di M partite calcolando con quale frequenza l'evento accade, ovvero quante volte in proporzione al numero totale di partite M otteniamo un colore servito per almeno un giocatore. Chiaramente, più M è grande, migliore sarà l'approssimazione della probabilità calcolata.

Script 6.2: colore_servito.py

```
import random

class Carta:
   '''una carta'''
   def __init__(self,seme,valore):
      self.seme=seme
      self.valore=valore

class Mazzo:
   '''un mazzo di carte'''
   def __init__(self):
      '''inizializza il mazzo'''
      self.carte = []
      for seme in ("quadri","cuori","picche","fiori"):
         for valore in range(1,14):
            self.carte.append(Carta(seme,valore))

   def mescola(self):
      '''mescola il mazzo'''
      ncarte = len(self.carte)
      for i in range(ncarte-1):
         r = random.randint(i,ncarte-1)
         self.carte[i],self.carte[r]=self.carte[r],self.carte
            [i]

   def pesca_carta(self):
      '''pesca una carta'''
      carta = self.carte.pop()
      return carta

   def aggiungi_carta(self,carta):
      self.carte.append(carta)

class Poker:
   '''il gioco del poker'''
   def __init__(self,n_giocatori,n_turni):
      self.n_giocatori = n_giocatori
      self.n_turni = n_turni
      self.mazzo = Mazzo()
      self.mani = None

   def distribuisci_carte(self):
      '''effettua la distribuzione delle carte'''
      self.mani=[set() for g in range(self.n_giocatori)]
      for g in range(self.n_giocatori):
```

```
45          for i in range(5):
46             self.mani[g].add(self.mazzo.pesca_carta())
47
48       def raccogli_carte(self):
49          '''ritira le carte dal tavolo'''
50          for g in range(self.n_giocatori):
51             for carta in self.mani[g]:
52                self.mazzo.aggiungi_carta(carta)
53             self.mani[g]=set()
54
55       def verifica(self):
56          '''almeno un giocatore ha un colore servito?'''
57          def colore(mano):
58             semi = set([carta.seme for carta in mano])
59             return len(semi)==1
60
61          for g in range(self.n_giocatori):
62             if colore(self.mani[g]): return True
63          return False
64
65       def play(self):
66          '''simula turni di gioco'''
67          n_colore = 0
68          for turno in range(1,self.n_turni+1):
69             self.mazzo.mescola()
70             self.distribuisci_carte()
71             if self.verifica():
72                n_colore+=1
73             print "num turni: %d, num colore: %d, prob: %f"%(
                  turno,n_colore,float(n_colore)/turno)
74             self.raccogli_carte()
75
76    # Poker(6,10000).play()
```

L'intero programma (Script 6.2) consta di tre classi: `Carta`, `Mazzo` e `Poker`.

La classe **Carta** rappresenta una carta da gioco con il suo seme, uno nell'insieme {`quadri`,`cuori`,`picche`,`fiori`} e il suo valore, da 1 a 13.

La classe **Mazzo** rappresenta il mazzo di carte e ha un attributo `carte` di tipo lista per contenere la lista ordinata di carte nel mazzo che viene inizializzata nel costruttore. Sono stati previsti inoltre metodi per mescolare le carte del mazzo, per pescare una carta dal mazzo e per aggiungere una carta in cima al mazzo. L'unico metodo che merita ulteriore discussione è il metodo `mescola()` nel quale viene applicato un algoritmo per mescolare le carte non ovvio. L'algoritmo prevede che la carta di indice p nella lista venga scambiata in modo casuale

con una delle rimanenti nella lista, ovvero tra quelle in posizione $[p, \ldots, N]$ dove N è il numero di carte totali nel mazzo. Si può dimostrare che questo algoritmo garantisce un eguale probabilità per ogni carta di finire in una qualsiasi posizione nel mazzo. In alternativa avremmo potuto utilizzare la funzione `shuffle()` presente nel modulo `random`.

Infine, la classe **Poker** implementa la simulazione vera e propria. Nel costruttore della classe vengono inizializzati gli attributi della classe, ovvero, numero di giocatori (`n_giocatori`), numero di turni (`n_turni`), il mazzo di carte (`mazzo`) e la lista (inizializzata a `None`) che dovrà contenere l'insieme di carte (la mano) di ogni giocatore dopo la distribuzione. Notare la scelta delle strutture dati. Abbiamo scelto un insieme per rappresentare la singola mano di un giocatore poiché l'ordine delle carte in mano al giocatore non è affatto rilevante. Viceversa, per le mani dei giocatori e per le carte nel mazzo è stata scelta la struttura dati lista poichè l'ordine degli oggetti contenuti è rilevante in questo caso. I metodi `distribuisci_carte()` e `raccogli_carte()` servono per distribuire ai giocatori la prima mano e per raccogliere le carte alla fine della simulazione, rispettivamente. I metodi più interessanti di questa classe sono gli ultimi due. Il metodo `verifica()` itera sulle mani dei giocatori e per ognuna di esse utilizza la funzione annidata `colore()` per verificare se contiene un colore, ovvero tutte le carte dello stesso seme. In caso positivo, ritorna `True`, altrimenti ritorna `False`. Il metodo `play()` è il metodo centrale della classe e serve per simulare le partite. Esso richiama uno dopo l'altro gli altri metodi della classe e stampa ad ogni turno l'approssimazione corrente della probabilità richiesta.

6.2 Ereditarietà

Uno dei maggiori vantaggi offerti dalla programmazione orientata agli oggetti è la riutilizzabilità del codice prodotto. Il meccanismo principale della OOP che ci permette di realizzare codice estendibile e riutilizzabile è l'*ereditarietà*, ovvero la capacità di un linguaggio di programmazione di definire strutture dati (classi) come estensione di altre classi. In Python, come in tutti gli altri linguaggi orientati agli oggetti, ogni oggetto di una certa classe (*detta sotto-classe*) può *ereditare* il comportamento di base da un'altra classe (detta *super-classe*) esteso con funzionalità e attributi più specifici degli oggetti della sotto-classe.

Nello Script 6.3 definiamo una nuova classe che rappresenta gli studenti. Essendo gli studenti (evidentemente) delle comuni persone, possiamo estendere la classe `Persona` con altri attributi e metodi tipici di istanze di tipo `Studente`, per esempio il corso di laurea frequentato, l'ateneo di appartenenza, la matricola e i voti ottenuti negli esami sostenuti dallo studente. Nello script che segue definiamo la classe `Studente` come sotto-classe della classe `Persona` definita in precedenza. Oltre al costruttore, implementiamo due metodi ulteriori: il metodo `esame_sostenuto()`, che aggiunge un nuovo esame col relativo voto alla carriera dello studente e il metodo `nro_esami_sostenuti()`, che ritorna il numero di esami sostenuti dallo studente.

Script 6.3: studenti.py

```
from persone import *

class Studente(Persona):
   '''Classe Studente (sotto-classe di Persona)'''
   def __init__(self, nome, cognome, dn, corso, ateneo, matr):
      Persona.__init__(self,nome,cognome,dn)
      self.corso = corso
      self.ateneo = ateneo
      self.matricola = matr
      self.voti = {}

   def esame_sostenuto(self, esame, voto):
      '''inserisce un nuovo esame per lo studente'''
      self.voti[esame]=voto

   def nro_esami_sostenuti(self):
      '''ritorna il numero di esami dello studente'''
      return len(self.esami)
```

Da questo esempio si possono subito notare alcune cose importanti:

1. per definire una classe `B` che eredita da una classe `A` mettiamo tra parentesi il nome `A` nella definizione di `B` come in `class Studente(Persona)`;
2. il costruttore `__init__()` della classe `Studente` usa al suo interno il costruttore `__init__()` della classe `Persona` per inizializzare gli attributi dello studente in quanto persona (ovvero nome, cognome, data di nascita). Notare che, in questo caso, risulta indispensabile adottare la seconda modalità di chiamata del metodo. Alternativamente, avremmo potuto semplicemente replicare le istruzioni presenti nel costruttore della super-classe.

Di seguito vediamo una sessione d'esempio per comprendere meglio il senso dell'ereditarietà delle classi.

```
>>> carlo = Studente("Carlo", "Magno",
        (1985,7,12), "Filosofia", "Padova","10000")
>>> # qui uso metodi della classe Persona
>>> carlo.presentati()
 Ciao! sono Carlo Magno, nato il 12/07/1985
>>> # qui uso metodi propri della classe Studente
>>> carlo.esame_sostenuto("Filosofia della Scienza","30L")
>>> carlo.esame_sostenuto("Storia della Filosofia","27")
>>> carlo.nro_esami_sostenuti()
 2
```

6.2.1 Il poker è servito

Partendo dal codice della classe `Poker` della sezione precedente, proviamo ad estenderla per calcolare altri tipi di statistiche, per esempio la probabilità di

ottenere un poker servito, invece del colore. Lo vogliamo fare minimizzando la quantità di codice da aggiungere o modificare rispetto a quello già scritto. Inoltre, vogliamo progettare le classi in modo che siano estendibili ad ulteriori statistiche in futuro. Per questi motivi, decidiamo di utilizzare l'ereditarietà. L'idea è quella di definire un metodo di verifica e un metodo di stampa specifici per ogni tipologia di statistica.

Script 6.4: epoker.py

```
class Poker():
   ...

   ### i metodi verifica() e play() sostituiscono quelli
   ### nella classe Poker della sezione precedente

   def verifica(self):
      '''funzione di verifica di default'''
      return False

   def play(self):
      '''simula turni di gioco'''
      n_ok = 0
      for turno in range(1,self.n_turni+1):
         self.mazzo.mescola()
         self.distribuisci_carte()
         if self.verifica():
            n_ok+=1
         self.stampa(turno,n_ok)
         self.raccogli_carte()

class PokerColore(Poker):
   '''sotto-classe di Poker che calcola il numero di colore
      servito'''

   def __init__(self, n_giocatori, n_turni):
      '''costruttore PokerColore'''
      Poker.__init__(self,n_giocatori,n_turni)

   def verifica(self):
      '''almeno un giocatore ha un colore servito?'''
      def colore(mano):
         semi = set([carta.seme for carta in mano])
         return len(semi)==1

      for g in range(self.n_giocatori):
         if colore(self.mani[g]): return True
      return False
```

```
38
39      def stampa(self,turno,n_colore):
40          print "num turni: %d, num colore: %d, prob: %f"%(turno,
                n_colore,float(n_colore)/turno)
41
42  class PokerFour(Poker):
43      '''sotto-classe di Poker che calcola il numero di poker
            servito'''
44
45      def __init__(self, n_giocatori, n_turni):
46          '''costruttore PokerFour'''
47          Poker.__init__(self,n_giocatori,n_turni)
48
49      def verifica(self):
50          '''almeno un giocatore ha un poker servito?'''
51          def poker(mano):
52              d = {}
53              for carta in mano:
54                  if carta.valore in d:
55                      d[carta.valore]+=1
56                  else:
57                      d[carta.valore]=1
58              # d: per ogni valore nella mano ho il nro di
59              # occorrenze
60              return max(d.values())==4
61
62          for g in range(self.n_giocatori):
63              if poker(self.mani[g]): return True
64          return False
65
66      def stampa(self,turno,n_poker):
67          print "num turni: %d, num poker: %d, prob: %f"%(turno,
                n_poker,float(n_poker)/turno)
```

Nella versione dello Script 6.4, la super-classe definisce solo metodi generali utili per tutti i tipi di simulazione. Quindi vengono definite classi specifiche per la particolare simulazione che dovranno solo ridefinire il metodo `verifica()`, per la verifica della particolare proprietà sulla distribuzione delle carte, e il metodo `stampa()` per la stampa a video della particolare statistica.

6.3 Metodi speciali e overloading degli operatori

Tra i vantaggi indiscutibili della sintassi di Python possiamo annoverare sicuramente la coerenza, la potenza e la leggibilità. Il meccanismo dell'overloading degli operatori permette di utilizzare la stessa sintassi su tipi di dati diversi adattando la semantica dell'operatore al tipo di dati su cui si sta operando.

La cosa veramente interessante è che Python permette di ridefinire il comportamento dei comuni operatori anche quando essi sono applicati a classi definite dall'utente. Il meccanismo che Python utilizza è molto semplice. Assumiamo che Python si trovi una operazione di tipo `A op B` dove `A` è di tipo `T`. Per risolvere l'operazione, Python cerca il particolare metodo (detto *metodo speciale*) della classe `A` corrispondente all'operazione richiesta. Per ogni possibile operatore esiste il nome del corrispondente metodo da definire. Nelle tabelle 6.1, 6.2, 6.3, 6.4 sono presentati i più comuni metodi speciali. In particolare, in Tabella 6.1 sono presentati quelli relativi alla rappresentazione o alla stampa di stringhe che descrivono un oggetto. Nella Tabella 6.2 sono enumerati i metodi speciali utilizzati con operatori relazionali o di test booleano. La Tabella 6.3, contiene i metodi speciali per l'accesso ad un dato, in lettura o scrittura, e un metodo per la lunghezza. Infine, in Tabella 6.4, consideriamo i metodi speciali da definire se vogliamo sovraccaricare gli operatori aritmetici standard.

Tabella 6.1: Metodi speciali per la rappresentazione

metodo	operatore	funzionalità
`__str__(self)`	`str()` o `print()`	Metodo chiamato dalla funzione built-in `str()` e dalla funzione `print()`, deve ritornare una rappresentazione informale (user-friendly) dell'oggetto.
`__repr__(self)`	`` `obj` `` o `repr()`	Metodo chiamato dalla funzione built-in `repr()` o dalle virgolette rovesciate utilizzate per convertire l'oggetto in stringa (es: `print(`obj`)`). Deve ritornare una stringa indicante formalmente l'oggetto. In assenza del metodo `__str__()`, viene utilizzato questo metodo al suo posto.

Tabella 6.2: Metodi speciali per la comparazione e i test

metodo	operatore	funzionalità
`__eq__(self,other)`	`==`	Richiamato dall'operatore binario di uguaglianza `==`.
`__ne__(self,other)`	`!=`	Richiamato dall'operatore binario di disuguaglianza `!=`.
`__lt__(self,other)`	`<`	Richiamato dall'operatore binario "minore" `<`.
`__le__(self,other)`	`<=`	Richiamato dall'operatore binario "minore uguale" `<=`.
`__gt__(self,other)`	`>`	Richiamato dall'operatore binario "maggiore" `>`.
`__ge__(self,other)`	`>=`	Richiamato dall'operatore binario "maggiore uguale" `>=`.
`__nonzero__(self)`	`bool()`	Metodo chiamato durante un test (es: `if obj: pass` o dalla funzione built-in `bool()`.

Tabella 6.3: Metodi speciali per l'accesso agli elementi

metodo	operatore	funzionalità
`__len__(self)`	`len()`	Richiamato dalla funzione `len()`.
`__getitem__(self,key)`	`obj[key]`	Accesso ad un dato in lettura (`obj[key]`).
`__setitem__(self,key,v)`	`obj[key]=v`	Accesso ad un dato in scrittura (`obj[key]=v`).

Tabella 6.4: Metodi speciali per le operazioni aritmetiche

metodo	operatore	funzionalità
`__add__(self,other)`	`+`	Richiamato dall'operatore `+`.
`__sub__(self,other)`	`-`	Richiamato dall'operatore `-`.
`__mul__(self,other)`	`*`	Richiamato dall'operatore `*`.
`__div__(self,other)`	`/`	Richiamato dall'operatore `/`.
`__mod__(self,other)`	`%`	Richiamato dall'operatore `%`.

6.3.1 La classe Ratio

Allo scopo di esemplificare l'uso dei metodi speciali, diamo di seguito una implementazione della classe `Ratio` per rappresentare numeri razionali (vedi Script 6.5). Gli oggetti della classe vengono inizializzati indicando due interi, numeratore e denominatore, nel costruttore. Vogliamo poi avere la possibilità di visualizzarli in modo opportuno come stringhe e per questo utilizziamo i metodi della Tabella 6.1. Inoltre, implementiamo anche i metodi speciali utili per comparare due numeri razionali (Tabella 6.2) e/o effettuare operazioni aritmetiche su di essi (Tabella 6.4). Infine, definiamo dei metodi per modificare numeratore o denominatore di un numero razionale (Tabella 6.3).

Script 6.5: ratio.py

```
1  class Ratio():
2     ''' classe per i numeri razionali '''
3     def __init__(self,num,den=1):
4        '''costruttore Ratio'''
5        if den==1:
6           self.num, self.den = num, 1
7           return
8        if den < 0:
9           self.num, self.den = -num, -den
10       else:
11          self.num, self.den = num, den
12       mcd = self.__mcd(abs(num),abs(den))
13       self.num /= mcd
14       self.den /= mcd
15    def __repr__(self):
16       '''rappresentazione di un numero razionale'''
17       return "Ratio(%d,%d)"%(self.num,self.den)
18    def __str__(self):
19       '''stringa di un numero razionale'''
20       if self.den!=1:
21          return "%d/%d"%(self.num,self.den)
22       return "%d"%self.num
23    def __eq__(self,other):
24       '''i due razionali sono equivalenti?'''
```

```
25        return float(self.num)/self.den == float(other.num)/
              other.den
26    def __lt__(self,other):
27        '''e' minore?'''
28        return float(self.num)/self.den < float(other.num)/
              other.den
29    def __le__(self,other):
30        '''e' minore uguale?'''
31        return float(self.num)/self.den <= float(other.num)/
              other.den
32    def __gt__(self,other):
33        '''e' maggiore?'''
34        return float(self.num)/self.den > float(other.num)/
              other.den
35    def __ge__(self,other):
36        '''e' maggiore uguale?'''
37        return float(self.num)/self.den >= float(other.num)/
              other.den
38    def __nonzero__(self):
39        '''True se diverso da 0 altrimenti False'''
40        return self.num
41    def __neg__(self):
42        '''cambia segno'''
43        return Ratio(-self.num,self.den)
44    def __add__(self,other):
45        '''somma di due numeri razionali'''
46        return Ratio(self.num*other.den+other.num*self.den,self
              .den*other.den)
47    def __sub__(self,other):
48        '''sottrazione di due numeri razionali'''
49        return Ratio(self.num*other.den-other.num*self.den,self
              .den*other.den)
50    def __mul__(self,other):
51        '''moltiplicazione di due numeri razionali'''
52        return Ratio(self.num*other.num,self.den*other.den)
53    def __div__(self,other):
54        '''divisione di due numeri razionali'''
55        return Ratio(self.num*other.den,self.den*other.num)
56    def __getitem__(self,key):
57        '''r['num'] -> numeratore, r['den'] -> denominatore'''
58        if key=="num": return self.num
59        if key=="den": return self.den
60    def __setitem__(self,key,value):
61        '''r['num'] = v o r['den'] = v e aggiorna il mcd'''
62        if key=="num": self.__init__(value,self.den)
63        if key=="den": self.__init__(self.num,value)
```

```
64      def __mcd(self,x,y):
65          '''metodo privato per il calcolo del mcd'''
66          if x==y: return x
67          elif x>y: return self.__mcd(x-y,x)
68          return self.__mcd(x,y-x)
```

Più nel dettaglio descriviamo un numero razionale con numeratore e denominatore, due interi. In aggiunta, abbiamo un metodo privato dell'oggetto `__mcd()` che calcola il MCD tra due numeri interi positivi con l'algoritmo di Euclide. Il costruttore della classe `Ratio` prende come argomenti due numeri interi, numeratore e eventualmente denominatore, e crea il corrispondente numero razionale. Quando il denominatore non è indicato si intende creare un numero intero. Altrimenti il costruttore prova a semplificare e normalizzare il numero in modo da avere denominatore positivo e numeratore e denominatore primi tra di loro. Siano `n` e `d` il numeratore e il denominatore del numero razionale, rispettivamente. La rappresentazione `repr()` di un numero è del tipo `'Ratio(n,d)'`. La versione `str()` stampabile del numero razionale è invece la stringa `'n/d'` o semplicemente `'n'` quando il numero è intero. Tutti gli operatori di confronto si basano sulla conversione a numero reale dei numeri razionali da confrontare. Infine, abbiamo definito anche l'accesso agli elementi (numeratore e denominatore del numero) sia in lettura che in scrittura. Notare che, per il secondo caso, viene richiamato il costruttore con gli opportuni parametri.

Proviamo ad effettuare una sessione di test per verificare empiricamente la correttezza della classe appena definita.

```
>>> from ratio import *
>>> r1 = Ratio(2,3)
>>> print(r1)
 2/3
>>> r2 = Ratio(-4,2)
>>> print(r2)
 -2
>>> r3 = Ratio(-2,-16)
>>> print(r3)
 1/8
>>> r1 + r2 - r3
 Ratio(-35,24)
>>> r1 * r2 / r3
 Ratio(-32,3)
>>> r1 > r3
 True
>>> r2 >= r3
 False
>>> r1['num'] = 6
>>> -r1 == r2
 True
```

Esercizi del capitolo

Esercizio 6.1

Si consideri il problema di progettare un applicazione Calendario. Definire le seguenti classi

- `Appuntamento` *con le informazioni relative ad un appuntamento che si dovrà tenere in un determinato giorno, a partire da un certo orario, fino ad un altro orario*;
- `Evento` *con le informazioni relative ad un evento associato ad un giorno intero*;
- `Attivita` *con le informazioni relative ad una attività su vari giorni consecutivi*.

Definire strutture dati appropriate per contenere separatamente l'elenco di appuntamenti, eventi e attività di una persona. Definire funzioni a piacere per effettuare l'inserimento di nuove informazioni (appuntamenti, eventi e attività) per la loro ricerca/cancellazione in un certo intervallo temporale. Non è prevista la possibilità che più appuntamenti possano sovrapporsi. Lo stesso dicasi per gli eventi e per le attività. In fase di inserimento e cancellazione si faccia attenzione che i dati immessi siano coerenti. Per esempio, non deve essere possibile inserire un appuntamento con orario di fine minore dell'orario di inizio.

Esercizio 6.2

L'azienda di trasporti su rotaia TRENIPADANI *ha bisogno di un programma Python che gestisca le prenotazioni dei passeggeri sui loro treni. Si definiscano le classi:*

- `Passeggero` *con le informazioni relative ad un passeggero*
- `Treno` *con le informazioni su un treno in particolare*
- `Posto` *con le informazioni relative ad un posto in un certo treno*
- `Prenotazioni` *che contiene le prenotazioni dei vari passeggeri, p.e. coppie passeggero-posto.*

Definire metodi a piacere per effettuare le prenotazioni. Considerare eventualmente che il treno ha un numero di posti massimo nelle varie classi (prima e seconda). Dare la possibilità di effettuare ricerche sulle prenotazioni (per esempio l'elenco di prenotazioni in un certo treno o per una certa classe, o di una certa persona).

Esercizio 6.3

Definire una semplice classe `Vettore2D` *che descriva un vettore in un piano cartesiano (coordinate x e y). Si diano almeno i seguenti metodi:*

1. `somma(v)`*: somma con un altro Vettore2D*

2. `smul(a)`: *moltiplicazione per uno scalare e*
3. `lunghezza()`: *lunghezza del vettore*

Indicare eventuali altri metodi opportuni.

Esercizio 6.4

Data la classe `Punto2D` *definita come segue:*

```
class Punto2D:
   def __init__(self,x,y):
      self.x = x
      self.y = y
```

Definire una classe `Rettangolo`*. Una istanza di* `Rettangolo` *viene costruita passando i due punti (di tipo* `Punto2D`*) in alto a sinistra e in basso a destra nel piano cartesiano. Definire i seguenti metodi per la classe:*

1. `Perimetro()`: restituisce il perimetro del rettangolo;
2. `Area()`: restituisce l'area del rettangolo.
3. `Interseca()`: dato un altro rettangolo, ritorna `True` o `False` a seconda che i due rettangoli si intersechino o meno.
4. `Intersezione()`: dato un altro rettangolo, modifica l'istanza in modo da rappresentare il rettangolo intersezione dei due.
5. `Centro()`: restituisce il punto medio (`Punto2D`) del rettangolo.

7

Strutture dati ricorsive

In questo capitolo discuteremo le due principali strutture dati ricorsive:

- liste concatenate;
- alberi binari

e studieremo come gestirle in maniera efficiente.

7.1 Liste concatenate

Una *lista concatenata* (o *lista linkata*) è una sequenza di oggetti, detti nodi della lista, concatenati uno all'altro in modo sequenziale. Ogni nodo della lista, oltre al dato vero e proprio da memorizzare, contiene un collegamento al nodo successivo nella lista. Seguendo questi collegamenti, da un qualsiasi nodo della lista concatenata, sarà possibile raggiungere un qualsiasi altro nodo successivo.

Python già prevede il tipo di dato `list` che è molto simile, e per certi versi più versatile, di quello che stiamo ora definendo. Il contenuto di questa sezione comunque non è inutile in quanto: *(i)* non tutti i linguaggi di programmazione prevedono un tipo lista, *(ii)* la lista concatenata rappresenta un primo esempio (forse il più semplice) di struttura dati definita ricorsivamente.

Definiamo formalmente e ricorsivamente una lista concatenata:

- una lista concatenata vuota (rappresentata in Python da `None`) è una lista concatenata;
- un nodo che contiene un *valore* (o *dato*) e un riferimento ad una lista concatenata è una lista concatenata.

Lo script 7.1 contiene la definizione della classe `Lista` e svariate funzioni utili per gestire le liste concatenate. Le vedremo una ad una in dettaglio nelle prossime sezioni.

Script 7.1: liste.py

```
class Lista():
   def __init__(self, val=None, next=None):
      self.val = val
      self.next = next

### funzioni di utilita'

def is_vuota(lista):
   return lista==None

def nro_nodi(lista):
   if is_vuota(lista):
      return 0
   return 1 + nro_nodi(lista.next)

def stampa_lista(lista):
   if not is_vuota(lista):
      print lista.val,
      stampa_lista(lista.next)

def stampa_lista_inv(lista):
   if not is_vuota(lista):
      stampa_lista_inv(lista.next)
      print lista.val,

### funzioni di gestione

def aggiungi_in_testa(lista,val):
   if is_vuota(lista):
      return Lista(val)
   return Lista(val,lista)

def aggiungi_in_coda(lista,val):
   if is_vuota(lista):
      return Lista(val)
   lista.next = aggiungi_in_coda(lista.next,val)
   return lista

def aggiungi_in(lista,ind,val):
   if ind==0:
      return aggiungi_in_testa(lista,val)
   if is_vuota(lista):
      return None
   lista.next = aggiungi_in(lista.next,ind-1,val)
   return lista
```

```
46
47 def lista_inversa(lista):
48     if is_vuota(lista):
49         return None
50     elle = lista_inversa(lista.next)
51     return aggiungi_in_coda(elle,lista.val)
```

7.1.1 Definizione della classe lista concatenata

Nella definizione della classe `Lista` si sono dichiarati due parametri opzionali nel costruttore della classe `__init__()`: il valore `val` (con default `None`) e il riferimento al nodo successivo `next` (anch'esso con default `None`), ovvero il resto della lista. Il costruttore non fa altro che inizializzare i membri `val` e `next` dell'oggetto di tipo `Lista`.

Utilizzando la definizione appena data possiamo già generare semplici liste concatenate.

```
>>> lista1 = Lista("A")
>>> lista2 = None
>>> lista3 = Lista("B", lista1)
```

In questo esempio si sono create tre liste concatenate. La prima lista (`lista1`) è stata creata utilizzando il costruttore della classe `Lista` con un solo parametro, modalità che ci permette di creare una lista con un unico nodo. Nel secondo caso (`lista2`) abbiamo creato una lista vuota. Infine, nel terzo caso, abbiamo creato una nuova lista di due elementi (`lista3`) passando al costruttore l'informazione sul primo nodo e la lista `lista1` precedentemente creata. Il diagramma di stato dopo queste tre righe di codice sarà quello di Figura 7.1.

7.1.2 Funzioni utili per le liste concatenate

La prima funzione che andiamo a definire [righe 8-9] ci permette di verificare se una lista concatenata è vuota. La funzione `is_vuota()` prende come parametro una lista concatenata e ritorna `True` se l'oggetto passato come argomento ha valore `None` e `False` altrimenti.

La funzione `nro_nodi()` [righe 11-14] ci permette di calcolare il numero di nodi di una lista concatenata. L'implementazione ricorsiva segue lo schema

$$N([]) = 0,\ N([x|L]) = 1 + N(L).$$

Ovvero, la lista vuota ha numero di nodi uguale a zero; se la lista non è vuota posso vederla come una concatenazione di un nodo x e una lista L e il numero di nodi totali sarà uguale a uno più il numero di nodi della lista concatenata L.

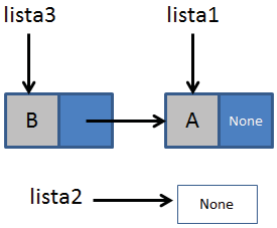


Figura 7.1: Primo esempio di liste concatenate

La funzione `stampa_lista()` [righe 16-19] stampa a video una lista concatenata. Anche in questo caso l'implementazione è ricorsiva. La ricorsione è in avanti. Se la lista è vuota non c'è niente da stampare. Altrimenti si stampa il primo elemento e ricorsivamente tutto il resto della lista.

Infine, la funzione `stampa_lista_inv()` [righe 21-24] stampa a video una lista concatenata nel senso inverso. L'implementazione è ancora ricorsiva ma la ricorsione è all'indietro. Se la lista è vuota non c'è niente da stampare; altrimenti, stampiamo ricorsivamente tutto il resto della lista concatenata in senso inverso e successivamente il valore nel primo nodo.

7.1.3 Funzioni per la gestione di liste concatenate

La funzione `aggiungi_in_testa()` [righe 28-31] aggiunge un nodo con un determinato valore (`val`) all'inizio di una lista (`lista`). La sua definizione ricorsiva è così definita: se la lista è vuota, si ritorna una nuova lista con un unico nodo avente valore `val`; altrimenti, costruiamo un nuovo oggetto `Lista` con valore `val` e campo `next` uguale alla lista originale, e lo ritorniamo.

La funzione `aggiungi_in_coda()` [righe 33-37] aggiunge un nodo con un determinato valore (`val`) in fondo ad una lista (`lista`). La sua definizione ricorsiva è data seguendo lo schema ricorsivo:

$$C([], v) = [v],\ C([x|L]) = [x|C(L)].$$

Ovvero, se la lista è vuota, si ritorna una nuova lista con un unico nodo avente valore `val` come nel caso dell'inserimento in testa; altrimenti, se esiste almeno un nodo, ricorsivamente aggiungiamo il valore `val` in coda al resto della lista e modifichiamo opportunamente il campo `next` della lista originale.

La funzione `aggiungi_in()` [righe 39-45] è utile per inserire un valore (`val`) in una certa posizione (`ind`) della lista concatenata. Qui abbiamo tre casi differenti da gestire. Nel primo caso base (`ind==0`) vogliamo inserire il nuovo elemento in posizione `0` ovvero in testa alla lista; per cui possiamo riutilizzare la funzione vista in precedenza per l'inserimento in testa. Nel secondo caso

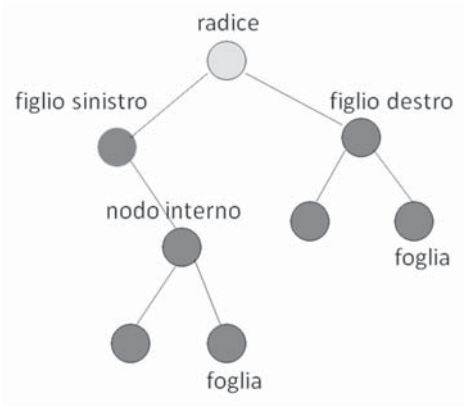


Figura 7.2: Rappresentazione di un albero binario e terminologia di base.

base, viene richiesto l'inserimento in un indice maggiore di 0 per una lista vuota (`is_vuota(lista)`). In questo caso dovremo notificare l'errore all'ambiente chiamante. Lo facciamo restituendo `None`. Infine, abbiamo il caso ricorsivo nel quale si implementa lo schema di ricorsione:

$$A([x|L], i, v) = [x|A(L, i-1, v)],$$

il quale in pratica significa che aggiungere un elemento in posizione i di una lista vuol dire aggiungere lo stesso elemento in posizione $i-1$ della lista ottenuta escludendo il primo nodo.

Infine, la funzione `lista_inversa()` [righe 47-51] ritorna una nuova lista con i valori dei nodi invertiti. Sia C la funzione che aggiunge in coda come definita precedentemente, lo schema ricorsivo utilizzato in questo caso è:

$$I([]) = [], I([v|L]) = C(I(L), v).$$

7.2 Alberi binari

Una struttura dati albero binario può essere definita formalmente e ricorsivamente nel modo seguente:

- un albero vuoto (rappresentato in Python da `None`) è un albero binario;
- un nodo che contiene un *valore*, linkato con due alberi binari (destro e sinistro) è un albero binario.

In Figura 7.2 è rappresentato un albero binario e la terminologia comunemente utilizzata per gli alberi binari. Il collegamento tra due nodi (detto *arco*) è diretto dal *nodo padre* al *nodo figlio*. L'unico nodo che non ha nodo padre nell'albero si chiama *nodo radice*. Un nodo che non ha né figlio destro né figlio sinistro

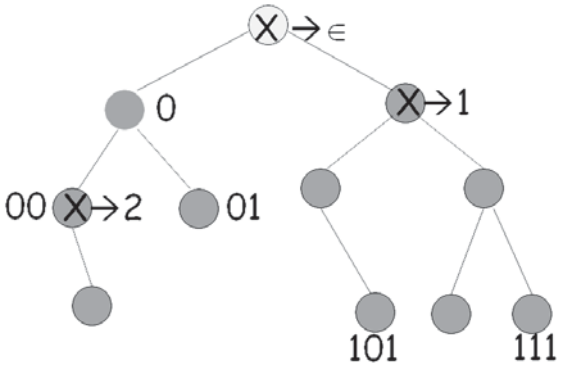


Figura 7.3: Cammini di un albero binario. Per alcuni nodi è anche specificata la profondità (ovvero, il numero di cifre binarie neccessarie per descrivere il cammino associato).

prende il nome di *nodo foglia*. Viceversa, se ha almeno uno dei due figli allora è un *nodo interno*.

Si dice *cammino* un percorso che dalla radice raggiunge un certo nodo. Un cammino in un albero binario può essere visto come una sequenza di simboli binari (per esempio $0/1$ o l/r) che descrive in modo univoco come sia possibile raggiungere il nodo a partire dalla radice. In Figura 7.3 per alcuni nodi di un albero di esempio è data la sequenza di $0/1$ che corrisponde al cammino corrispondente. La *profondità* (o *livello*) di un nodo è intuitivamente la distanza di quel nodo dalla radice. Possiamo definirla ricorsivamente nel seguente modo: il nodo radice ha profondità uguale a 0; se un nodo ha profondità n, allora i figli hanno profondità $n + 1$. Possiamo altresí notare che la profondità di un nodo corrisponde sempre al numero di simboli necessari per descrivere il cammino per raggiungere quel nodo. Ancora, si dice *altezza* di un albero (o di un sotto-albero) la lunghezza massima dei suoi cammini (o equivalentemente la profondità massima dei nodi dell'albero). Possiamo dare una definizione ricorsiva di altezza usando la quale risulta più efficiente calcolarne il valore: l'altezza di un albero con un solo nodo è 0 e ogni albero ha un'altezza che è uguale all'altezza del più alto dei suoi sotto-alberi più 1. Infine, un albero binario è detto *completo* se, sia h la sua altezza, allora ha esattamente $2^{h+1} - 1$ nodi. Un albero completo può anche essere definito equivalentemente come un albero binario tale che *per ogni suo nodo* vale la proprietà che i sotto-alberi sinistro e destro hanno uguale altezza.

7.2.1 Definizione della classe albero binario

Lo script 7.2 contiene la definizione della classe `Albero` Nella definizione della classe si dichiarano tre parametri opzionali nel costruttore della classe `__init__()`: il valore del nodo radice `val` (con default `None`) e il riferimento ai due nodi figli, ovvero ai sotto alberi sinistro (`sx`) e destro (`dx`). Come per il caso delle liste con-

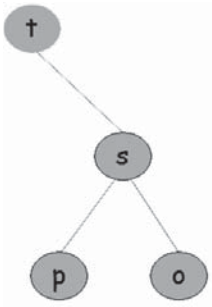


Figura 7.4: Semplice albero binario di esempio.

catenate, il costruttore non fa altro che inizializzare i membri dell'oggetto di tipo `Albero`.

Script 7.2: alberi.py

```
1 class Albero():
2    def __init__(self,val=None,sx=None,dx=None):
3       self.val = val
4       self.sx = sx
5       self.dx = dx
```

Per esempio, la seguente serie di comandi produce l'albero in Figura 7.4.

```
>>> P = Albero('p')
>>> O = Albero('o')
>>> S = Albero('s',P,O)
>>> T = Albero('t',None,S)
```

7.2.2 Funzioni utili per alberi binari

Lo Script 7.3 contiene la definizione di funzioni di utilità per la classe `Albero`. Come per il caso delle liste, la prima funzione che andiamo a definire è la funzione che ci permette di verificare se un albero è vuoto. La funzione `is_vuoto()` [righe 6-7] prende come parametro un albero e ritorna `True` se l'oggetto passato come argomento ha valore `None` e `False` altrimenti.

La funzione `is_foglia()` [righe 9-10], dato un albero come argomento, verifica invece se tale albero è una foglia. Anche in questo caso, la verifica risulta molto semplice, ovvero verifichiamo se entrambi, sotto-albero sinistro e destro, sono vuoti. In questo caso, restituiamo `True` o restituiamo `False` altrimenti.

La funzione `nro_nodi()` [righe 12-15], dato un albero come argomento, calcola il numero di nodi dell'albero. Il numero dei nodi è calcolato ricorsivamente seguendo lo schema ricorsivo:

$$N((v, L, R)) = N(L) + N(R) + 1$$

ovvero il numero di nodi di un albero non vuoto risulta uguale al numero di nodi del suo sotto-albero sinistro più il numero di nodi del suo sotto-albero destro più 1; se l'albero invece è vuoto, banalmente, il numero di nodi è uguale a 0.

La funzione `altezza()` [righe 17-20], dato un albero come argomento, ne calcola l'altezza. Qui utilizziamo uno schema ricorsivo ispirato alla definizione di altezza che abbiamo proposto precedentemente. In particolare, definiamo l'altezza di un albero vuoto uguale a -1 (caso base) e usiamo lo schema ricorsivo

$$A((v, L, R)) = \max(A(L), A(R)) + 1$$

per il calcolo dell'altezza di un albero non vuoto (caso ricorsivo).

Script 7.3: alberi.py

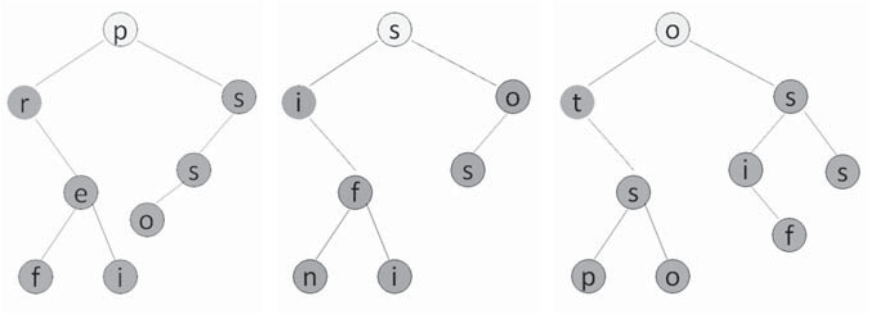
```
6  def is_vuoto(albero):
7     return albero==None
8
9  def is_foglia(albero):
10    return albero.sx==None and albero.dx==None
11
12 def n_nodi(albero):
13    if is_vuoto(albero):
14       return 0
15    return n_nodi(albero.sx)+n_nodi(albero.dx)+1
16
17 def altezza(albero):
18     if is_vuoto(albero):
19         return -1
20     return max(altezza(albero.sx),altezza(albero.dx))+1
```

7.2.3 Visite

Molti algoritmi operanti su alberi binari richiedono di esplorare (visitare) tutti i nodi dell'albero in modo sistematico, ovvero percorrendo tutti e soli i nodi una sola volta. Al contrario delle liste, dove esiste un ordine di visita standard dal primo all'ultimo elemento della lista, nel caso degli alberi, non esiste un ordine di visita "migliore" di un altro.

In questa sezione, ci occuperemo brevemente delle cosiddette visite in profondità: visita *pre-ordine* (o *prefissa*), visita *in-ordine* (o *infissa*) e visita *post-ordine* (o *postfissa*). In Figura 7.5 sono presentate graficamente le tre visite e nello Script 7.4, per ognuna di esse, ne viene data una implementazione che ritorna la lista dei nodi dell'albero in ordine rispetto a quella specifica visita.

Nella visita pre-fissa, prima viene visitata la radice dell'albero, poi tutto il sotto-albero sinistro ricorsivamente, infine tutto il sotto-albero destro. Nella visita in-fissa, prima viene visitato il sotto-albero sinistro, poi la radice e infine il sotto-albero destro. Infine, nella visita post-fissa, prima viene visitato il sotto-albero sinistro, poi il sotto-albero destro, infine la radice.



(a) (b) (c)

Figura 7.5: (a) visita prefissa, (b) visita infissa, (c) visita postfissa

Script 7.4: alberi.py

```
21 def visita_pre(a):
22     if is_vuoto(a):
23         return []
24     return [a] + visita_pre(a.sx) + visita_pre(a.dx)
25
26 def visita_in(a):
27     if is_vuoto(a):
28         return []
29     return visita_in(a.sx) + [a] + visita_in(a.dx)
30
31 def visita_pos(a):
32     if is_vuoto(a):
33         return []
34     return visita_pos(a.sx) + visita_pos(a.dx) + [a]
```

7.2.4 Funzioni per la visualizzazione di alberi binari

Nello Script 7.5 sono proposte due funzioni utili per visualizzare gli alberi.

La prima funzione `stampa()` [righe 35-39] effettua una stampa a video indentando per livelli. La struttura ricorsiva è abbastanza semplice. In particolare, andando in profondità nell'albero aumentiamo l'indentazione della stampa. La stampa avviene seguendo una visita infissa inversa, vale a dire prima il sottoalbero destro, poi la radice, poi il sinistro. In questo modo l'albero sarà stampato ruotato di 90 gradi in senso anti-orario e perfettamente leggibile.

```
>>> stampa(T)
    o
  s
    p
t
```

La seconda funzione `as_str()` serve per "linearizzare" un albero binario, ovvero restituisce una stringa unidimensionale nel formato

```
(valore-radice,sotto-albero-sx,sotto-albero-dx)
```

seguendo una visita prefissa dell'albero. Per l'albero binario d'esempio utilizzato precedentemente otterremo:

```
>>> as_str(T)
'(t,_,(s,(p,_,_),(o,_,_)))'
```

Script 7.5: alberi.py

```
35 def stampa(albero,livello=0):
36     if albero==None: return
37     stampa(albero.dx,livello+1)
38     print '  '*livello + str(albero.val)
39     stampa(albero.sx,livello+1)
40
41 def as_str(albero):
42     if not albero:
43         return "_"
44     s = "(" + str(albero.val) + ","
45     s += as_str(albero.sx) + "," + as_str(albero.dx)
46     s += ")"
47     return s
```

Infine, nello Script 7.6 viene definita una funzione `crea_random()` per la creazione di un nuovo albero "casuale". In particolare, la funzione risulta ricorsiva. Ad ogni livello della ricorsione, con probabilità via via decrescente, si decide o meno di creare un nodo con etichetta scelta casualmente tra quelle disponibili in un certo alfabeto.

Script 7.6: alberi.py

```
48 def crea_random(pr, decay, alphabet):
49     from random import random
50     if random()<=pr:
51         albero = Albero(r.choice(alphabet),None,None)
52         pr*=decay
53         albero.sx = crea_random(pr, decay, alphabet)
54         albero.dx = crea_random(pr, decay, alphabet)
55         return albero
56      return None
```

Esercizio risolto. *Dare una funzione `somma()` che, preso un albero binario come argomento, ritorna la somma dei valori dei suoi nodi. Indicare pre-condizioni e post-condizioni e dimostrare la correttezza.*

```
### PRE: albero e' None o un albero binario definito come
### nello Script 7.2 e l'attributo val e' di tipo numerico
### POS: ritorna la somma dei valori sui nodi dell'albero
### (0 se l'albero e' vuoto)
def somma(albero):
   if is_vuoto(albero):
      return 0
   return albero.val+somma(albero.sx)+somma(albero.dx)
```

Per dimostrarne la correttezza supponiamo che le pre-condizioni siano vere e vogliamo dimostrare induttivamente le post-condizioni. Assumiamo la verità delle pre-condizioni su `albero`.

1. `POS(somma(None))`: banalmente ritorna 0 quando `albero==None`;
2. `PRE(albero.sx)` e `PRE(albero.dx)`: vero poiché l'albero binario `albero` è ben definito e i due sotto-alberi (`albero.sx` e `albero.dx`) sono ancora alberi binari ben definiti;
3. `POS(somma(albero))`: assumendo l'albero non vuoto e le post-condizioni di `somma()` vere sui sotto-alberi sinistro e destro, allora la riga 8 ritorna la somma dei valori sui nodi del sotto-albero sinistro più la somma dei valori sui nodi del sotto-albero destro più il valore sul nodo radice, che è esattamente la somma dei valori sui nodi di tutto l'albero.

Esercizi del capitolo

Esercizio 7.1

Data la definizione seguente:

```
class Al:
   def __init__(self,val=None,sx=None,dx=None):
      self.val = val
      self.sx = sx
      self.dx = dx
```

descrivere dettagliatamente le pre e post condizioni delle seguenti funzioni. Esemplificare la loro invocazione con esempi sensati e valori opportuni dei parametri.

```
def lin_a(a):
   if not a:
      return None
   return [a.val,lin_a(a.sx),lin_a(a.dx)]

def B(c,a):
   if not a:
      return Al(c)
   if c<a.val:
```

```
      return Al(a.val,B(c,a.sx),a.dx)
   return Al(a.val,a.sx,B(c,a.dx))

def A(s):
   if not s:
      return None
   return B(s[-1],A(s[:-1]))
```

Indicare il valore stampato dai seguenti comandi.

```
print lin_a(A("ciao"))
print lin_a(A("papero"))
```

Esercizio 7.2

Descrivere dettagliatamente le pre e post condizioni della seguente funzione. Dare inoltre una dimostrazione per induzione della correttezza.

```
def S(c,d):
   if not (c and d):
      return 0
   return (c.val + d.val + S(c.sx,d.sx) + S(c.dx,d.dx))
```

Esercizio 7.3 *Descrivere dettagliatamente le pre e post condizioni delle seguenti funzioni. Dare almeno due esempi con output diverso. Dimostrare per induzione la correttezza del programma rispetto alle pre/post definite in precedenza.*

```
def A(a,t):
   if not t and not a: return True
   if not t: return False
   b = True
   if a.sx:
      b = A(a.sx,t)
   if not b or t[0]!=a.val:
      return False
   del t[0]
   if a.dx:
      b = A(a.dx,t)
   return b
```

Esercizio 7.4 *Descrivere dettagliatamente le pre e post condizioni della seguente funzione. Esemplificare la loro invocazione con un esempio sensato e valori opportuni ai parametri. Dimostrare per induzione la correttezza del programma rispetto alle pre/post definite in precedenza.*

```
def A(a):
   if not a.left and not a.right:
      return 0.0
   return A(a.left) + A(a.right) + 1
```

Esercizio 7.5 *Descrivere dettagliatamente le pre e post condizioni della seguente funzione. Esemplificare la loro invocazione con un esempio sensato e valori opportuni ai parametri. Dimostrare per induzione la correttezza del programma rispetto alle pre/post definite in precedenza.*

```
def A(a):
   if not a:
      return (0.0,[])
   asx = A(a.sx)
   adx = A(a.dx)
   if asx[0]>=adx[0]:
      return (a.val+asx[0],[0]+asx[1])
   else:
      return (a.val+adx[0],[1]+adx[1])
```

8

Algoritmi

In questo capitolo presenteremo gli strumenti di analisi utilizzati per valutare gli algoritmi e faremo una carrellata su quelli che sono gli algorimi notevoli in informatica. In particolare, discuteremo di:

- complessità computazionale
- ordinamenti
- ricerca per chiave
- ricerca su liste concatenate
- ricerca su alberi binari
- tecnica del backtracking

8.1 Complessità computazionale

In informatica, la complessità computazionale rappresenta una misura dell'efficienza di un algoritmo nel risolvere un dato problema. Tale efficienza si può misurare seguendo diverse dimensioni, quali il tempo di esecuzione o la memoria richiesta da un algoritmo.

Per poter valutare le prestazioni di un algoritmo bisogna prima di tutto definire una misura che sia indipendente dalla tecnologia utilizzata. Questo perché lo stesso programma che implementa un certo algoritmo fatto eseguire su due calcolatori, uno del 1990 e l'altro dei giorni nostri, richiederà un tempo di esecuzione molto diverso. Diamo quindi una definizione di complessità computazionale in tempo che dipende dal numero di passi elementari dell'algoritmo e dalla dimensione dell'input: *dato un algoritmo, si dice che questo opera in tempo* $f(n)$ *se, dato un input di dimensione* n*, produce un output in* $f(n)$ *passi.*

Siamo quindi interessati a studiare il numero di passi richiesti da un algoritmo al crescere della dimensione dell'input. In realtà, potrebbe succedere di avere informazioni aggiuntive riguardo al tipo di input che mi posso aspettare. Per questo motivo, generalmente si studia la complessità nel caso *ottimo*, *pessimo* o *medio*.

- **caso ottimo o migliore**: nel quale i dati presentati sono i più favorevoli per l'algoritmo;
- **caso pessimo o peggiore**: nel quale i dati presentati sono i più sfavorevoli per l'algoritmo;
- **caso medio**: comportamento in media al variare dei dati possibili in ingresso.

Diremo che la complessità di un algoritmo è:

$O(f(n))$ che si legge "o di $f(n)$" quando vogliamo dare una valutazione per il caso pessimo, ovvero per dire che la quantità di risorse richiesta cresce *non più* di $f(n)$ al crescere di n;

$\Omega(g(n))$ che si legge "omega di $g(n)$" quando vogliamo dare una valutazione per il caso ottimo, ovvero per dire che la quantità di risorse richiesta cresce *non meno* di $g(n)$ al crescere di n;

$\Theta(h(n))$ che si legge "theta di $h(n)$" quando vogliamo dire che i casi ottimo e pessimo hanno simili prestazioni, ovvero che la quantità di risorse richiesta cresce *come* $h(n)$ al crescere di n.

8.1.1 Complessità in tempo

Possiamo dare alcune semplici regole utili a calcolare l'ordine di grandezza del numero di operazioni richieste dall'esecuzione di un certo algoritmo nel caso pessimo. Qui usiamo il linguaggio Python come linguaggio di descrizione di algoritmi ma regole simili potrebbero essere fornite per qualsiasi altro linguaggio formale.

1. Le operazioni aritmetico-logiche e di assegnamento hanno costo computazionale costante;
2. una sequenza di comandi ha costo pari alla somma dei costi dei singoli comandi;
3. Il costrutto di selezione:

```
if COND_A :
   BLOCCO_A
elif COND_B :
   BLOCCO_B
...
else :
   BLOCCO_ELSE
```

   ha costo computazionale nel caso pessimo dato da

```
c(IF-ELIF-ELSE) = c(COND_A) + c(COND_B) + ... +
       max(c(BLOCCO_A),c(BLOCCO_B), ... ,c(BLOCCO_ELSE))
```

4. il costrutto di iterazione:

```
while COND :
   BLOCCO
```

   ha costo computazionale nel caso pessimo che dipende dal numero massimo di volte che la condizione risulta vera (che chiamiamo `m`) moltiplicato per la somma dei costi computazionali della condizione e del blocco, ovvero

```
c(WHILE) = m * (c(COND) + c(BLOCCO))
```

   dove, per semplificare, abbiamo assunto che il costo della valutazione della condizione e l'esecuzione del blocco non vari con le differenti iterazioni. Tale assunzione non è sempre ragionevole. Quando non lo è, occorrerà sostituire la moltiplicazione con la somma dei costi delle singole iterazioni;

5. l'iteratore:

```
for x in IT :
   BLOCCO
```

   ha costo computazionale nel caso pessimo che dipende da `len(IT)` e vale

```
c(FOR) = len(IT) * c(BLOCCO)
```

   assumendo anche in questo caso che l'esecuzione del blocco abbia lo stesso costo sulle varie iterazioni;

6. le chiamate a funzioni (metodi, o operatori) hanno costo uguale a quello del corpo della stessa funzione più quello dovuto al calcolo degli argomenti passati alla funzione.

8.1.2 Complessità in spazio

Considerazioni simili a quelle appena fatte per la complessità in tempo possono essere fatte per la complessità in spazio, troppe volte tralasciata ingiustamente. Tale costo rappresenta generalmente lo spazio in memoria utilizzato dal nostro algoritmo relativamente alla grandezza dei dati di ingresso.

Facciamo un semplice esempio di programma dove vogliamo calcolare il valore massimo di una serie di interi positivi inseriti dall'utente. Un primo modo per farlo è utilizzare una lista di appoggio dove andiamo ad inserire via via gli elementi inseriti dall'utente. Alla fine dell'inserimento, possiamo calcolare il massimo con la funzione `max()` sulla lista.

```
dato_in = int(raw_input())
L = []
while dato_in:
   L.append(dato_in)
   dato_in = int(raw_input())
print("Massimo elemento inserito: %d"%max(L))
```

Notiamo che, in questa soluzione, se l'utente inserisce n elementi, il programma necessiterà di spazio in memoria proporzionale a n per contenere la lista. Tale n potrebbe essere anche molto grande in applicazioni reali. Nella soluzione che segue, invece, aggiorniamo il massimo via via che l'utente inserisce nuovi dati e non avremo bisogno di una struttura dati ingombrante come la lista di appoggio.

```
dato_in = int(raw_input())
M = dato_in
while dato_in:
   if dato_in > M:
      M = dato_in
   dato_in = int(raw_input())
print("Massimo elemento inserito: %d"%M)
```

Quello appena presentato rappresenta un semplice esempio che mostra come, a parità di complessità in tempo, una soluzione che richieda ordini di grandezza di memoria in meno è decisamente preferibile.

8.2 Algoritmi di ordinamento

Gli algoritmi di ordinamento di sequenze sono storicamente tra gli algoritmi più studiati dell'informatica. Ciò è dovuto sia all'importanza del problema in sé sia alla semplicità degli algoritmi proposti a questo scopo che rendono generalmente possibile uno studio teorico approfondito della loro complessità computazionale.

Ordinare una sequenza di elementi su cui sia definita una relazione di ordine totale (riflessiva, antisimmetrica e transitiva su ogni coppia di elementi) significa generare una permutazione della sequenza in modo da rispettare tale relazione. Il motivo principale per cui si ordina una sequenza è che in una sequenza di valori ordinata diventa poi più semplice effettuare ricerche. Chiaramente la procedura di ordinamento non è indolore in quanto ha un costo computazionale. Quindi, la convenienza o meno dell'avere una sequenza preventivamente ordinata dipenderà dal numero di ricerche che si prevede di dover fare.

Nel seguito presenteremo una serie di algoritmi di ordinamento di sequenze e ci concentreremo su sequenze di valori naturali da ordinare secondo la relazione *minore uguale*, ovvero da ordinare in modo non decrescente. Tali algoritmi possono essere facilmente modificati in modo da considerare diversi tipi di valori per gli elementi (p.e. reali, stringhe, caratteri, ecc.) e anche relazioni d'ordine diverse (p.e. ordine *maggiore uguale* per i reali, la lunghezza per le stringhe o l'ordine *lessicografico* per i caratteri).

La complessità computazionale in tempo degli algoritmi di ordinamento si basa sulla valutazione del numero di operazioni elementari necessarie (confronti, scambi) in funzione del numero n di elementi della sequenza.

Gli algoritmi di ordinamento sono detti *interni* quando è possibile contenere tutta la sequenza in memoria e l'accesso ad ogni valore della sequenza è diretto. Altrimenti, un algoritmo di ordinamento viene detto *esterno* quando è utilizzato per sequenze in cui l'accesso agli elementi avviene in modo sequenziale, per esempio da disco. Inoltre, si dice che un algoritmo è un algoritmo *in-place* quando

non necessita di effettuare copie della sequenza da ordinare, risparmiando così memoria rispetto ad un algoritmo *non in-place*.

8.2.1 Selection Sort

L'algoritmo `SelectionSort` o algoritmo per selezione, è il primo algoritmo di ordinamento che vediamo. Nonostante esso non sia tra i più efficienti ha il vantaggio di essere decisamente semplice, cosa che lo rende pratico quando abbiamo pochi elementi da ordinare e vogliamo poterlo implementare velocemente.

Data una sequenza $S = \{s_1, \ldots, s_n\}$ da ordinare, l'idea di base dell'algoritmo di ordinamento per selezione è quella di considerare, durante l'esecuzione del passo k, l'intera sequenza come suddivisa in due parti: una sotto-sequenza correttamente ordinata, che occupa le prime $k-1$ posizioni della sequenza, e una sotto-sequenza ancora da ordinare, che costituisce la parte restante della sequenza. Ad ogni passo $k \in \{1, \ldots, n-1\}$, l'algoritmo scambia il primo elemento s_k nella sotto-sequenza non ordinata $\{s_k, \ldots, s_n\}$ con il minimo valore trovato nella stessa sotto-sequenza. Ogni iterazione di questo algoritmo fa crescere di una unità la dimensione della sotto-sequenza ordinata.

Definiamo A l'insieme di posizioni della sotto-sequenza ordinata e B l'insieme di posizioni della sotto-sequenza non ordinata nella sequenza S e proviamo a vedere passo per passo come si comporta l'algoritmo.

- Iterazione 1: $A = \Phi$ (insieme vuoto), $B = \{1, \ldots, n\}$. Cerchiamo il minimo in $S[1..n]$ e lo scambiamo con il valore in $S[1]$. Otteniamo $A = \{1\}, B = \{2, \ldots, n\}$ e il valore minimo degli elementi della sequenza S si trova adesso nella sua posizione corretta, ovvero la prima.

- Iterazione 2: $A = \{1\}, B = \{2, \cdots, n\}$. Cerchiamo il minimo in $S[2..n]$ e lo scambiamo con il valore in $S[2]$. Otteniamo $A = \{1, 2\}, B = \{3, \ldots, n\}$ e il secondo minimo di S si trova adesso nella sua posizione corretta, ovvero la seconda.

- Iterazione k: $A = \{1, \ldots, k-1\}, B = \{k, \ldots, n\}$. Cerchiamo il minimo in $S[k, \ldots, n]$ e lo scambiamo con il valore in $S[k]$. Otteniamo $A = \{1, \ldots, k\}, B = \{k+1, \ldots, n\}$ e il k-esimo valore più piccolo di S si trova adesso nella sua posizione corretta, ovvero la k-esima.

- Iterazione $n-1$: $A = \{1, \ldots, n-2\}, B = \{n-1, n\}$. Cerchiamo il minimo tra $S[n-1]$ e $S[n]$ e lo scambiamo con $S[n-1]$ ottenendo la sequenza ordinata finale, ovvero $A = \{1, \ldots, n\}, B = \Phi$ (insieme vuoto).

Segue una implementazione in Python dell'algoritmo `SelectionSort`.

Script 8.1: selection_sort.py

```
1 def selection_sort(seq):
2     for i in range(0, len(seq)-1):
3         imin = i
4         for j in range(i+1, len(seq)):
5             if seq[j] < seq[imin]:
6                 imin = j
7         seq[i], seq[imin] = seq[imin], seq[i]
```

L'operazione dominante per il `SelectionSort` è il confronto tra coppie di elementi. Per $n-1$ volte bisogna determinare il minimo tra gli elementi non ordinati e se necessario effettuare uno scambio. Durante la prima iterazione bisogna effettuare $n-1$ confronti per determinare il minimo tra gli n elementi. Nella seconda iterazione bisogna effettuare $n-2$ confronti per determinare il minimo tra gli $n-1$ elementi rimasti nella sequenza non ordinata. Infine, nella iterazione $n-1$ bisogna effettuare un solo confronto per determinare il minimo tra 2 elementi. Riassumendo, il numero di confronti è:

$$\sum_{i=0}^{n-1}(n-i) = \frac{n*(n+1)}{2} = O(n^2) \text{ operazioni!}$$

8.2.2 Insertion Sort

L' `InsertionSort`, in italiano ordinamento per inserimento, è un altro algoritmo relativamente semplice per ordinare una sequenza. L'idea di base non è molto diversa dal modo con il quale un essere umano tipicamente ordina una mano di carte da gioco. Pur essendo comunque poco efficiente rispetto ad algoritmi più avanzati, ha alcuni vantaggi: è semplice da implementare ed è efficiente per sequenze di partenza che sono quasi ordinate.

Ad ogni passo k dell'algoritmo, si assume che la sequenza $S = \{s_1, \ldots, s_n\}$ da ordinare sia partizionata in una sotto-sequenza individualmente già ordinata $S[A]$, con $A = \{1, \ldots, k\}$ e la rimanente $S[B]$, con $B = \{k+1, \ldots, n\}$ non ordinata. Nota che all'inizio della procedura di ordinamento la sotto-sequenza $S[1]$ è banalmente ordinata poiché composta da un solo elemento, e quindi abbiamo $A = \{1\}, B = \{2, \ldots, n\}$. In ogni iterazione dell'algoritmo, viene rimosso il primo elemento dalla sotto-sequenza non ordinata $S[B]$ e inserito nella sua posizione corretta della sotto-sequenza ordinata $S[A]$. Alla k-esima iterazione, la sequenza ordinata conterrà k elementi e quindi alla iterazione n avremo la sequenza correttamente ordinata in S.

Allo scopo di inserire nella posizione corretta in $S[A]$ l'elemento in $S[k+1]$ si scorre all'indietro la sequenza, mediante un indice j, a partire da $S[k]$ fino ad arrivare ad un valore $S[j]$ che sia minore o uguale a $S[k+1]$. In ognuna di queste iterazioni, si scambiano i valori di $S[j]$ e $S[j-1]$. Questi scambi hanno lo scopo di "spostare" di una posizione a destra l'intera sotto-sequenza dei valori in $S[A]$ maggiori del valore da inserire, in modo tale da preparare la posizione

corretta in cui inserire l'elemento $S[k+1]$. Segue una implementazione in Python dell'algoritmo `InsertionSort`.

Script 8.2: insertion_sort.py

```
1 def insertion_sort(seq):
2     for i in range(1, len(seq)):
3         seq_i = seq[i]
4         j = i
5         while j > 0 and seq[j - 1] > seq_i:
6             seq[j] = seq[j - 1]
7             j -= 1
8         seq[j] = seq_i
```

Passiamo adesso a fare un'analisi della complessità dell'algoritmo iniziando dal caso più semplice, il caso ottimo. Immaginiamo inizialmente di dover effettuare l'ordinamento in una sequenza già ordinata. In questo caso è facile notare che il controllo nel `while` in riga 5 risulterà non verificato in ognuna delle iterazioni della variabile `i`. Questo ci permette di dire che il numero di operazioni dell'algoritmo, in questo caso particolare, è dell'ordine di n.

Prendiamo adesso il caso pessimo, ovvero il caso in cui la sequenza da ordinare sia "capovolta" (detto in altri termini, abbiamo che la sequenza invertita $S[n, \ldots, 1]$ risulta ordinata). In questo caso, per ogni iterazione della variabile `i`, la sotto sequenza $A = S[1, \ldots, i]$ verrà scansionata interamente all'indietro (effettuando uno scambio ogni volta) poiché il valore da inserire dovrà finire in prima posizione. E quindi, per ogni `i` avremo un numero `i` di scambi. Riassumendo, il numero di operazioni richieste sarà:

$$\sum_{i=1}^{n-1} i = \frac{n * (n-1)}{2} = O(n^2) \text{ operazioni!}$$

Un'analisi formale del caso medio richiederebbe più spazio di quello che possiamo dedicare in questo testo. Non è difficile convincersi però che in media la complessità dell'algoritmo rimane quella del caso peggiore. Il ragionamento che si può fare intuitivamente è che l'iterazione interna (della variabile `j`) viene ripetuta un numero di volte `i/2` in media (invece di `i` volte come nel caso pessimo). Ne segue che il numero di operazioni richieste varierà solo in termini di una costante moltiplicativa ($1/2$ appunto) non facendo cambiare così la complessità asintotica dell'algoritmo rispetto al caso pessimo.

8.2.3 Bubble Sort

Il nome `BubbleSort` (ordinamento a bolle) deriva dalla analogia tra i successivi spostamenti che compiono gli elementi dalla posizione di partenza a quella ordinata simile appunto alla risalita delle bolle di aria in un liquido.

Sia $S = \{s_1, \ldots, s_n\}$ la sequenza da ordinare, nella versione più semplice l'algoritmo `BubbleSort` scorre n volte l'intera sequenza. Ad ogni iterazione si com-

parano le coppie di elementi adiacenti eventualmente scambiandoli di posizione se essi non sono nell'ordine corretto.

Volendo far "risalire le bolle dal fondo", in ognuna delle passate, e per ogni posizione $k = 1, .., n-1$ della sequenza, viene comparato il valore in $S[k]$ con quello in $S[k+1]$. Se i due valori sono tali che $S[k] > S[k+1]$ (ordine non corretto) i loro valori vengono scambiati.

Facciamo subito un esempio partendo dalla sequenza

```
S = [4, 5, 1, 2, 3]
```

Monitoriamo quello che succede durante una singola passata dell'algoritmo sulla sequenza.

```
S = [4, 5, 1, 2, 3] # 1^ iterazione (4<5, ok)
S = [4, 1, 5, 2, 3] # 2^ iterazione (5>1, scambiati)
S = [4, 1, 2, 5, 3] # 3^ iterazione (5>2, scambiati)
S = [4, 1, 2, 3, 5] # 4^ iterazione (5>3, scambiati)
```

Possiamo notare che, dopo una passata dell'algoritmo sulla sequenza, il valore più alto viene posizionato alla fine della sequenza. Dopo una ulteriore passata sulla sequenza otteniamo:

```
S = [1, 4, 2, 3, 5] # 1^ iterazione (4>1, scambiati)
S = [1, 2, 4, 3, 5] # 2^ iterazione (4>2, scambiati)
S = [1, 2, 3, 4, 5] # 3^ iterazione (4>3, scambiati)
S = [1, 2, 3, 4, 5] # 4^ iterazione (4<5, ok)
```

La seconda passata ha fatto sì che il secondo massimo sia ora posizionato nella posizione corretta. Inoltre, l'ultimo confronto fatto è stato inutile poichè, per ipotesi, l'ultima posizione era già occupata dal massimo della sequenza e non richiedeva di essere modificata. In generale, dopo la k-esima passata, possiamo evitare di confrontare gli ultimi k valori della sequenza che sono già nella posizione corretta.

Riprendendo l'esempio sopra ci accorgiamo che le coppie adiacenti adesso sono tutte correttamente ordinate. Per cui ripetendo l'algoritmo sulla sequenza, non avremmo nessuno scambio ulteriore. In generale, sono necessarie al più $n-1$ scansioni dell'intera sequenza per ottenere una sequenza ordinata.

Riassumendo, abbiamo individuato due possibili ottimizzazioni che possiamo applicare all'algoritmo standard. In particolare:

Ottimizzazione n. 1: per la passata k-esima si itera la sequenza dall'inizio fino a confrontare il valore in posizione $(n-k)$ con quello in posizione $(n-k+1)$ tralasciando quelli con posizione maggiore di $n-k+1$ poichè essi saranno già nella posizione corretta.

Ottimizzazione n. 2: se per una intera iterazione non sono stati effettuati scambi, allora significa che la sequenza è già ordinata e possiamo terminare l'algoritmo.

Di seguito presentiamo l'implementazione in Python dell'algoritmo `BubbleSort` dove sono implementate entrambe le ottimizzazioni citate sopra.

Script 8.3: bubble_sort.py

```
def bubble_sort(seq):
    i,swap_test = 0,True
    while swap_test and i < len(seq)-1:
        swap_test = False
        for j in range(0, len(seq)-i-1):
            if seq[j] > seq[j+1]:
                seq[j], seq[j+1] = seq[j+1], seq[j]
                swap_test = True
        i+=1
```

Per accorgersi se sono stati effettuati scambi per una certa iterazione (ottimizzazione 2) abbiamo utilizzato un flag `swap_test`, che viene re-inizializzato a `False` all'inizio di ogni iterazione su `i`, ossia si ipotizza il vettore ordinato a meno di trovare una coppia da scambiare che dimostri che non lo è. In quest'ultimo caso il valore di `swap_test` sarà `True` all'uscita del `for`. Quando viceversa il valore del flag `swap_test` all'uscita dal `for` è `False` significa che la sequenza è correttamente ordinata e la guardia sul `while` garantisce la terminazione anticipata dell'algoritmo.

L'algoritmo `BubbleSort` esegue un numero di operazioni variabili a seconda dell'ordinamento già presente nella sequenza. Il caso ottimo si ha quando la sequenza è già ordinata in partenza, ed in questo caso è sufficiente effettuare solo un ciclo per accorgersi che tutti gli elementi sono ordinati, quindi sono sufficienti $n-1$ confronti. Nota che questo è dovuto all'ottimizzazione 2. Il caso peggiore si ha quando gli elementi nella sequenza sono ordinati in senso decrescente. Nel caso peggiore il `BubbleSort` effettua i seguenti confronti:

- prima iterazione: vengono eseguiti $n-1$ confronti e altrettanti scambi;
- seconda iterazione: vengono eseguiti $n-2$ confronti e altrettanti scambi;
- $(n-1)$-esima iterazione: viene eseguito un confronto e uno scambio.

Il numero di confronti/scambi del `BubbleSort` può quindi essere così calcolato:

$$\sum_i (n-i) = n\sum_i 1 - \sum_i i = n*(n-1) - n*(n-1)/2 = n*(n-1)/2 = O(n^2)$$

8.2.4 Merge Sort

DIVIDE ET IMPERA, dicevano i latini, volendo significare che, una volta conquistato un nuovo territorio, è meglio tenere diviso il popolo, se ostile, per evitare rivolte e sovversioni, ma anche con il significato più generale di partizionare il territorio per amministrarlo meglio.

Proprio per questa seconda interpretazione lo stesso motto latino è stato acquisito dall'informatica (molto più tardi) per descrivere una tecnica generale per la risoluzione di problemi complessi. In questo tipo di tecniche, partendo da un

problema complesso, si genera una sequenza di istanze (o sotto-problemi) via via più semplici, fino all'istanza che non è più suddivisibile e che ha soluzione banale.

L'algoritmo di ordinamento `MergeSort` è formulato ricorsivamente secondo la tecnica del divide et impera. Esso divide ripetutamente la sequenza da ordinare via via a metà, fino a generare segmenti contenenti un solo elemento, quindi banalmente già ordinati. Una volta che siano ordinate due metà di una sequenza si effettua la cosiddetta *fusione* fra le sottosequenze in ordine inverso rispetto alla fase di divisione.

Diamo quindi una formulazione della soluzione ricorsiva:

- **Caso base:** se la sequenza ha un solo elemento, allora la sequenza è già ordinata.
- **Caso ricorsivo:** consideriamo le due metà della sequenza ("divide") e ricorsivamente ordiniamo le due sotto-sequenze ("impera"). Le due sotto-sequenze individualmente ordinate vengono poi fuse per formare la sequenza ordinata (fusione).

Questo algoritmo ricorsivo assume l'esistenza di una procedura efficiente per aggregare due sotto-sequenze individualmente ordinate in una sequenza ordinata. Per fare questo, dato che le due sotto-sequenze sono ordinate, basterà scorrerle entrambe dall'inizio estraendo di volta in volta l'elemento minore delle due per collocarlo alla fine della sequenza aggregata. Questo algoritmo iterativo prende il nome di `Merge` (fusione, appunto).

Di seguito presentiamo l'implementazione in Python dell'algoritmo `MergeSort`. Vengono utilizzate tre funzioni annidate. La funzione più esterna (`merge_sort()`) richiama la funzione ricorsiva vera e propria `merge_sort_r()` con i parametri corretti. Quest'ultima funzione, a sua volta, usa la funzione annidata `merge()` che effettua la fusione di sotto-sequenze ordinate.

Script 8.4: merge_sort.py

```
1  def merge_sort(seq):
2
3      # merge sort ricorsivo (usato da merge_sort)
4      def merge_sort_r(seq, primo, ultimo):
5
6          # fusione (usato da merge_sort_r)
7          def merge(seq, primo, ultimo, medio):
8              merge_result = []
9              i = primo
10             j = medio + 1
11             while i <= medio and j <= ultimo:
12                 if seq[i] <= seq[j]:
13                     merge_result.append(seq[i])
14                     i += 1
15                 else:
```

```
16                merge_result.append(seq[j])
17                j += 1
18            while i <= medio:
19               merge_result.append(seq[i])
20               i +=1
21            while j <= ultimo:
22               merge_result.append(seq[j])
23               j += 1
24            for k in range(0, ultimo - primo + 1):
25               seq[primo+k] = merge_result[k]
26
27         ## merge_sort_r
28         if primo < ultimo:
29            medio = (primo + ultimo)/2
30            merge_sort_r(seq, primo, medio)
31            merge_sort_r(seq, medio+1, ultimo)
32            merge(seq, primo, ultimo, medio)
33
34      ## merge_sort
35      merge_sort_r(seq, 0, len(seq)-1)
```

Si può dimostrare che il `MergeSort` richiede un tempo di esecuzione di ordine $O(n \cdot log_2(n))$, se n è la lunghezza della sequenza da ordinare. Il `MergeSort`, per sequenze di grandi dimensioni, è il più efficiente algoritmo di ordinamento basato sui confronti. È possibile dimostrare che, nel caso pessimo, non possono esistere algoritmi di ordinamento di sequenze basati sui confronti che hanno una complessità computazionale inferiore a $n \cdot log_2(n)$. Il principale svantaggio di `MergeSort` è che, a differenza degli altri algoritmi che abbiamo visto precedentemente, è un algoritmo non in-place.

8.3 Algoritmi di ricerca per chiave

Il problema della ricerca può essere descritto nel modo seguente. Dato un insieme di elementi, vogliamo verificare se un certo elemento fa parte dell'insieme oppure no. I singoli elementi dell'insieme sono caratterizzati da una *chiave* e dal dato vero e proprio che vogliamo recuparare con quella chiave. Nella descrizione degli algoritmi di ricerca, i dati associati possono essere ignorati in quanto non utilizzati ai fini dela ricerca. La chiave è quell'insieme di valori che identifica un elemento dell'insieme.

In caso l'elemento cercato sia effettivamente nell'insieme, può essere richiesto di restituire il dato associato oppure la posizione della chiave nel caso di ricerca su sequenze. La chiave può essere di qualsiasi tipo e può anche essere formata da una tupla di valori. La chiave può essere *univoca* in tutto l'insieme di elementi, oppure *multipla* qualora sia consentito di condividerla tra più elementi distinti. In questo secondo caso è fondamentale specificare il corretto comportamento di

un algoritmo di ricerca. Bisogna infatti decidere se dovrà essere restituito il primo elemento dotato di una certa chiave, l'ultimo, uno qualsiasi o anche tutti.

Qui supponiamo per semplicità di avere una sequenza di valori (chiavi) su cui vogliamo ricercare un certo valore. Se il valore viene trovato all'interno della sequenza, la ricerca ritorna l'indice dell'elemento cercato, viceversa la ricerca ritorna `None` per indicare che un tale elemento non è stato trovato nella sequenza.

8.3.1 Ricerca lineare

La ricerca lineare (Script 8.5) effettua una scansione della sequenza dall'inizio. Non appena trova un elemento della sequenza avente valore uguale a quello cercato ritorna l'indice dell'elemento. Una volta raggiunta la fine della sequenza, se l'elemento non è stato trovato ritorna `None`. Sia n il numero di elementi della sequenza, la complessità computazionale in tempo nel caso peggiore di questo algoritmo è $O(n)$, che corrisponde al caso in cui l'elemento non esiste nella sequenza. Anche nel caso medio l'ordine non cambia ($O(n)$) in quanto (in media) ci possiamo aspettare di trovare l'elemento cercato più o meno nel mezzo della sequenza. Nel caso ottimo, invece, quando l'elemento da cercare è il primo elemento della sequenza, la complessità è ovviamente costante.

Script 8.5: ricerca_lin.py

```
1 def ricerca_lineare(seq,x):
2     '''Funzione di ricerca lineare.'''
3     i,n = 0,len(seq)
4     while i<n:
5         if seq[i]==x:
6             return i
7         else:
8             i += 1
9     return None
```

8.3.2 Ricerca binaria

La ricerca binaria (Script 8.6) è applicabile solo nel caso in cui la sequenza `seq` su cui vogliamo effettuare la ricerca è ordinata. In questo caso possiamo procedere nel seguente modo. Ad ogni iterazione controlliamo se l'elemento cercato è quello che sta in mezzo nella sequenza (chiamiamo `im` il suo indice). Se lo è ritorniamo l'indice e abbiamo finito. Altrimenti, il valore da cercare `x` viene confrontato con quello corrispondente all'indice `im` della sequenza. Se `x < seq[im]` significa che l'elemento cercato, se esiste, si troverà nella prima metà della sequenza e, viceversa, se `x > seq[im]` allora l'elemento cercato, se esiste, si troverà nella seconda metà della sequenza. In questo modo, ad ogni iterazione dividiamo la sequenza su cui cercare esattamente a metà e una delle metà non viene poi più

considerata nelle iterazioni successive. Il numero di confronti totali sarà quindi proporzionale a $log_2(n)$ dove n è la lunghezza della sequenza iniziale.

Script 8.6: ricerca_bin.py

```
def ricerca_binaria(seq,x):
   '''Funzione di ricerca binaria.'''
   def rb(beg,end):
      if beg>end: return None
      im = (beg+end)/2
      if seq[im]==x: return im
      if seq[im]<x: return rb(im+1,end)
      if seq[im]>x: return rb(beg,im-1)

   return rb(0,len(seq)-1)
```

8.3.3 Ricerca su liste concatenate

Presentiamo adesso tre funzioni di ricerca su liste concatenate che sono implementate nello Script 8.7.

La funzione `trova()` cerca nella lista il primo nodo contenente un certo valore e lo ritorna se esiste, altrimenti ritorna `None`. La sua definizione ricorsiva è data seguendo lo schema:

$$T([], v) = \texttt{None},\ T([v|L]) = v,\ T([x \neq v|L]) = T(L, v)$$

La funzione `check_init_pattern()` verifica la presenza o meno di una determinata sequenza di valori (pattern) in nodi consecutivi a partire dal primo della lista. Se un tale pattern esiste ritorna `True` altrimenti ritorna `False`. L'implementazione proposta ha ben tre casi base. Il primo si ha quando è richiesta la verifica di un pattern vuoto. In questo caso banalmente la risposta è `True`. Escludendo che il pattern sia vuoto, il secondo caso base si ha quando la lista è vuota. In questo caso è chiaro che il pattern non esiste e quindi ritorniamo `False`. Il terzo e ultimo caso base si ottiene quando il primo nodo della lista non ha il valore cercato. Poichè sappiamo che il pattern da cercare non è vuoto possiamo tranquillamente ritornare `False` in questo caso. Infine, il caso ricorsivo nel quale, dato che il primo nodo della lista ha lo stesso valore del primo valore del pattern, allora la lista avrà l'intero pattern all'inizio se e solo se è vera ricorsivamente la `check_init_pattern()` applicata alla lista concatenata `lista.next` e il pattern ridotto `pattern[1:]`.

La funzione `check_pattern()` verifica la presenza o meno di una determinata sequenza di valori (pattern) in nodi consecutivi a partire da qualunque nodo della lista concatenata. In realtà l'implementazione è molto semplice avendo già definito la `check_init_pattern()`. L'idea è quella di controllare la presenza del pattern cercato iterativamente sulle sotto-liste ottenute a partire dal primo nodo, dal secondo, dal terzo e così via.

Script 8.7: liste_ricerca.py

```
### funzioni di ricerca per liste concatenate

from liste import *

def trova(lista, val):
   if is_vuota(lista):
      return None
   if (lista.val==val):
      return lista
   return trova_indice(lista.next,val)

def check_init_pattern(lista,pattern):
   if not pattern:
      return True
   if is_vuota(lista):
      return False
   if lista.val!=pattern[0]:
      return False
   return check_init_pattern(lista.next,pattern[1:])

def check_pattern(lista,pattern):
   while lista:
      if check_init_pattern(lista,pattern):
         return True
      lista = lista.next
   return False
```

8.3.4 Ricerca su alberi binari

Passiamo adesso a definire un insieme di funzioni utili per la ricerca di valori in un albero binario. In particolare, vedremo

- ricerca di un nodo corrispondente ad un cammino dalla radice;
- ricerca mediante una visita;
- ricerca e BST (alberi binari di ricerca).

Lo Script 8.8 presenta tre implementazioni di algoritmi standard per questi problemi. Commenteremo di seguito ognuna di queste soluzioni.

Script 8.8: ricerca_alberi.py

```
def trova_cammino(albero, C):
    if not C:
        return albero
    if not albero:
        return None
    if C[0]==0:
        return trova_cammino(albero.sx, C[1:])
    else:
        return trova_cammino(albero.dx, C[1:])

def trova_pre(albero, v):
    if not albero:
        return None
    if albero.val==v:
        return albero
    a = trova_pre(albero.sx,v)
    if a:
        return a
    return trova_pre(albero.dx,v)

def bst_search(albero, v):
    if not albero:
        return None
    if albero.val==v:
        return albero
    if albero.val<v:
        return bst_search(albero.sx,v)
    return bst_search(albero.dx,v)
```

Ricerca per cammino

La funzione `trova_cammino()` cerca in un albero binario il nodo corrispondente ad un cammino dato. L'implementazione proposta presenta due casi base. Il primo caso base si presenta quando il cammino passato per argomento è una lista vuota. Ciò significa che il nodo cercato è proprio il nodo radice. L'altro caso base, considera la situazione nella quale l'albero è vuoto e il cammino cercato non è vuoto. In questo caso dobbiamo ritornare `None` poichè è chiaro che il nodo corrispondente a quel cammino non esiste. Infine, abbiamo il caso ricorsivo nel quale andiamo a verificare il primo valore del cammino. Se questo è uguale a 0 significa che il nodo va cercato nel sotto-albero a sinistra. Viceversa, se è uguale a 1 significa che il nodo va cercato nel sotto-albero a destra. In entrambi i casi si tratterà di effettuare una chiamata ricorsiva alla funzione passando come argomenti il sotto-albero corretto e la parte rimanente del cammino.

Ricerca lineare con visita prefissa

La funzione `trova_pre()` effettua una semplice ricerca di un valore su un albero binario effettuando una visita pre-ordine. Per ogni nodo visitato si effettua il controllo verificando se quel nodo contenga o meno il valore cercato. In caso positivo ritorniamo il nodo all'ambiente chiamante, altrimenti procediamo con la visita.

Ricerca su alberi binari di ricerca

Gli *alberi binari di ricerca* (BST, detti anche alberi ordinati) sono particolari alberi binari con la seguente proprietà: *il valore di ogni nodo è maggiore del valore di tutti i nodi nel sotto-albero sinistro e minore del valore di tutti i nodi nel sotto-albero destro*.

Tale proprietà dell'albero binario rende la ricerca di un valore in un albero binario più efficiente rispetto ad una semplice ricerca con visita dell'albero. In particolare, per ogni nodo dell'albero, possiamo verificare se tale nodo contiene il valore che stiamo cercando. In questo caso, lo abbiamo trovato e lo ritorniamo. In caso contrario, il valore che stiamo cercando potrebbe essere maggiore o minore di quello associato al nodo. Se è minore allora, per la proprietà di un BST, sappiamo che il nodo cercato, se esiste, si trova nel sotto-albero sinistro. Viceversa, se è maggiore del valore del nodo che stiamo considerando, allora sappiamo che si trova nel sotto-albero destro. In entrambi i casi possiamo focalizzare la ricerca su di un sotto-albero e non considerare alcun nodo nell'altro sotto-albero. Se l'albero è bilanciato, ovvero avente altezza proporzionale al logaritmo del numero di nodi $h \approx log(n)$, allora la complessità della ricerca si riduce da n (ricerca lineare su albero binario) a $log_2(n)$, un notevole miglioramento! Una possibile implementazione di questo metodo è data nella funzione `bst_search()`.

8.4 Algoritmo backtracking

Il *backtracking* è una tecnica per esplorare tutte le possibili configurazioni di uno spazio di ricerca in modo sistematico. Tali configurazioni possono corrispondere, per esempio, ai possibili arrangiamenti di un insieme di oggetti in una sequenza (*permutazioni*) o tutte le possibili parti di tale insieme (*sotto-insiemi*). In questo tipo di problemi, ogni configurazione ammissibile deve venir generata *esattamente* una volta, evitando di perdere o replicare alcune delle configurazioni.

Rappresentiamo ogni configurazione come un vettore $\mathbf{a} = \{a_1, \ldots, a_n\}$ di lunghezza n, dove $a_i \in A_i$ e A_i si suppone un insieme ordinato di elementi *candidati* per la posizione i-esima della configurazione. Possiamo considerare inoltre che l'insieme dei candidati in A_i dipenda solo dai valori assegnati precedentemente nella configurazione parziale $\mathbf{a}_{i-1} = \{a_1, \ldots, a_{i-1}\}$, ovvero $A_i = A_i(\mathbf{a}_{i-1})$.

L'idea generale dell'algoritmo di backtracking è quella di costruire soluzioni parziali con elementi fissati fino all'indice k ed estenderla di un elemento alla volta cercando in ordine tra i candidati in $A_{k+1}(\mathbf{a}_k)$.

L'algoritmo ad ogni passo tiene traccia delle scelte effettuate in precedenza in modo da poter tornare indietro e annullare alcune di queste scelte per perseguire soluzioni alternative.

La tecnica del backtracking corrisponde, in pratica, ad una visita in profondità (pre-ordine) di un albero (detto *albero delle soluzioni*) dove ciascun nodo interno x di livello k corrisponde ad una soluzione parziale di lunghezza k e c'è un nodo y a livello $k+1$, figlio di x, se e solo se la soluzione parziale y è stata costruita estendendo la soluzione parziale in x con un candidato in A_{k+1}.

Nello Script 8.9 viene proposto uno schema generale dell'algoritmo di backtracking. L'algoritmo `BACKTRACK` ritorna la lista di tutte le configurazioni ammissibili per un determinato problema. Essa prende tre argomenti. I primi due rappresentano una soluzione parziale $\mathbf{a_k}$ (`s` è la lista contenente i k candidati della soluzione parziale). Il terzo argomento `n` è la dimensione del problema. L'algoritmo è ricorsivo. Il caso base si ottiene quando `k==n` ovvero quando la soluzione parziale è una soluzione del problema. Quando questo è il caso viene ritornata la lista con un solo elemento corrispondente alla copia della soluzione. Viceversa, calcoliamo l'insieme A_{k+1} a partire da $\mathbf{a}_k$ e ricorsivamente generiamo la lista di soluzioni ottenibili al variare del candidato prescelto in A_{k+1}.

Script 8.9: Schema Backtracking

```
def BACKTRACK(s,k,n):
   if k==n: # s soluzione completa:
          return [s[:]] # ne ritorna una copia
   sols = []
   # Calcolo di A(k+1) in funzione di s
   for i in A(k+1):
      s.append(i)
      sols += BACKTRACK(s,k+1,n)
      s.pop()
   return sols

### sols = BACKTRACK([],0,n)
```

Di seguito vengono presentati una serie di esempi di applicazione dello schema precedente che dimostrano quanto esso sia generale e adatto per un largo spettro di problemi pratici.

8.4.1 Sotto-insiemi su n elementi

Volendo calcolare tutti i possibili sotto-insiemi di un insieme di n elementi possiamo considerare il fatto che i diversi sotto-insiemi possono essere visti come configurazioni con elementi $A_k = A = \{T, F\}$ intendendo che $a_k = T$ rappresenta il fatto che l'elemento k-esimo viene preso nel sotto-insieme e $a_k = F$ viceversa. Applicando lo schema precedente otteniamo la seguente implementazione.

```
def get_sottoinsiemi(s,k,n):
    if k==n:
        return [s[:]]
    sols = []
    A = (True,False)
    for b in A:
        s.append(b)
        sols += get_sottoinsiemi(s,k+1,n)
        s.pop()
    return sols
```

8.4.2 Permutazioni di n elementi

Per ottenere le permutazioni di n elementi possiamo considerare l'insieme $A_1 = \{1, \ldots, n\}$ come l'insieme di candidati per la prima posizione. Quindi, data una soluzione parziale $\{a_1, \ldots, a_k\}$, allora poniamo $A_{k+1} = A_1 - \cup_{i=1}^{k}\{a_i\}$, ovvero l'insieme di elementi da 1 a n escludendo quelli che sono già presenti nella soluzione parziale. Applicando lo schema generale a questo caso otteniamo la seguente implementazione.

```
def get_permutazioni(s,k,n):
    if k==n:
        return [s[:]]
    sols = []
    A = [i for i in range(n) if i not in s]
    for a in A:
        s.append(a)
        sols += get_permutazioni(s,k+1,n)
        s.pop()
    return sols
```

8.4.3 Disposizioni di c elementi su n

Vediamo adesso come generare tutte le disposizioni di c elementi su un insieme di n elementi con $0 < c < n$. Per fare questo consideriamo l'insieme $A_1 = \{0, 1\}$ come l'insieme di candidati per la prima posizione. Data una soluzione parziale $\mathbf{a}_k = \{a_1, \ldots, a_k\}$, definiamo

$$S_k = \sum_{i=1}^{k} a_i.$$

S_k è il numero di elementi già disposti nella configurazione parziale $\mathbf{a}_k$. Allora, sia $\mathbf{a}_k$ tale che $S_k \leq c$, possiamo definire i vincoli per la costruzione dell'insieme A_{k+1} come segue:

- $1 \in A_{k+1} \iff S_k < c$, ovvero possiamo disporre l'elemento $(k+1)$-esimo se e solo se non abbiamo ancora disposto tutti gli elementi.

- $0 \in A_{k+1} \iff c - S_k < n - k$, ovvero possiamo non disporre l'elemento $(k+1)$-esimo se e solo se il numero di elementi ancora da disporre $(c - S_k)$ è strettamente minore del numero di scelte ancora da effettuare $(n - k)$.

Applicando lo schema generale a questo caso e le regole per la generazione dell'insieme di candidati A_{k+1} come descritto sopra, otteniamo la seguente implementazione.

```
def get_disposizioni(s,k,n,c):
   if k==n:
      return [s[:]]
   sols = []
   sk = sum(s) # sk: numero di elementi gia' disposti in s
   A = ([1] if sk<c else [])+([0] if (c-sk < n-k) else [])
   for a in A:
      s.append(a)
      sols += get_disposizioni(s,k+1,n,c)
      s.pop()
   return sols
```

Esercizi del capitolo

Esercizio 8.1

Si dia una funzione `spirale()` *che, dato* $n > 0$*, crea e ritorna una matrice (come lista di liste) con elementi* $1, .., n^2$ *ordinati in una spirale. Per esempio, dato* n*=4, dobbiamo generare la matrice*

```
 1   2  3  4
12 13 14  5
11 16 15  6
10  9  8  7
```

Esercizio 8.2

Siano date due matrici come liste di liste, implementare la funzione che ne calcoli il prodotto matriciale usando le note formule di algebra lineare:

```
def mul_mat(m1,m2)
```

Esercizio 8.3

Dare una funzione `Riordina(L,v)`*, che data una lista* `L` *e un valore* `v`*, riordini gli elementi di* `L` *in modo tale da avere tutti gli elementi minori di* `v` *che precedono quelli maggiori di* `v`*.*

Esercizio 8.4

Dare uno script che, presa una stringa in input, determina e stampa:

- il massimo numero di caratteri uguali consecutivi,
- il carattere coinvolto,
- e la sua posizione nella stringa immessa

Esempi

- per `'hgskjdhgkjgfhgdgggwer'` ⇒ stampa `3 'g' 15`
- per `'abvsdvdvvvvvdseeeevv'` ⇒ stampa `5 'v' 7`

Esercizio 8.5 *Si dia uno script* efficiente *che richieda numeri interi da input e verifichi che l'ennesimo numero inserito corrisponda all'ennesimo numero di Fibonacci ($f_n = f_{n-1} + f_{n-2}, f_0 = 0, f_1 = 1$). All'inserimento del numero 0 il programma dovrà terminare. La complessità dell'algoritmo di verifica non deve essere più che costante!*

9

Progetti

9.1 Il gioco del campo minato

In questa sezione realizzeremo un programma per il gioco del campo minato. Il programma

- deve definire un'interfaccia con il giocatore. Al giocatore si deve permettere l'inserimento delle mosse;
- dovrà occuparsi di controllare la validità delle mosse e decretare l'eventuale esito della partita;
- deve essere parametrico rispetto alle dimensioni del campo di gioco e al numero di bombe presenti.

9.1.1 Regole del gioco

Il campo di gioco consiste in un campo rettangolare (o quadrato) di celle. Ogni cella viene ripulita, o scoperta, cliccando su di essa. Se una cella contenente una bomba viene cliccata il gioco termina e il giocatore perde la partita. Se la cella cliccata non contiene una bomba, possono accadere due eventi: se appare un numero, esso indica la quantità di celle adiacenti (incluse quelle in diagonale) che contengono bombe; se non appare nessun numero, il gioco ripulisce automaticamente le celle adiacenti a quella vuota (fino a quando esse non conterranno un numero). Il giocatore vince la partita quando tutte le celle che non contengono bombe saranno scoperte.

9.1.2 Progettazione e implementazione

Il gioco viene progettato come una classe (`CampoMinato`). Un oggetto di tipo `CampoMinato` viene costruito passando le dimensioni (numero di righe `DIMX` e numero di colonne `DIMY`) del campo di gioco e il numero di bombe `nbombe` da distribuire casualmente nel campo.

L'implementazione completa della classe `CampoMinato` è riportata nello Script 9.1. La Tabella 9.1 riassume i due metodi interfaccia della classe, ovvero quei metodi che è necessario conoscere per poter utilizzare la classe.

Iniziamo col definire la struttura dati che ci permette di memorizzare lo stato del campo. Il campo viene descritto da una matrice `celle` (una lista di liste), dove per ogni riga verrà memorizzato con una lista lo stato delle celle in quella riga. Lo stato sarà -1 quando la cella è coperta. Un numero maggiore o uguale a zero indicherà il numero di bombe nel suo intorno. Useremo poi un insieme `bombe` per contenere le coordinate dove si trovano le bombe.

Script 9.1: campo_minato.py

```
import random

class CampoMinato():
   def __init__(self, DIMX = 8, DIMY = 8, nbombe=5):
      self.DIMX=DIMX
      self.DIMY=DIMY
      self.nbombe=nbombe
      self.bombe = set(random.sample(self.tutte_le_celle(),
         nbombe))
      self.celle = [[-1]*self.DIMX for i in range(self.DIMY)]
      # -1 coperta, numero di bombe intorno

   def in_board(self,x,y):
      return 0<=x<self.DIMX and 0<=y<self.DIMY

   def tutte_le_celle(self):
      tutti_x = range(self.DIMX)
      tutti_y = range(self.DIMY)
      return [(x,y) for x in tutti_x for y in tutti_y]

   def cella_coperta(self,x,y):
      return self.celle[x][y]==-1

   def numero_celle_coperte(self):
      return len([(x,y) for (x,y) in self.tutte_le_celle() \
                        if self.cella_coperta(x,y)])

   def vicini(self,x,y):
      return set([(xx,yy) for yy in (y-1,y,y+1) \
                          for xx in (x-1,x,x+1) \
                          if self.in_board(xx,yy) \
                          and (xx,yy)!=(x,y)])

   def print_board(self):
      print " ",
```

```
35        for i in range(self.DIMY):
36           print i,
37        print
38        for x in range(0,self.DIMX):
39           print(x),
40           for y in range(0,self.DIMY):
41              if self.cella_coperta(x,y):
42                  print("#"),
43              elif self.celle[x][y]==0:
44                  print(" "),
45              else:
46                  print("%d"%self.celle[x][y]),
47           print
48
49     def scopri(self,x,y):
50        self.celle[x][y]= len(self.bombe & self.vicini(x,y))
51        if not self.celle[x][y]:
52           for (xx,yy) in self.vicini(x,y):
53              if self.cella_coperta(xx,yy) :
54                 self.scopri(xx,yy)
55
56     def clicca(self,x,y):
57        if (x,y) in self.bombe:
58           print "La cella (%d,%d) e' una BOMBA!!!"%(x,y)
59           return False
60        if not self.cella_coperta(x,y):
61           print "La cella (%d,%d) e' gia' scoperta!"%(x,y)
62           return True
63        self.scopri(x,y)
64        if self.numero_celle_coperte()==self.nbombe:
65           print "\nOK hai vinto!!!"
66           return False
67        return True
68
69     def play(self):
70        self.print_board()
71        ret = True
72        while ret:
73           (x,y) = raw_input("Cella: ").split()
74           (x,y) = (int(x),int(y))
75           ret = self.clicca(x,y)
76           self.print_board()
77
78  ### per giocare su una griglia 8x8 con 5 bombe:
79  ### CampoMinato(8,8,5).play()
```

Tabella 9.1: Metodi interfaccia della classe CampoMinato.

Metodo	Parametri	Output
`__init__()`	DIMX: numero di righe, DIMY: numero colonne, nbombe: numero di bombe nel campo.	-
`play()`	-	gioca una partita

Trattiamo ora alcuni metodi ausiliari della classe. Il metodo `in_board()` prende due coordinate `x` e `y` come argomenti e verifica la validità delle coordinate rispetto ai limiti del campo. Il metodo `tutte_le_celle()` ritorna semplicemente la lista delle coppie di coordinate valide. Il metodo `cella_coperta()` prende due coordinate `x` e `y` come argomenti e verifica se la cella corrispondente è coperta o meno. Il metodo `vicini()` prende in input due coordinate `x` e `y` rappresentanti una determinata cella e restituisce l'insieme di coppie di coordinate che rappresentano celle vicine a tale cella. Infine, il metodo `print_board()` si incarica di stampare a video lo stato della griglia di gioco. Il simbolo `#` denota la cella ancora coperta. Lo spazio denota la cella scoperta senza bombe attorno. Alternativamente sarà visualizzato il numero di bombe presenti nelle celle vicine.

Il metodo principale della classe è il metodo `play()`. Questo metodo visualizza la griglia a schermo utilizzando il metodo `print_board()` e ripete un ciclo dove ad ogni iterazione viene letta la mossa del giocatore e eseguita col metodo `clicca()`.

Il metodo `clicca()` esegue una mossa e ritorna `True` o `False` a seconda che il gioco debba o meno continuare. Questo metodo controlla prima di tutto se il giocatore ha cliccato su una cella che contiene una bomba. In questo caso, visualizza il messaggio opportuno e ritorna `False`. In caso contrario, se la cella cliccata è una cella già scoperta, segnala l'errore con un messaggio e ritorna `True`. Se nessuna delle precedenti condizioni si verifica, si esegue il metodo `scopri()` e si controlla l'eventuale vittoria del giocatore (condizione in cui tutte le celle senza bombe sono scoperte).

Il metodo `scopri()` è un metodo ricorsivo che scopre a cascata tutte le celle vicine che non confinano con bombe. Prima di tutto esso scopre la cella di coordinate `x,y`. Il caso base di questo metodo è il caso in cui nessuno dei vicini della cella sia ancora coperto. Invece, il caso ricorsivo si ha quando almeno una cella vicina deve essere scoperta. In quest'ultimo caso, si richiama ricorsivamente il metodo `scopri()` sulla cella vicina.

9.2 Progetti di analisi numerica

In questa sezione presentiamo due progetti orientati all'analisi numerica: gli algoritmi di ricerca degli zeri di una funzione e l'integrazione numerica.

9.2.1 Algoritmi di ricerca degli zeri di una funzione

I metodi per calcolare in modo approssimato le radici di una equazione, ovvero i valori dell'incognita che soddisfano l' equazione, si articolano in due fasi: nella prima fase si separano le radici, ovvero si determinano gli intervalli della retta reale che contengono una sola radice dell'equazione; nella seconda fase si calcola un valore approssimato della radice dell'equazione.

Supponiamo di aver già separato le radici, ovvero abbiamo trovato un intervallo $[a, b]$ tale che la radice α è compresa nell'intervallo. Mediante algoritmi iterativi vogliamo restringere tale intervallo in modo da ottenere valori più corretti, secondo una approssimazione fissata.

Metodo di Bisezione

Il metodo di ricerca della radice più semplice è il metodo di bisezione o metodo dicotomico. Volendo ricercare una radice di una funzione continua, consideriamo due valori a e b presi in maniera tale che $f(a)$ e $f(b)$ assumano segno opposto; per il teorema degli zeri, l'intervallo conterrà sicuramente una radice x per l' equazione. Definito l'intervallo, con successive iterazioni si procede a progressivi dimezzamenti dello stesso. Alla prima iterazione si sceglie fra i sotto-intervalli $[a, c]$ e $[c, b]$, dove $c = (a + b)/2$ è il punto medio tra a e b, attraverso la valutazione del segno di $f(c)$. La convergenza del procedimento è sempre garantita, ma è abbastanza lenta, in quanto ha andamento *solo* lineare. Questo metodo può essere in parte paragonato all'algoritmo di ricerca binaria per la ricerca di chiavi all'interno di una sequenza.

Script 9.2: zeri_bisection.py

```
def bisezione(f,x1,x2,toll):
    '''Funzione di ricerca zero di una funzione.'''
    m = (x1+x2)/2
    fm = f(m)
    if abs(fm)<toll:
        return m
    f1,f2 = f(x1),f(x2)
    if f1*fm>0:
        return bisezione(f,m,x2,toll)
    else:
        return bisezione(f,x1,m,toll)
```

Metodo delle secanti

Un'alternativa al metodo di bisezione è il metodo delle secanti nel quale sono dati due punti iniziali (a, b) e costruiamo una successione secondo il criterio:

$$x_0 = a,\ x_1 = b,\ x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$$

Lo svantaggio di questo metodo è che la convergenza è locale ovvero dipende dalla scelta dei punti iniziali. Il vantaggio rispetto al metodo di bisezione è una maggiore velocità di convergenza (super-lineare). Segue una implementazione basilare del metodo.

Script 9.3: zeri_secanti.py

```
def zeri_secanti(f,a,b,toll):
    '''Funzione di ricerca zero di una funzione.'''
    while abs(b-a) > toll:
      x = b - (b-a)*f(b)/(f(b)-f(a))
      a,b=b,x
    return x
```

9.2.2 Integrazione numerica

Una ulteriore applicazione della programmazione alla matematica è la integrazione numerica, ovvero l'approssimazione numerica di un integrale definito nella forma:

$$\int_a^b f(x)\,dx$$

Di seguito, proponiamo la famosa formula di quadratura di Eulero per l'approssimazione dell'integrale precedente. In pratica, suddividiamo l'intervallo di integrazione in un insieme di n sotto-intervalli $dx(i) = [a + i * d, a + (i + 1) * d]$ per $i = 1, \ldots, n$, ognuno di dimensione $d = (b - a)/n$ e definiamo l'approssimazione:

$$S = \sum_{i=1}^{n} dx * f(a + i * d)$$

Questo semplice algoritmo può essere implementato come nel programma seguente:

Script 9.4: eulero.py

```
def eulero(f,a,b,n):
    integrale = 0.0
    d = float(b-a)/n
    for k in range(n+1):
       h = a + k*d
       integrale += d*f(h)
    return integrale

### ESEMPIO:
### def f(x):
###     return 2*x + 3*x*x
### eulero(f,0,1,100)
```

A

Il modulo random

Il modulo `random` contiene funzionalità che ci permettono di operare con i numeri pseudo-casuali (o pseudo-random). La funzione più importante definita in questo modulo è la funzione `random()` che ritorna un oggetto float casuale nell'intervallo $[0.0, 1.0)$. Vediamo un esempio del suo utilizzo nella console di Python.

```
>>> from random import *
>>> random()
0.5782931739133819
>>> random()
0.8282314594656393
```

Chiaramente, se provando in proprio ottenete dei numeri diversi da quelli che ho ottenuto io, niente paura.. è tutto normale, altrimenti che numeri casuali sarebbero!

Se vogliamo generare un numero intero casuale, possiamo utilizzare la funzione `randint()` alla quale bisogna specificare l'intervallo di interi da cui scegliere. La chiamata `randint(a,b)` ritorna un numero intero x tale che $a \leq x \leq b$. Continuiamo con altri esempi (assumiamo di aver già importato il modulo `random`).

```
>>> randint(1,6) #simulo il lancio di un dado a 6 facce
5
>>> randint(-5,5) #un numero casuale compreso tra -5 e +5
-3
```

Altre funzioni molto spesso utilizzate sono:

`choice(seq)` che ritorna uno a caso tra gli elementi contenuti nella sequenza `seq`. Dà errore se la sequenza in input è vuota.

`shuffle(seq)` che permuta casualmente gli elementi di una sequenza `seq` mutabile. Dà errore se la sequenza in input è vuota.

`sample(seq, k)` che ritorna una lista di lunghezza k di elementi (unici) della sequenza `seq`. Dà errore se la sequenza è vuota e/o quando k supera la lunghezza della sequenza.

```
>>> # un valore casuale dispari estratto da una tupla
>>> choice((1,3,5,7,9))
3
>>> # permuta gli oggetti di una lista
>>> elle = [1,3,5,7,9]
>>> shuffle(elle)
>>> elle
[5, 3, 9, 1, 7]
>>> # estrae quattro elementi (caratteri) da una stringa
>>> sample("abecedario",4)
['r', 'd', 'a', 'e']
>>> sample("abecedario",4)
['d', 'a', 'a', 'e']
```

Nota che nel secondo esempio della `sample` la stringa `'a'` è ripetuta due volte. Questo può succedere solo perché lo stesso carattere occorre due volte nella stringa passata come argomento. Più formalmente, l'estrazione avviene *senza rimpiazzo*.

B

Il modulo math

Il modulo `math` contiene molte funzioni e costanti matematiche di uso comune. In Tabella B.1 sono riportate alcune tra le più utilizzate. Segue un breve esempio di utilizzo del modulo.

```
>>> import math
>>> math.exp(2.0)
 7.38905609893065
>>> math.log10(100)
 2.0
>>> math.pi
 3.141592653589793
```

`acos(x)`	arcocoseno di x
`asin(x)`	arcoseno di x
`atan(x)`	arcotangente di x
`ceil(x)`	approssimazione per eccesso di x come numero in virgola mobile
`exp(x)`	e^x
`cos(x)`	coseno di x
`cosh(x)`	coseno iperbolico di x
`degrees(x)`	converte l'angolo x da radianti a gradi
`floor(x)`	approssimazione per difetto di x come numero in virgola mobile
`fabs(x)`	valore assoluto di x
`hypot(x,y)`	distanza euclidea ($\sqrt{x^2 + y^2}$)
`log(x[,b])`	logaritmo di x in base b. Se la base non è specificata, restituisce il logaritmo naturale di x
`log10(x)`	logaritmo di x in base 10
`sin(x)`	seno di x
`sinh(x)`	seno iperbolico di x
`sqrt(x)`	radice quadrata di x
`tan(x)`	tangente di x
`tanh(x)`	tangente iperbolica di x
`radians(x)`	converte l'angolo x da gradi a radianti
`e`	costante matematica e o di numero di Eulero
`pi`	costante matematica π (pi greco)

Tabella B.1: Alcune funzioni e costanti disponibili nel modulo math.

C

File e modulo sys

C.1 File di testo

Leggere e scrivere informazioni da file non è molto diverso rispetto a fare la stessa cosa dal comune terminale del sistema operativo.

Per operare correttamente su un file occorre eseguire le tre operazioni seguenti in ordine:

- aprire il file in lettura o in scrittura
- leggere o scrivere su file (a seconda della modalità con cui si è scelto di aprirlo)
- chiudere il file

Il primo passo è creare un oggetto `file` con la funzione `open()`. La sintassi è

```
oggetto_file = open (nome_file, modo)
```

dove `oggetto_file` è il riferimento ad un oggetto di tipo file, `nome_file` è la stringa con il nome del file, e `modo` è la modalità di apertura del file: `'r'` in lettura, `'w'` in scrittura.

Una volta creato e aperto il file, possiamo chiamare i suoi metodi per operare sul file. I due metodi più comuni sono `read()` per la lettura e `write()` per la scrittura. Il metodo `read()` legge il file e ne ritorna l'intero contenuto in una stringa. La funzione `write()` invece aggiunge una stringa alla fine del file.

Per chiudere un file si usa il metodo `close()`.

```
# Scrittura su di un file.
ofile = open("test_file.txt","w")
ofile.write("Questo e' un testo di prova.\n"+
   "Prova ad aprire il file e guardare cosa c'e' dentro.\n")
ofile.close()

# Lettura del contenuto di un file.
```

```
ifile = open("test_file.txt","r")
testo = ifile.read()
ifile.close()
print testo
```

In Tabella C.1 sono presentate altri utili metodi per oggetti di tipo `file`.

`readline()`	lettura di una riga (per files di testo).
`readlines()`	restuisce lintero file come lista di righe (per files di testo).
`writelines(L)`	scrive la lista L in righe nel file.

Tabella C.1: Altri metodi dell'oggetto file

C.2 Modulo sys

Il modulo sys fornisce l'accesso ad alcune variabili usate o mantenute dall'interprete, e a funzioni che interagiscono con l'interprete stesso (vedi Tabella C.2).

`argv`	La lista degli argomenti della riga di comando passati a uno script Python. `argv[0]` è il nome dello script.
`exit(arg)`	Esce da Python. `arg=0` viene considerato "terminato con successo" e qualsiasi valore diverso da zero viene considerato "terminato in modo anomalo".
`getrecursionlimit()`	Restituisce il valore attuale del limite di ricorsione. Questo parametro pu essere impostato con `setrecursionlimit()`.
`setrecursionlimit(limite)`	Imposta la massima profondità della ricorsione. Il massimo limite possibile dipende dalla piattaforma in uso.
`stdin`	File oggetto corrispondente allo standard input.
`stdout`	File oggetto corrispondente allo standard output.
`stderr`	File oggetto corrispondente allo standard error.
`version`	Una stringa contenente la versione dell'interprete Python, insieme alle informazioni supplementari del numero di compilazione e del compilatore utilizzato.

Tabella C.2: Variabili e funzioni principali del modulo sys

FABIO AIOLLI
è Ricercatore Universitario
e Professore Aggregato
presso il Dipartimento
di Matematica dell'Università
di Padova dal 2006.
Ha una laurea in Scienze
dell'Informazione
e una laurea specialistica
in Tecnologie Informatiche,
conseguite presso
l'Università di Pisa.
Sempre a Pisa ha conseguito
il titolo di Dottore di Ricerca
in Informatica.
La sua attività di
ricerca si colloca
nell'area di Intelligenza
Artificiale, in particolare
nell'Apprendimento
Automatico e nella Pattern
Recognition.
Ha vinto due competizioni
di ricerca internazionali
in ambito Apprendimento
Automatico.

È coinvolto in diverse
collaborazioni a progetti
sia italiani che internazionali
e co-autore di numerosi
articoli su riviste e conferenze
scientifiche internazionali.
Ha esperienza didattica
pluriennale.
Dal 2005 insegna in corsi
di laurea triennale e
magistrale dell'Università
di Padova.

Saper programmare un computer è una capacità oramai necessaria a chiunque voglia addentrarsi nello studio delle scienze.
Da questo punto di vista, il linguaggio Python sta diventando sempre più popolare per la programmazione in ambito scientifico.
Python è un linguaggio di alto livello che usa il paradigma della programmazione orientata agli oggetti (OOP).
È un linguaggio semplice e, a differenza di altri linguaggi di programmazione, non richiede l'apprendimento di tecnicismi che spesso distolgono dal vero scopo: imparare a programmare!

Questo libro propone una guida completa alla programmazione, fornendo i principi fondamentali utili ad esprimere con un algoritmo la soluzione ad un problema specifico.
Nel libro sono schematizzate le problematiche ricorrenti della programmazione e per queste vengono presentati esempi e soluzioni corrette, efficienti ed eleganti.
Il volume approfondisce i fondamentali della programmazione: la selezione e la iterazione, la ricorsione, le strutture dati complesse quali liste concatenate ed alberi e gli algoritmi di ricerca, ordinamento e il backtracking.
I concetti teorici presentati vengono poi ampiamente applicati a problemi scientifici reali e ai giochi.

ISBN 978-88-7488-678-4
9 788874 886784

Euro 15,00

SOCIETÀ EDITRICE ESCULAPIO
www.editrice-esculapio.it